Sławomir SAMOLEJ

Programowanie współbieżnych i rozproszonych aplikacji czasu rzeczywistego dla Linuxa – receptury

Skrypt dla studentów studiów informatycznych

Wydano za zgodą Rektora

R e c e n z e n t
dr inż. Rafał MRÓWKA – AGH

R e d a k t o r n a c z e l n y
Wydawnictw Politechniki Rzeszowskiej
dr hab. inż. Lesław GNIEWEK, prof. PRz

R e d a k t o r
Marzena TARAŁA

S k ł a d i ł a m a n i e t e k s t u
p r o j e k t o k ł a d k i
Joanna MIKUŁA

Skrypt do wykładów, ćwiczeń laboratoryjnych i projektowych z przedmiotów „programowanie współbieżne i rozproszone", „programowanie i projektowanie systemów czasu rzeczywistego" dla studentów studiów informatycznych

systemy czasu rzeczywistego, Linux, programowanie

© Copyright by Oficyna Wydawnicza Politechniki Rzeszowskiej, Rzeszów 2024

Wszelkie prawa autorskie i wydawnicze zastrzeżone. Każda forma powielania oraz przenoszenia na inne nośniki bez pisemnej zgody Wydawcy jest traktowana jako naruszenie praw autorskich, z konsekwencjami przewidzianymi w *Ustawie o prawie autorskim i prawach pokrewnych* (Dz.U. z 2018 r., poz. 1191 t.j.). Autor i Wydawca dołożyli wszelkich starań, aby rzetelnie podać źródło zamieszczonych ilustracji oraz dotrzeć do właścicieli i dysponentów praw autorskich. Osoby, których nie udało się ustalić, są proszone o kontakt z Wydawnictwem.

e-ISBN 978-83-7934-730-8
p-ISBN 978-83-7934-726-1

Oficyna Wydawnicza Politechniki Rzeszowskiej
al. Powstańców Warszawy 12, 35-959 Rzeszów
https://oficyna.prz.edu.pl

Ark. wyd. 6,63. Ark. druk. 6,50. Wydrukowano w czerwcu 2024 r.
Drukarnia Oficyny Wydawniczej Politechniki Rzeszowskiej, al. Powstańców Warszawy 12, 35-959 Rzeszów
Zam. nr 27/24

Mojej żonie Renacie

Spis treści

Wprowadzenie ... 7
O czym jest ta książka? ... 7
Wymagania wstępne i styl wprowadzania treści ... 8
Treści omawiane w poszczególnych rozdziałach ... 8
Charakterystyka systemów czasu rzeczywistego ... 10
Znaczenie terminowości obliczeń ... 11
Cechy systemów czasu rzeczywistego ... 11
Zastosowanie systemów czasu rzeczywistego ... 12
System czasu rzeczywistego a system operacyjny czasu rzeczywistego ... 12
System czasu rzeczywistego jako platforma programowa ... 14
Literatura uzupełniająca ... 15
Ćwiczenia ... 16
Receptura 1. – aplikacja wielowątkowa ... 17
Pierwsza aplikacja wielowątkowa ... 17
Scenariusz działania aplikacji ... 19
Literatura uzupełniająca ... 20
Ćwiczenia ... 20
Receptura 2. – cykliczna aplikacja wielowątkowa ... 21
Interfejs sygnałów ... 21
Wielowątkowy program obsługujący sygnały ... 22
Scenariusz działania aplikacji ... 23
Zegary i budziki czasu rzeczywistego ... 24
Wielowątkowy program sterowany zdarzeniami czasowymi ... 25
Scenariusz działania aplikacji ... 28
Literatura uzupełniająca ... 28
Ćwiczenia ... 29
Receptura 3. – szeregowanie cyklicznych wątków ... 31
Problem szeregowania w systemach czasu rzeczywistego ... 31
Współczynnik wykorzystania procesora ... 32
Algorytm planowania ze stałymi priorytetami ... 34
Wielowątkowy program z algorytmem planowania ze stałymi priorytetami ... 35
Przyłączanie wątków do rdzenia mikroprocesora ... 40
Ustalanie dla wątków parametrów szeregowania ... 40
Technika uruchamiania aplikacji ... 42
Funkcje spoza standardu POSIX zastosowane w aplikacji ... 42
Literatura uzupełniająca ... 43
Ćwiczenia ... 43
Receptura 4. – techniki lokalnej komunikacji ... 45
Techniki komunikacji lokalnej stosowane w systemach czasu rzeczywistego ... 45
Współdzielenie pamięci ... 45

Problem wzajemnego wykluczania ... 46
Inwersja priorytetów ... 49
Protokół dziedziczenia priorytetów ... 50
Warunek szeregowalności wątków komunikujących się za pomocą zmiennej dzielonej ... 52
Wielowątkowy program czasu rzeczywistego z wymianą danych przez zmienną dzieloną ... 53
Technika uruchamiania aplikacji ... 56
Kolejki komunikatów ... 57
Wielowątkowy program czasu rzeczywistego z wymianą danych przez kolejkę ... 58
Technika uruchamiania aplikacji ... 63
Techniki komunikacji międzyprocesowej – uzupełnienie ... 64
Literatura uzupełniająca ... 64
Zadania ... 65
Receptura 5. – techniki komunikacji sieciowej ... 67
Dwa podstawowe warianty komunikacji sieciowej ... 67
Aplikacja sieciowa opierająca się na protokole UDP ... 68
Badanie opóźnień w komunikacji ... 74
Technika uruchamiania aplikacji ... 74
Aplikacja sieciowa opierająca się na protokole TCP ... 74
Technika uruchamiania aplikacji ... 79
Uwagi o interfejsie sieciowym jako medium komunikacji ... 79
Literatura uzupełniająca ... 80
Zadania ... 80
Receptura 6. – techniki wykrywania uszkodzeń ... 82
Wykrywanie przekroczenia ograniczeń czasowych ... 83
Technika uruchamiania aplikacji ... 86
Przeterminowanie dostępu do sekcji krytycznej ... 87
Technika uruchamiania aplikacji ... 91
Wykrywanie blokujących się sekcji krytycznych ... 91
Technika uruchamiania aplikacji ... 95
Literatura uzupełniająca ... 95
Zadania ... 95
Podsumowanie ... 97
Bibliografia ... 99
Dodatki ... 101

Wprowadzenie

Zawsze ceniłem sobie książki pokazujące zasady programowania na konkretnych przykładach. W podobny sposób staram się również prowadzić swoje zajęcia dydaktyczne. Kiedy więc przyszło mi przygotować niniejszy skrypt, postanowiłem podążyć tą samą ścieżką.

Skrypt, który oddaję Czytelnikowi do rąk, zawiera te wiadomości o programowaniu systemów czasu rzeczywistego, które chciałbym posiadać, gdybym miał się wrócić w czasie na początek zdobywania moich doświadczeń. Jest on owocem kilkudziesięciu lat uczestniczenia w pracach nad projektowaniem i programowaniem aplikacji czasu rzeczywistego.

Zdaję sobie sprawę, że w wielu miejscach zagadnienia potraktowane są skrótowo. W samą treść opracowania mogły wkraść się błędy, a zaproponowane programy mogły mieć bardziej spójny i profesjonalny charakter. Zachęcam zatem Czytelników do wysyłania ewentualnych uwag i korekt na adres: ssamolej@prz.edu.pl.

O czym jest ta książka?

Komputerowe systemy czasu rzeczywistego mają ponadpięćdziesięcioletnią historię. W kontekście współczesnej klasyfikacji systemów informatycznych stanowią podzbiór systemów cyberfizycznych. Służą nam na co dzień, chociaż nie zawsze jesteśmy tego świadomi. Ich podstawową cechą jest wykonywanie obliczeń, które muszą się zakończyć na czas. Spóźnione obliczenia mogą skutkować utratą majątku lub zdrowia ludzi, choć czasami mogą powodować tylko irytację użytkowników. To komputerowym systemom czasu rzeczywistego powierzamy nasze życie, podróżując samolotami, kolejami, statkami czy samochodami. Wspomagają one również wykonywanie wielu procesów technologicznych w fabrykach lub elektrowniach, można je spotkać w urządzeniach medycznych.

Na temat komputerowych systemów czasu rzeczywistego powstała obszerna literatura, szczególnie dotycząca metod ich projektowania. Znacznie mniejsza liczba publikacji skupia się na programowaniu tego typu systemów. Celem niniejszego skryptu jest zwięzłe przeprowadzenie Czytelnika przez metodę programowania systemów czasu rzeczywistego z zastosowaniem bibliotek dostępnych w systemach operacyjnych Linux.

Wymagania wstępne i styl wprowadzania treści

Proponowany kurs nie ma na celu omówienia wszystkich możliwych wariantów programowania systemów czasu rzeczywistego. Zamiast tego pokazuje pewną ścieżkę, która pozwala na zrozumienie najważniejszych programowych składników i zasad ich integracji w poprawnie skonstruowane aplikacje. W pracy celowo zostaną pominięte szczegółowe informacje na temat funkcji, które zastosowano w przykładach. Wiedzę tę można uzupełnić z dostępnych w Internecie darmowych dokumentacji.

Wszystkie przykłady programów będą napisane w języku C z zastosowaniem standardowych bibliotek wbudowanych w systemy Linux. Większość oprogramowania na poziomie kodu źródłowego będzie zgodna ze standardem POSIX, powinna więc, po przetworzeniu przez odpowiednie kompilatory, bez wnoszenia dodatkowych modyfikacji dać się uruchamiać na systemach operacyjnych klasy UNIX. Przyjęto, że Czytelnik zna podstawy programowania w języku C oraz potrafi przygotować i uruchomić plik wykonywalny na podstawie otrzymanego kodu źródłowego. W treści pracy znajdą się również odwołania do elementarnej wiedzy z zakresu architektury systemów komputerowych oraz systemów operacyjnych, w tym do podstawowych zagadnień programowania współbieżnego.

Każdy z rozdziałów skryptu, z wyjątkiem pierwszego, jest skonstruowany jako receptura rozwiązująca pewien problem związany z tworzeniem aplikacji czasu rzeczywistego. Rozwiązaniem problemu jest zawsze pełny, skomentowany w treści podrozdziałów, przykładowy kod aplikacji. Przestudiowanie kolejnych receptur powinno pozwolić Czytelnikowi na samodzielne konstruowanie tego typu aplikacji. Przedłożone programy można traktować jako podstawy do konstruowania własnych rozwiązań. Są one dostępne na stronie: http://ssamolej.kia.prz.edu.pl/Skrypt_Linux_RTS/.

Każdy rozdział skryptu zawiera sekcję „literatura uzupełniająca”. Wskazane są tam publikacje, które mogą posłużyć do rozszerzenia wiedzy Czytelnika dotyczącej omawianej receptury. W niektórych rozdziałach sekcja ta wskazuje również publikacje wykraczające poza zakres niniejszego skryptu.

Końcowa sekcja każdego rozdziału zawiera propozycję kilku przykładowych ćwiczeń, które Czytelnik może samodzielnie przeprowadzić w celu ugruntowania bądź rozszerzenia nabytej wiedzy.

Treści omawiane w poszczególnych rozdziałach

Pierwszy rozdział książki jest krótkim wprowadzeniem do komputerowych systemów czasu rzeczywistego. Zwrócono tam uwagę na ich szczególne właściwości oraz zastosowania.

Częstą sytuacją, z jaką spotykają się twórcy systemów czasu rzeczywistego, jest konieczność reagowania na współbieżnie występujące w otoczeniu zdarzenia. Powszechnym rozwiązaniem takiego problemu jest konstruowanie oprogramowania, które również jest współbieżne i stanowi pewnego rodzaju odbicie rzeczywistości. W związku z tym pierwsza receptura pokazuje, jak skonstruować współbieżną wielowątkową aplikację.

Aplikacja czasu rzeczywistego jest często konstruowana jako zbiór współbieżnych wątków cyklicznie wznawiających swoje obliczenia. Przygotowanie tak działającego oprogramowania jest przedmiotem receptury numer dwa. Do rozwiązania problemu zastosowano interfejsy sygnałów i budzików.

Od systemów czasu rzeczywistego oczekuje się, między innymi, wykonywania obliczeń przez wszystkie wątki w zadanych ograniczeniach czasowych. Receptura trzecia pokazuje, jak rozwiązać problem terminowości wykonywania zadań czasu rzeczywistego uruchomionych na wybranym rdzeniu procesora. Wymaga to analizy czasowej aplikacji, przeprogramowania algorytmu szeregowania wątków oraz przypisania do nich priorytetów.

Lokalna komunikacja we współbieżnych systemach czasu rzeczywistego wymaga szczególnej uwagi. Okazuje się, że może ona wprowadzać dodatkowe opóźnienia w obliczeniach. Receptura czwarta pokazuje, jak skomunikować wątki aplikacji i uwzględnić wpływ komunikacji na terminowość obliczeń. Rozwiązanie tego problemu wymaga specyficznego zastosowania systemowych mechanizmów wzajemnego wykluczania oraz kolejek.

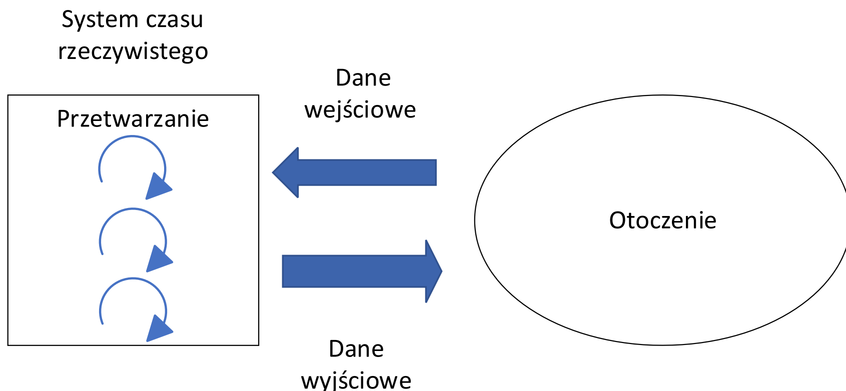
Poważnym wyzwaniem w konstruowaniu aplikacji czasu rzeczywistego jest komunikacja przez sieć komputerową. Można się w tym celu posłużyć specjalizowanymi urządzeniami i protokołami sieciowymi. W skrypcie pokazano jednak inne podejście, w którym można zastosować „standardowe" urządzenia i protokoły sieciowe. Pokazano, że sieciową komunikację czasu rzeczywistego udaje się implementować w umiarkowanie obciążonych standardowych sieciach komputerowych, przy czym oprogramowanie komunikacyjne powinno mieć możliwość mierzenia opóźnień w komunikacji, a poszczególne węzły sieci powinny posiadać zsynchronizowane zegary. Receptura piąta pokazuje zastosowania protokołów UDP i TCP w aplikacjach czasu rzeczywistego.

Ostatnia, szósta receptura opisuje metody wykrywania nieprzewidywalnych opóźnień w czasie wykonywania oprogramowania czasu rzeczywistego. Na pozór precyzyjnie zaprojektowany system nie powinien mieć takich opóźnień. Praktyka utrzymywania oprogramowania czasu rzeczywistego pokazuje jednak, że warto rozważyć takie programistyczne rozszerzenia.

Charakterystyka systemów czasu rzeczywistego

Istotny dział współczesnej informatyki stanowią systemy cyberfizyczne (ang. Cyber-Physical Systems). Rozumie się je jako systemy agregujące urządzenia fizyczne wpływające na otoczenie człowieka z jednostkami obliczeniowymi (komputerami). Systemy stosują różnego rodzaju czujniki do zbierania informacji o swoim otoczeniu, mogą również wykorzystywać człony wykonawcze (siłowniki) do modyfikacji świata fizycznego. Tego typu systemy stosuje się w sterowaniu i monitorowaniu procesów przemysłowych, urządzeniach medycznych, autonomicznych pojazdach, inteligentnych budynkach i miastach, a także systemach robotyki. Wśród takich systemów znaczący podzbiór stanowią systemy czasu rzeczywistego (ang. Real-Time Systems).

Systemy czasu rzeczywistego to systemy cyberfizyczne, których odpowiednie działanie zależy nie tylko od wytworzenia poprawnych logicznych odpowiedzi, ale również od fizycznego czasu, kiedy te obliczenia są produkowane.



Rysunek 1. Ogólny schemat przetwarzania systemu czasu rzeczywistego

Typowy system czasu rzeczywistego konstruuje się w celu sterowania, nadzorowania bądź terminowego reagowania na zdarzenia lub procesy występujące w jego otoczeniu. Ogólny schemat takiego systemu pokazano na rys. 1. Dane wejściowe mogą być rozumiane jako zdarzenia lub strumień informacji dostarczane do systemu w sposób synchroniczny lub asynchroniczny. Jeśli system ma reago-

wać na zdarzenia (ang. Event-Triggered System), to definiuje się dla niego maksymalne opóźnienie, z jakim na wskazane zdarzenie ma zareagować. Jeśli zadaniem systemu jest ciągłe śledzenie i modyfikowanie otoczenia, to projektuje się go w postaci aplikacji działających cyklicznie, które potencjalnie w nieskończoność: odczytują, przetwarzają i zwracają dane. Częstotliwość cyklu przetwarzania jest wymuszana przez otoczenie (np. dynamika sterowanego procesu). Kolejne cykle obliczeniowe są wznawiane pod wpływem budzików systemowych, a systemy tego typu są nazywane systemami wyzwalanymi czasowo (ang. Time-Triggered Systems).

Znaczenie terminowości obliczeń

Od systemów czasu rzeczywistego oczekuje się w równej mierze poprawnych logicznie wyników działania i terminowego ich dostarczania. Przykładowo, system reagujący na zdarzenia musi po otrzymaniu sygnału alarmowego w ciągu zadanego nieprzekraczalnego odcinka czasu uruchomić daną procedurę. W przeciwnym wypadku może to grozić utratą mienia lub zdrowia ludzi. W systemach wyzwalanych czasowo, jeśli wznowienia następują w precyzyjnie wyliczonych momentach czasu i przetwarzanie zawsze kończy się w zadanym oknie czasowym, to system zapewnia np. wysoką jakość działania algorytmów sterujących. W przeciwnym wypadku roboty lub autopiloty mogą działać nieprawidłowo lub w ogóle przestać pracować.

Cechy systemów czasu rzeczywistego

Podstawowymi cechami systemów czasu rzeczywistego są:

- ciągłość działania (system pracuje bez przerwy od uruchomienia do wycofania);
- zależność od otoczenia (system jest rozważany zawsze w kontekście pewnego, zwykle statycznego otoczenia);
- współbieżność (naturalna współbieżność otoczenia narzuca współbieżną strukturę systemu);
- przewidywalność (bez względu na nieprzewidywalność otoczenia, wynik pracy systemu powinien być przewidywalny);
- punktualność (wyniki działania systemu powinny być dostarczane terminowo).

Biorąc pod uwagę terminowość reakcji na stan otoczenia, systemy czasu rzeczywistego można podzielić na trzy klasy: systemy o twardych wymaganiach czasowych (ang. Hard Real-Time Systems), systemy o miękkich wymaganiach czasowych (ang. Soft Real-Time Systems), systemy o solidnych wymaganiach czasowych (ang. Firm Real-Time Systems). Systemy o twardych wymaganiach cza-

sowych muszą ściśle spełniać postawione dla nich ograniczenia czasowe. Oznacza to, że system tego typu zawsze musi wygenerować odpowiedź zanim upłynie ustalony przedział czasu lub niekiedy odpowiedź musi się pojawić w ściśle określonej chwili. Od systemów o miękkich wymaganiach czasowych wymaga się jedynie zachowania średniego czasu odpowiedzi. Systemy o solidnych wymaganiach czasowych zakładają możliwość przekroczenia ograniczeń czasowych, ale prowadzą również swego rodzaju samoocenę jakości. Przykładowo, dla pewnych systemów przekroczenie ograniczeń czasowych kilka razy na godzinę jest do przyjęcia, częstsze zaś oznacza zbyt dużą degenerację systemu i konieczność jego rekonfiguracji lub zatrzymania.

Zastosowanie systemów czasu rzeczywistego

Okazuje się, że systemy czasu rzeczywistego można spotkać w wielu gałęziach przemysłu i rozrywki, takich jak: sterowanie instalacjami chemicznymi i energetycznymi, sterowanie procesami przemysłowymi, zarządzanie ruchem kolejowym, aplikacje w samochodach, autopiloty, urządzenia telekomunikacyjne, urządzenia medyczne, automatyka przemysłowa, robotyka, misje kosmiczne, urządzenia elektroniczne codziennego użytku, multimedia, zabawki i wirtualna rzeczywistość. W części z wymienionych aplikacji systemów czasu rzeczywistego oczekuje się bardzo rygorystycznego przestrzegania ograniczeń czasowych (np. automatyka, robotyka, autopiloty, urządzenia medyczne), w innych zaś wystarcza spełnienie solidnych lub miękkich wymagań czasowych (np. multimedia, zabawki, wirtualna rzeczywistość).

System czasu rzeczywistego a system operacyjny czasu rzeczywistego

Pojęcia „system czasu rzeczywistego" i „system operacyjny czasu rzeczywistego" są często mylone. System czasu rzeczywistego jest pojęciem bardziej ogólnym i odnosi się do wszystkich systemów i aplikacji komputerowych spełniających omówione wcześniej wymagania, czyli zapewnianie poprawności logicznej i terminowości wykonywanych obliczeń. Z kolei system operacyjny czasu rzeczywistego jest wyspecjalizowanym systemem operacyjnym, który udostępnia usługi umożliwiające uruchamianie na nim aplikacji (systemów) czasu rzeczywistego.

Samo zastosowanie systemu operacyjnego czasu rzeczywistego usprawnia tworzenie złożonych aplikacji. Ułatwione staje się zarządzanie pamięcią, integracja z urządzeniami zewnętrznymi oraz tworzenie aplikacji wykorzystujących komunikację międzyprocesową i sieciową.

Szczególną cechą systemów operacyjnych czasu rzeczywistego jest udostępnienie użytkownikowi możliwości uruchamiania lub wznawiania podprogramu po pojawieniu się w systemie pewnych zdarzeń. Niezależnie od obciążenia systemu przedział czasowy od pojawienia się zdarzenia do uruchomienia danego podprogramu jest przewidywalny i stały. Tylko wtedy można przyjąć, że przyszłe oprogramowanie będzie spełniać założone ograniczenia czasowe. Od systemu operacyjnego czasu rzeczywistego oczekuje się również umożliwienia uruchamiania aplikacji współbieżnych oraz możliwości decydowania przez programistę o priorytetach i algorytmach szeregowania uruchomionych procesów i wątków.

Docelowym przeznaczeniem aplikacji czasu rzeczywistego jest ich uruchomienie pod nadzorem systemu operacyjnego czasu rzeczywistego na określonej platformie sprzętowej. Integracja systemu operacyjnego czasu rzeczywistego z taką platformą nie jest, zdaniem autora, zagadnieniem trywialnym. Nie polega ona jedynie na „zainstalowaniu" systemu operacyjnego na danym systemie mikroprocesorowym. Prawidłowe osadzenie takiego systemu operacyjnego powinno zostać potwierdzone wieloma ustandaryzowanymi testami stwierdzającymi przewidywalność opóźnień uruchamiania podprogramów pod wpływem zdarzeń, niezależnie od obciążenia systemu. Osoby zainteresowane szybszym wdrożeniem aplikacji czasu rzeczywistego (z pominięciem integracji systemu operacyjnego z platformą sprzętową) powinny się zainteresować zakupem gotowego zestawu złożonego z komputera z zainstalowanym systemem operacyjnym czasu rzeczywistego, który wspomniane testy przeszedł i jest certyfikowany przez dostawcę danego systemu. Jest to praktyka dosyć powszechna, a wokół dojrzałych systemów operacyjnych czasu rzeczywistego jest często oferowany pewien zbiór platform mikroprocesorowych do wytwarzania prototypów oprogramowania.

Jak już wspomniano, materiał zaprezentowany Czytelnikowi w skrypcie zawiera programy przetestowane w systemie operacyjnym Linux. Można je uruchomić we wszystkich współczesnych wersjach Linuxa, w tym nieopisywanych jako systemy operacyjne czasu rzeczywistego. Należy jednak zaznaczyć, że w pełni przewidywalne ze względu na ograniczenia czasowe wykonywanie tych programów jest możliwe tylko pod nadzorem systemu operacyjnego Linux czasu rzeczywistego. System Linux czasu rzeczywistego, w odróżnieniu od jego standardowej wersji, stosuje inny moduł szeregowania zadań, który będzie wspierał terminowość wykonywania obliczeń, ale z punktu widzenia systemów ogólnego przeznaczenia będzie w sposób mniej zrównoważony zarządzał zasobami systemu mikroprocesorowego. Zadania czasu rzeczywistego o wyższym priorytecie w systemie Linux czasu rzeczywistego zawsze będą natychmiast wywłaszczać zadania „mniej ważne", podczas gdy w systemach ogólnego przeznaczenia wywłaszczenia te mogły być odraczane. Taka zmiana metod zarządzania zadaniami pozwala na wdrożenie zasad spełniania ograniczeń czasowych dla aplikacji.

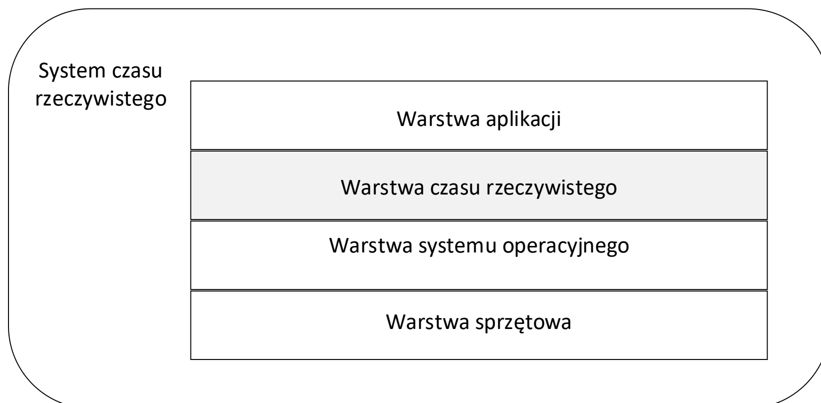
Posługując się pojęciem systemu operacyjnego Linux czasu rzeczywistego, należy pamiętać, że jest to w zasadzie rodzina systemów operacyjnych zaproponowanych przez wiele instytucji i projektów badawczych. Najpopularniejsze z nich to „Xsenomai", „RTLinux", „RTAI", „Real-Time Ubuntu". Warto również wiedzieć, że oprócz rozwiązań opierających się na systemach Linux można do tworzenia aplikacji czasu rzeczywistego zastosować komercyjne systemy operacyjne czasu rzeczywistego, takie jak VxWorks czy QNX. Co ciekawe, programy zaproponowane w niniejszym skrypcie po niewielkich modyfikacjach mogłyby być wykonywane pod nadzorem tych systemów operacyjnych.

System czasu rzeczywistego jako platforma programowa

Początek rozwoju systemów czasu rzeczywistego tworzonych na podstawie systemów operacyjnych sięga lat 70. XX wieku. Systemy operacyjne już wówczas pozwalały na konstruowanie oprogramowania w postaci współbieżnie wykonywanych procesów. Zwrócono wówczas uwagę, że jeśli dysponuje się systemem operacyjnym, to współbieżne rzeczywiste otoczenie systemu może zostać odwzorowane jako współbieżna aplikacja. Zaproponowano wtedy również pierwsze zasady szeregowania współbieżnych procesów, pozwalające na ich terminowe wykonywanie. Z czasem opracowano metody zarządzania zbiorem procesów czasu rzeczywistego, które podlegają synchronizacji i komunikacji. W miarę rozwoju sprzętu komputerowego w systemach operacyjnych wprowadzono nowy rodzaj jednostek obliczeniowych – wątki. W skrócie, pojedynczy proces z punktu widzenia systemu operacyjnego może utrzymywać własne współbieżne jednostki obliczeniowe.

Dalsze rozważania prowadzone w tym skrypcie skoncentrują się na zasadach tworzenia aplikacji czasu rzeczywistego złożonych z procesu uruchomionego na platformie systemu operacyjnego, w którym mogą zostać powołane współbieżne wątki. Uzupełnieniem takiej architektury systemu będzie warstwa komunikacji pomiędzy wątkami, procesami, a także komputerami połączonymi w sieć komputerową.

Częstym zadaniem stawianym przed inżynierami oprogramowania systemów czasu rzeczywistego jest udostępnienie programowych usług czasu rzeczywistego na potrzeby danych aplikacji. Pomiędzy programową warstwą systemu operacyjnego a warstwą aplikacji pojawia się wtedy kolejna programowa warstwa czasu rzeczywistego. Jest ona z jednej strony szkieletem aplikacji, a z drugiej strony – dostawcą mechanizmów czasu rzeczywistego zaimplementowanych w systemie operacyjnym (por. rys. 2.).



Rysunek 2. Umiejscowienie warstwy czasu rzeczywistego w systemie

Z doświadczenia autora tego skryptu wynika, że zadaniem inżyniera oprogramowania czasu rzeczywistego jest przygotowanie zbioru zadań (wątków) czasu rzeczywistego, które będą wykonywać dostarczone (od specjalistów) algorytmy. Konfiguracja oprogramowania powinna zapewnić efektywną komunikację pomiędzy wybranymi wątkami na poziomie pojedynczej aplikacji. Należy również przewidzieć możliwość komunikacji pomiędzy różnymi aplikacjami uruchamianymi na tym samym lub innym systemie komputerowym (komunikacja przez sieć komputerową). Oprogramowanie czasu rzeczywistego powinno zostać tak skonstruowane, aby wszystkie jego składniki mogły zakończyć swoje obliczenia przed założonymi terminami.

Literatura uzupełniająca

Systemy czasu rzeczywistego przez ostatnie kilkadziesiąt lat wyodrębniły się jako osobna klasa systemów komputerowych. Na temat ich wytwarzania powstało bardzo wiele cennych publikacji. Czytelnikowi zainteresowanemu poszerzeniem przekrojowej wiedzy na temat tego typu systemów można polecić publikacje: [KS2022], [Coo2019], [Erc2019] oraz [GCL2016]. Cennymi przeglądowymi publikacjami nakierowanymi zarówno na projektowanie, jak i programowanie aplikacji czasu rzeczywistego są: [BW2009], [Szm2000], [Szm2001], [Gom2000], [Dou1999]. Szczególną pozycją literatury wartą wspomnienia jest [HS1991]. Jako jedna z niewielu przeprowadza dyskusję nad architekturą sprzętu komputerowego, który powinien być konstruowany na potrzeby systemów czasu rzeczywistego. Osoby zainteresowane najnowszymi osiągnięciami naukowymi z dziedziny systemów czasu rzeczywistego mogą śledzić publikacje w czasopiśmie „Real-Time Systems, The International Journal of Time-Critical Computing Systems” publikowanym przez wydawnictwo Springer.

Ćwiczenia

1. Podaj definicję systemu czasu rzeczywistego.
2. Jaka jest różnica pomiędzy systemem czasu rzeczywistego a systemem operacyjnym czasu rzeczywistego?
3. Dlaczego terminowość obliczeń ma znaczenie w systemach czasu rzeczywistego?
4. Gdzie są stosowane komputerowe systemy czasu rzeczywistego?
5. Na podstawie danych z Internetu wyszukaj aktualnie stosowane i wspierane systemy operacyjne czasu rzeczywistego.

Receptura 1.
– aplikacja wielowątkowa

System czasu rzeczywistego zwykle bywa odzwierciedleniem otoczenia, które z natury jest współbieżne. Podstawową umiejętnością tworzenia współbieżnych aplikacji czasu rzeczywistego jest więc tworzenie aplikacji wielowątkowych.

Pierwsza aplikacja wielowątkowa

Na listingu 1. pokazano pierwszą przykładową aplikację wielowątkową. Kompilację takiego programu można wykonać poleceniem: `gcc -o nazwa_aplikacji nazwa_pliku_źródłowego.c -lpthread -lrt`.

Listing 1. Aplikacja tworząca dwa wątki potomne pobierające parametry wywołania i zwracające wartości całkowite

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_and_increment(void *arg) {
    int num = *((int *)arg);
    printf("Wartosc: %d\n", num);
    num++;
    int *result = malloc(sizeof(int));
    *result = num;
    pthread_exit(result);
}

int main() {
    pthread_t thread1, thread2;
    int num1 = 5, num2 = 10;
    int *res1, *res2;

    pthread_create(&thread1, NULL,
                   print_and_increment, &num1);
    pthread_create(&thread2, NULL,
                   print_and_increment, &num2);
```

```
    pthread_join(thread1, (void **)&res1);
    pthread_join(thread2, (void **)&res2);

    printf("Wątek 1 zwrócił: %d\n", *res1);
    printf("Wątek 2 zwrócił: %d\n", *res2);
    free(res1);
    free(res2);

    return 0;
}
```

Cała przykładowa aplikacja składa się z trzech wątków: wątku głównego (funkcji `main`) oraz dwóch wątków potomnych. Utworzenie wątku potomnego odbywa się przez wywołanie funkcji `pthread_create`. Funkcja ta przyjmuje cztery parametry. Pierwszy to uchwyt do wątku typu `pthread_t`. Jest to struktura danych identyfikująca powstały wątek w naszej aplikacji i systemie operacyjnym. Drugi parametr jest przeznaczony na przekazanie tak zwanych atrybutów wątku. Znaczenie tej struktury danych zostanie omówione później. Trzeci parametr jest wskaźnikiem na funkcję, która ma być wykonywana po uruchomieniu wątku. Innymi słowy, kiedy wątek obliczeniowy zostanie utworzony, to kontrolę nad nim przejmuje wskazana funkcja. Funkcja ta musi zwracać wartość `(void*)` oraz przyjmować argument wywołania `(void*)`. Według zasad tworzenia programów zgodnych ze standardem języka C pozwala to na „podstawianie" pod te wartości innych typów danych. Ostatnim argumentem funkcji `pthread_create` jest pojedynczy argument, który można przekazać do funkcji wątku.

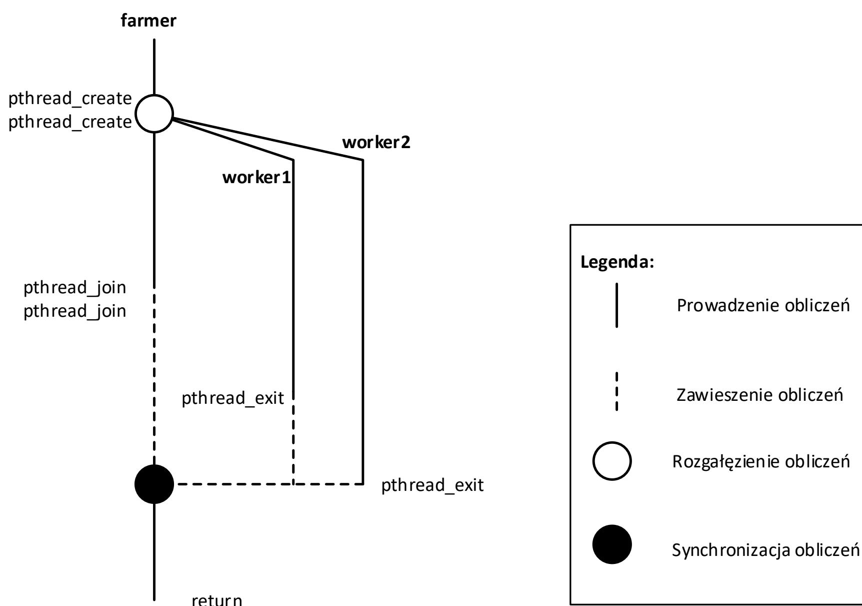
W programie z listingu 1. główny wątek (funkcja `main`) po utworzeniu dwóch wątków potomnych i przekazaniu każdemu jako parametr wywołania jednej liczby całkowitej wywołuje dwukrotnie funkcję `pthread_join`. Funkcja ta zawiesza wykonywanie wątku głównego w oczekiwaniu na zakończenie obliczeń przez wątek potomny. Parametrami tej funkcji są uchwyt do wątku, na którego zakończenie się czeka, oraz wartość zwracana przez kończący swoje obliczenia wątek. Co do zasady funkcję `pthread_join` powinno się wywołać tyle razy, ile utworzono wątków potomnych. Główna funkcja programu „czeka" wtedy na zakończenie obliczeń przez wszystkie utworzone wątki, a następnie kontynuuje obliczenia wątku głównego. „Rodzina" wywołań funkcji `pthread_join` jest więc z punktu widzenia programowania współbieżnego punktem synchronizacyjnym, w którym kończą wykonywanie wszystkie wątki potomne. Funkcja `pthread_join`, jak już wspomniano, pozwala również na przechwycenie danych od kończących swoje obliczenia wątków potomnych. W przykładowej aplikacji wartości zwracane przez wątki są przechwytywane odpowiednio przez zmienne wskaźnikowe `*res1, *res2`.

Do obu utworzonych wątków przekazano taką samą funkcję sterującą o nazwie `print_and_increment`. Każdy wątek wykonuje własną kopię tej funkcji. Świadczą o tym różne wyniki działania wątków, do których podczas inicjalizacji przekazano różne argumenty. Nic nie stoi na przeszkodzie, aby każdy z tworzonych wątków wykonywał osobno zdefiniowaną funkcję wątku.

Funkcja p `print_and_increment` przyjmuje wskaźnik do liczby całkowitej jako argument (`void *arg`). Następnie przechwytuje ten argument do zmiennej typu całkowitego (`int num = *((int *)arg);`), wypisuje wartość tej liczby, a potem zwiększa ją o jeden. Funkcja alokuje pamięć dla nowej zmiennej całkowitej (`int *result = malloc(sizeof(int));`), przypisuje do niej zwiększoną wartość liczby i zwraca wskaźnik do tej zmiennej. Funkcja kończy swoje działanie wywołaniem `pthread_exit(result);`, co powoduje zakończenie wątku i zwrócenie wartości przez wątek.

Scenariusz działania aplikacji

Struktura wykonywania aplikacji pokazanej na listingu 1. w teorii systemów współbieżnych jest nazywana „farmer-worker”.



Rysunek 3. Przykładowy diagram wykonywania aplikacji pokazanej na listingu 1.

Główny wątek (farmer) „rozdziela” prace wątkom potomnym (worker), a następnie oczekuje na zakończenie przez nich obliczeń. Przykładowy scenariusz działania omawianej aplikacji pokazano na rys. 3.

Główny wątek programu rozpoczyna swoje obliczenia, a następnie powołuje do życia dwa wątki potomne. Po pewnym czasie od utworzenia wątków potomnych zawiesza swoje obliczenia (linia przerywana) i oczekuje na zakończenie przez nie obliczeń. Kiedy ostatni z wątków zakończy swoje obliczenia, to wznawiany jest wątek główny. Owale w diagramie wskazują punkty rozgałęziania bądź synchronizacji (oczekiwania na zakończenie działania wszystkich wątków).

Literatura uzupełniająca

Literatura dotycząca programowania w systemie Linux, w tym współbieżnego, jest bardzo bogata. Warto zwrócić uwagę na pozycję [MS2007]. Przykłady programów wielowątkowych pokazanych w tej pracy są napisane przez osoby, które były przeciwnikami wdrożenia wątków w systemie Linux. Pokazano tam pewne meandry wykonywania wątków oraz częste błędy w postrzeganiu aplikacji przez osoby rozpoczynające swoje doświadczenia z programowaniem systemów współbieżnych. Pozycjami wartymi polecenia do nauki programowania zgodnego ze standardem POSIX są [BFN1996], [But1997]. Podręcznikiem akademickim, który szeroko i systematycznie omawia liczne metody programowania w standardzie POSIX jest [Uła2007]. Pomocnymi pracami w szeroko pojętych zagadnieniach programowania w Linux są: [Fus2007], [KLR2012], [Mat2000]. Pełną dokumentację systemu Linux oraz wiele materiałów szkoleniowych udostępniono na stronie [Ker2024].

Ćwiczenia

1. Jakie są ograniczenia nakładane na definicję funkcji, która ma być główną funkcją wątku?
2. Jeśli aplikacja składa się z kilku wątków potomnych, a każdy z nich wykonuje tę samą funkcję wątku, to czy wszystkie wątki muszą wykonywać dokładnie taki sam kod?
3. Na czym polega technika budowania aplikacji współbieżnych „farmer – worker”?
4. Zaproponuj kod aplikacji wielowątkowej, w której główny wątek tworzy wątek potomny, z kolei wątek potomny tworzy kolejny wątek potomny.
5. Zaproponuj kod aplikacji wielowątkowej, w której automatycznie tworzy się kilkanaście wątków.

Receptura 2. – cykliczna aplikacja wielowątkowa

Często spotykaną strukturą aplikacji czasu rzeczywistego jest zbiór cyklicznie wykonywanych wątków. Każdy z wątków w pojedynczym cyklu pobiera dane z otoczenia, przetwarza je, a następnie zwraca informację na zewnątrz. W podstawowej konfiguracji, która teraz będzie rozważana, wątki są niezależne. Uwaga zostanie skupiona na przygotowaniu zbioru wątków, które cyklicznie, co określony przedział czasu będą wznawiały swoje obliczenia, a potem zawieszały się do chwili ponownej aktywacji. Poprawne zaprogramowanie takiego schematu wykonywania się programu z punktu widzenia inżynierii oprogramowania czasu rzeczywistego wymaga zastosowania interfejsu sygnałów oraz budzików współpracujących z precyzyjnymi zegarami systemowymi.

Interfejs sygnałów

Sygnały w systemach operacyjnych to mechanizmy komunikacji międzyprocesowej. Są to krótkie komunikaty wysyłane do procesu przez system operacyjny lub inny proces w celu powiadomienia o jakimś zdarzeniu. Sygnały mogą być generowane przez różne zdarzenia, takie jak błędy, wyjątki, zakończenie procesu i inne.

Sygnały jako komunikaty nie niosą w sobie dodatkowych informacji poza swoim unikatowym numerem. Wysłanie sygnału można utożsamić z przekazaniem do danego procesu zdarzenia o określonym numerze. W systemie Linux jest zdefiniowana lista sygnałów powiązanych z danymi zdarzeniami, co pozwala programiście identyfikować przyczyny nadesłanych do jego aplikacji sygnałów. Na potrzeby tworzenia systemów czasu rzeczywistego jest natomiast wydzielona osobna lista sygnałów, które można powiązać z arbitralnie ustalonymi przez programistę zdarzeniami.

Dysponując identyfikatorem procesu w systemie (ang. Process IDetifier – PID), można wysłać do niego sygnał z zastosowaniem polecenia systemowego lub funkcji wywołanej w programie użytkownika o nazwie `kill`. Parametrami wywołania polecenia/funkcji są odpowiednio PID procesu, do którego ma zostać wysłany sygnał oraz numer sygnału.

Należy pamiętać, że domyślnym zachowaniem procesu, do którego wysłano jakikolwiek sygnał, jest jego natychmiastowe zakończenie. Posługiwanie się tym interfejsem wymaga więc „przygotowania” programu na „nadchodzące” do niego sygnały.

Wielowątkowy program obsługujący sygnały

Na listingu 2. pokazano przykładowy program reagujący na dwa wybrane sygnały. Zarówno wątek główny (funkcja `main`), jak i wątek potomny są „przygotowane" na nadejście sygnałów systemowych `SIGINT` i `SIGTERM`. Takie przygotowanie w praktyce oznacza włączenie do programu struktury typu `sigset_t`, która jest w stanie przechowywać zbiór sygnałów, na których nadejście jest gotowa dana aplikacja. Rejestracja sygnałów odbywa się przez opróżnienie zbioru (wywołanie funkcji `sigemptyset`), przypisanie do zbioru ustalonych sygnałów (wywołanie funkcji `sigaddset`), a następnie wskazanie bieżącemu wątkowi informacji, że ma być „przygotowany" na dodane do zbioru sygnały (wywołanie funkcji `pthread_sigmask`). W przykładowym programie czynność ta jest wykonywana zarówno w wątku głównym, jak i potomnym.

Listing 2. Aplikacja wielowątkowa przygotowana na odebranie sygnałów SIGINT i SIGTERM

```
#include <stdio.h>
include <stdlib.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>

void *child_thread(void *arg) {
    sigset_t set;
    int sig;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    if (pthread_sigmask(SIG_BLOCK, &set, NULL) != 0) {
        perror("pthread_sigmask");
        return NULL;
    }

    if (sigwait(&set, &sig) != 0) {
        perror("sigwait");
        return NULL;
    }

    printf("Otrzymano sygnal: %d\n", sig);

    pthread_exit(NULL);
}

int main() {
```

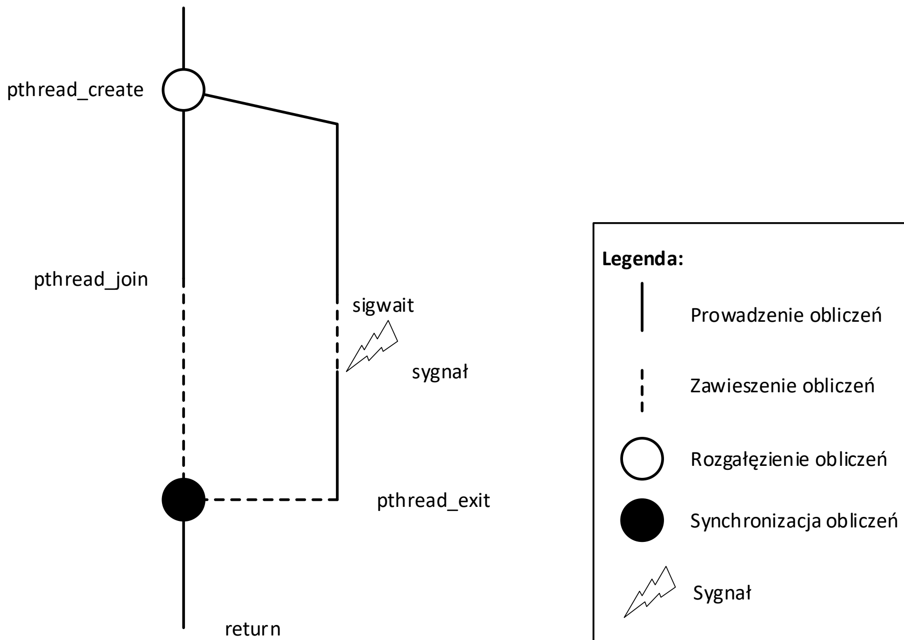
```
    pthread_t thread;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);

    if (pthread_sigmask(SIG_BLOCK, &set, NULL) != 0) {
        perror("pthread_sigmask");
        return EXIT_FAILURE;
    }

    if (pthread_create(&thread, NULL, child_thread, NULL)
           != 0) {
        perror("pthread_create");
        return EXIT_FAILURE;
    }

    if (pthread_join(thread, NULL) != 0) {
        perror("pthread_join");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Wątek potomny, oprócz wbudowanej „odporności" na wskazane nadchodzące sygnały, jest także wyposażony w mechanizm oczekiwania na nadejście sygnału. Wywołana w nim funkcja `sigwait` zawiesza jego wykonywanie do momentu nadejścia sygnału. W przykładowym programie pojawienie się sygnału, na który wątek jest przygotowany, powoduje odblokowanie się wykonywania wątku oraz wypisanie numeru sygnału, który został do niego przekazany. Warto zaznaczyć, że jeśli do wykonywanego programu zostanie wysłany sygnał inny, niż założono, to program zostanie natychmiast zakończony i nie zwróci żadnej informacji.

Jeśli program zostanie uruchomiony w terminalu systemu Linux, to łatwo można do niego wysłać sygnał SIGINT (np. naciskając kombinację klawiszy Ctrl+C w konsoli, z której uruchomiono program). Program powinien wtedy zwrócić tekst „Otrzymano sygnal: 2" i zakończyć działanie.

Scenariusz działania aplikacji

Na rysunku 4. pokazano przykładowy scenariusz działania omawianej aplikacji.



Rysunek 4. Przykładowy scenariusz działania aplikacji oczekującej na sygnał

Wątek potomny po wywołaniu funkcji `sigwait` zawiesza swoje wykonywanie i oczekuje na pojawienie się sygnału. Sygnał skierowany do aplikacji odblokowuje wykonywanie wątku i pozwala na jego zakończenie, synchronizację z wątkiem głównym i zakończenie całej aplikacji.

Zegary i budziki czasu rzeczywistego

Aby zapewnić terminowe wykonywanie obliczeń, systemy operacyjne czasu rzeczywistego powinny umożliwiać precyzyjny odczyt bieżącego czasu oraz udostępnić precyzyjne budziki do wznawiania obliczeń przez zadania cykliczne. Standard POSIX narzuca, że w systemie z nim zgodnym musi istnieć przynajmniej jeden zegar czasu rzeczywistego (`CLOCK_REALTIME`). Powinien on precyzyjnie odmierzać czas systemu komputerowego z częstotliwością co najmniej 50 Hz (generując impulsy zegarowe przynajmniej co 20 ms). Najważniejszą cechą tego zegara jest to, że niezależnie od obciążenia systemu, czas od momentu zgłoszenia zdarzenia przez ten zegar do momentu uruchomienia odpowiedniego programu odpowiadającego na to zdarzenie jest przewidywalny i z góry ustalony. Innymi słowy, funkcja odczytująca bieżący czas lub reagująca na sygnał zawsze zostanie uruchomiona nie później niż w ustalony przedział czasowy.

Pewnym nieporozumieniem jest fakt, że praktycznie wszystkie systemy operacyjne rodziny UNIX, aby pozostały zgodne ze standardem POSIX, oferują zegary klasy `CLOCK_REALTIME`. Nie należy ich jednak uznawać za systemy operacyjne czasu rzeczywistego. W praktyce należy się posługiwać tylko tymi systemami operacyjnymi, które mają być przeznaczone do osadzania na nich aplikacji czasu rzeczywistego. Zdaniem autora zawsze trzeba wykazać, że dana para: system mikroprocesorowy i system operacyjny czasu rzeczywistego gwarantują ograniczony i precyzyjnie określony czas uruchamiania procedur obsługi zdarzeń czasowych. Jak już wspomniano, najprościej jest nabyć system mikroprocesorowy wraz z zainstalowanym systemem operacyjnym, polecany przez producenta systemu operacyjnego. Inną ścieżką otrzymania takiej platformy sprzętowo-programowej jest samodzielne zainstalowanie systemu i przeprowadzenie na nim odpowiednich testów. Tylko system mikroprocesorowy certyfikowany lub zweryfikowany do pracy z danym systemem operacyjnym czasu rzeczywistego gwarantuje spełnienie omówionych zależności czasowych pomiędzy układem elektronicznym obliczającym czas a oprogramowaniem.

Wielowątkowy program sterowany zdarzeniami czasowymi

Zakładając, że dysponuje się zegarem czasu rzeczywistego poprawnie współpracującym z systemem operacyjnym czasu rzeczywistego, można skonstruować aplikację, która cyklicznie wysyła do samej siebie sygnały. Te z kolei pozwalają na wznawianie kolejnych cykli obliczeniowych aplikacji. Generatorami sygnałów mogą być budziki czasu rzeczywistego (ang. Real-Time Timers). Na listingu 3. zamieszczono przykładową aplikację spełniającą takie założenia.

Listing 3. Przykładowa aplikacja z dwoma wątkami sterowanymi zdarzeniami czasowymi

```
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)

void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1;
    struct itimerspec its1;
```

```
    struct sigevent sev1;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                                 // 2-sekundowe opóźninie
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 1; // 1-sekundowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("Watek 1: obliczenia wznowione\n");
    }
    pthread_exit(NULL);
}

void *thread2(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
    timer_create(CLOCK_REALTIME, &sev2, &timer2);

    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                                  // 2-sekundowe opóźninie
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;
```

```
    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("Wątek 2: obliczenia wznowione\n");
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);
    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

W aplikacji zastosowano mechanizm obsługi sygnałów omówiony we wcześniejszych rozdziałach. Zasada „przygotowania" aplikacji do obsługi sygnałów jest taka sama jak w przykładzie z listingu 2., inne są natomiast obsługiwane sygnały. Na potrzeby aplikacji czasu rzeczywistego w systemach operacyjnych zgodnych ze standardem POSIX powinny być zdefiniowane wydzielone sygnały w zakresie od `SIGRTMIN` do `SIGRTMAX`. W przykładowej aplikacji są generowane sygnały o numerach `SIGRTMIN` i `SIGRTMIN+1`. Główny wątek programu (funkcja main) jest „przygotowany" na pojawienie się obu sygnałów. Wątek pierwszy jest gotowy na przyjmowanie sygnału `SIGRTMIN`, drugi natomiast – na przyjmowanie sygnału `SIGRTMIN+1`.

W programie zastosowano dwa typy struktur do przechowywania czasu: `timespec` oraz `itimerspec`. Pierwsza służy w programie do przechowania odczytanego czasu systemowego, druga zaś zawiera dwie wartości czasowe: jednorazowe opóźnienie, z jakim budzik ma rozpocząć generowanie sygnałów oraz czas cyklu budzika. Struktura danych typu `sigevent` zawiera z kolei informację, jakiego rodzaju sygnał ma być wytwarzany podczas każdego przebudzenia budzika. Struktura danych typu `timer_t` służy do przechowywania identyfikatora budzika.

Główna funkcja programu powołuje do życia dwa wątki potomne. Funkcje wątków potomnych w omawianej aplikacji kolejno:

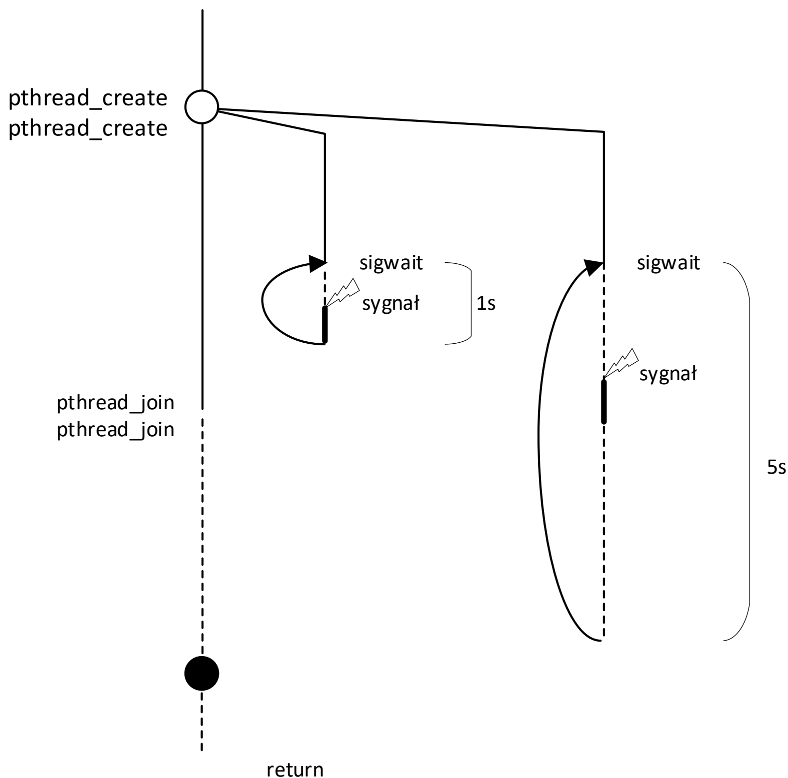
- rejestrują sygnał, na który będą reagować (wywołanie funkcji `sigemptyset` i `sigaddset`);
- tworzą budzik czasu rzeczywistego (wywołanie funkcji `timer_create`);
- pobierają bieżący czas systemowy (wywołanie funkcji `clock_gettime`);
- wypełniają strukturę danych opisujących działanie budzika w taki sposób, że budzik ma rozpocząć generowanie sygnałów po upływie 2 s od bieżącej chwili, a interwały czasowe pomiędzy kolejnymi przebudzeniami budzika ustalono odpowiednio na 1 s (dla wątku `thread1`) i 5 s (dla wątku `thread2`);
- przekazują do budzika ustalone parametry czasowe pracy (wywołanie funkcji `timer_settime`);
- uruchamiają nieskończone pętle obliczeniowe, które oczekują na sygnał (wywołanie funkcji `sigwait`), po jego odebraniu wykonują jeden cykl obliczeniowy wątku i ponownie zawieszają swoje działanie w oczekiwaniu na kolejny sygnał.

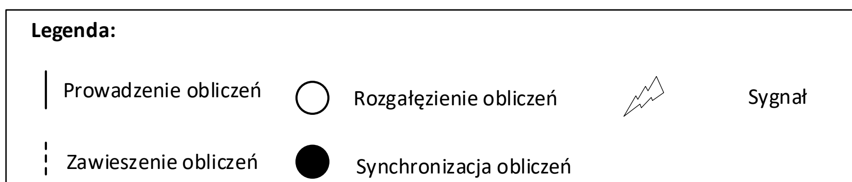
Scenariusz działania aplikacji

Na rysunku 5. przedstawiono schemat działania aplikacji z listingu 3. W programie utworzone zostają dwa wątki potomne. Każdy z nich po uruchomieniu „własnego” budzika zawiesza swoje działanie w oczekiwaniu na sygnał. Po otrzymaniu sygnału wątek się odblokowuje, wykonuje swoje obliczenia i ponownie zawiesza się do otrzymania kolejnego sygnału. Praca wątków potomnych nigdy się nie kończy i nigdy nie następuje synchronizacja z głównym wątkiem aplikacji.

Literatura uzupełniająca

Zasady posługiwania się sygnałami i budzikami czasu rzeczywistego do tworzenia cyklicznie wznawianych wątków omówiono w pracach: [BW2009], [Gal1995] i [Uła2007]. Testy i ocenę jakości systemu operacyjnego zainstalowanego na danej platformie sprzętowej zaproponowano w dokumentach [DSE2001a] i [DSE2001b]. Cenne informacje o zasadach programowania cyklicznych wątków zgodnych ze standardem POSIX można również znaleźć w bogatej dokumentacji komercyjnych systemów operacyjnych czasu rzeczywistego, takich jak VxWorks i QNX. Na uwagę zasługują dokument [VxW2011] i podręcznik akademicki [Uła2007]. Tworzenie cyklicznych wątków na podstawie rozszerzonej listy funkcji z rodziny pthreads można przestudiować w pozycji [LRO2003].





Rysunek 5. Schemat wykonywania dwóch cyklicznych wątków sterowanych czasowo

Ćwiczenia

1. Jakie jest domyślne zachowanie procesu, jeśli zostanie do niego wysłany sygnał?
2. Funkcja `sigwait` nie jest jedyną metodą programowania reakcji programu na sygnał. Znajdź w dostępnej literaturze inną metodę przechwytywania i reakcji na sygnał.
3. Jak wyobrażasz sobie zastosowanie funkcji `clock_gettime` do obliczania czasu wykonywania fragmentu programu.

4. Znajdź w zasobach Internetu gotowe systemy uruchomieniowe składające się z systemu operacyjnego czasu rzeczywistego i certyfikowanego dla niego sprzętu komputerowego.
5. Uzupełnij wiedzę na temat standardowych sygnałów zdefiniowanych w systemach Linux.

Receptura 3. – szeregowanie cyklicznych wątków

Program z listingu 3. wydaje się spełniać podstawowe wymagania stawiane aplikacjom czasu rzeczywistego. Z punktu widzenia inżynierii oprogramowania czasu rzeczywistego wymaga jednak pewnego uzupełnienia. Na przykład nie wiadomo, czy zawsze spełni wymagania czasowe. Aby to rozstrzygnąć, należy wprowadzić zagadnienia szeregowania zbioru zadań czasu rzeczywistego wraz z odpowiednim zbiorem funkcji pozwalających na zarządzanie zadaniami w systemie operacyjnym.

Problem szeregowania w systemach czasu rzeczywistego

Jednym z podstawowych zadań projektanta współbieżnego systemu czasu rzeczywistego jest wykazanie, że wszystkie cykliczne wątki zawsze obliczą swoje wyniki w zadanych ograniczeniach czasowych. W tej części skryptu problem szeregowania wątków zostanie rozstrzygnięty w takim zakresie, jak pozwalają na to standard POSIX i techniki zarządzania wątkami dostępne w systemie Linux.

Postawiony problem wiąże się ściśle z procesem harmonogramowania obliczeń w systemie operacyjnym. Wątki obliczeniowe uruchamiane pod nadzorem systemu operacyjnego są przydzielane do wykonywania do rdzeni procesora. Określa się dla nich priorytety, czyli swego rodzaju stopień ważności dla systemu operacyjnego. Wątek o najwyższym priorytecie otrzymuje czas procesora. Kiedy wątek zakończy swoje obliczenia, lub kiedy w systemie pojawi się inny wątek o wyższym priorytecie system operacyjny podejmuje decyzję o przełączeniu kontekstu, czyli wstrzymaniu wykonywania obliczeń przez jeden wątek na rzecz innego. W systemach operacyjnych czasu rzeczywistego programista, wywołując odpowiednie usługi systemowe, może w szerszy sposób niż dopuszczają standardowe systemy operacyjne ingerować w algorytm harmonogramowania oraz przydzielania priorytetów. Może się również posłużyć precyzyjnymi budzikami systemowymi, które mogą w określonych momentach czasowych ponownie wprowadzać wątki do programu harmonogramującego.

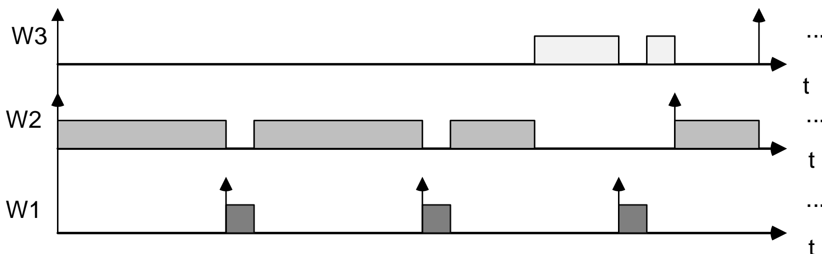
Dla rozwiązania problemu projektowego przyjęte zostaną następujące założenia:

- dany podsystem czasu rzeczywistego będzie się składał ze zbioru wątków na stałe przypisanych do wskazanego pojedynczego rdzenia systemu mikroprocesorowego;

- pokazana zostanie strategia przełączania kontekstu oraz nadawania priorytetu wątkom, gwarantująca z punktu widzenia matematycznego spełnienie ograniczeń czasowych dla wszystkich wątków;
- system będzie się składał ze zbioru niezależnych (niekomunikujących się ze sobą) cyklicznych wątków czasu rzeczywistego;
- system operacyjny zezwala na wywłaszczanie wątków, co oznacza, że dopuszcza się przerwanie obecnie wykonywanego wątku i uruchomienie na jego miejsce innego wątku w dowolnej chwili.

Przy takich założeniach schemat wykonywania obliczeń przez poszczególne wątki na jednym rdzeniu procesora można pokazać na wykresie czasowym (rys. 6.).

Poszczególne prostokąty na rys. 6. oznaczają wykonywanie porcji obliczeń przez wątki. Strzałki wskazują chwile w czasie, kiedy dany wątek otrzymuje sygnał do ponownego wznowienia obliczeń. Wątek W1, najczęściej wznawiany do wykonywania, kilkukrotnie wywłaszcza wątki W2 i W3. Obliczenia wątku W3 nie rozpoczynają się od razu po otrzymaniu sygnału, ale są odroczone, ponieważ inne wątki prowadzą obliczenia. Mimo to oczekuje się, że wątek ten zakończy swoje obliczenia przed sygnałem oznaczającym jego ponowne uruchomienie.



Rysunek 6. Przykładowy przebieg wykonywania trzech wątków na pojedynczym rdzeniu procesora

Wykres pokazuje tylko fragment przebiegu obliczeń. Zadaniem projektanta systemu jest wykazanie, że do końca działania systemu (system może działać potencjalnie nieskończenie długo) wszystkie wątki zawsze ukończą swoje obliczenia przed uzyskaniem nowego sygnału wznawiającego ich pracę.

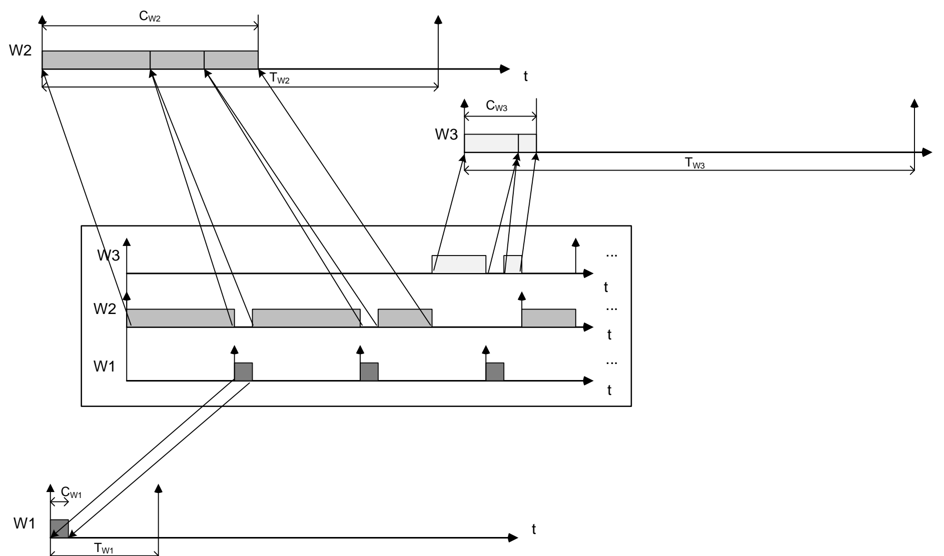
Współczynnik wykorzystania procesora

Jednym z matematycznych narzędzi, które może posłużyć do stwierdzenia, czy dany zbiór wątków zawsze wykona swoje obliczenia przed terminem ponownego ich wznowienia, jest obliczenie współczynnika wykorzystania procesora

(ang. Processor Utilization Factor). W przypadku pojedynczego rdzenia procesora wartość współczynnika wykorzystania procesora oblicza się ze wzoru:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

Dla każdego wątku wyznacza się maksymalny czas obliczeń w pojedynczym cyklu (C_i) i dzieli się go przez czas pojedynczego cyklu (T_i). Następnie sumuje się tak wyliczone ułamki dla wszystkich n wątków. Rysunek 7. pokazuje, jak należy rozumieć maksymalny czas wykonywania obliczeń oraz czas pojedynczego cyklu w omawianym wzorze, wychodząc ze scenariusza szeregowania pokazanego na rys. 6.



$$U = \frac{C_{W1}}{T_{W1}} + \frac{C_{W2}}{T_{W2}} + \frac{C_{W3}}{T_{W3}}$$

Rysunek 7. Interpretacja maksymalnego czasu obliczeń, długości cyklu dla wątku cyklicznego i współczynnika wykorzystania procesora

Wartości długości cyklu dla poszczególnych wątków wynikają z wymagań postawionych aplikacji. Ustala się na przykład, że dany algorytm sterujący musi wyliczać kolejne wartości sterowania co 50 ms. Trudniejszym zadaniem jest

wskazanie maksymalnego czasu obliczeń w pojedynczym cyklu. Można tę wartość eksperymentalnie oszacować, uruchamiając na danym rdzeniu tylko jeden wątek (w celu wyeliminowania przełączania kontekstu), a następnie rejestrując czas rozpoczęcia obliczeń i czas zakończenia obliczeń. Systemy operacyjne czasu rzeczywistego pozwalają precyzyjnie odczytywać czas w ustalonych chwilach (por. funkcję `clock_gettime` wspomnianą we wcześniejszym rozdziale skryptu). Pozostaje wówczas wyliczenie różnicy czasu. Niektóre systemy operacyjne dysponują dodatkowymi narzędziami, które monitorują i archiwizują przebieg pracy aplikacji czasu rzeczywistego. Wtedy oszacowanie maksymalnego czasu wykonywania obliczeń w jednym cyklu można uzyskać na podstawie otrzymanych danych historycznych.

Algorytm planowania ze stałymi priorytetami

Jeśli dla danego zestawu wątków przypisanych do pojedynczego rdzenia mikroprocesora można obliczyć wartość współczynnika wykorzystania procesora, to przy założeniu, że wątki mogą być wywłaszczane w dowolnej chwili oraz mają przypisane priorytety (na stałe związane z danymi wątkami), można ustalić regułę przypisywania priorytetów do wątków gwarantującą zawsze ich szeregowalność. Wspomniana reguła jest nazywana algorytmem planowania ze stałymi priorytetami (ang. rate monotonic) i sprowadza się do następującej zasady: **im dany wątek jest częściej wznawiany (ma krótszy cykl wznawiania obliczeń), tym jego priorytet ma być wyższy**.

Analizując scenariusz wykonywania wątków pokazany na rys. 6., można zauważyć, że były one wykonywane właśnie z zastosowaniem podanej reguły szeregowania. Wątek W1, jako najczęściej wznawiany, był wykonywany z najwyższym priorytetem. Zawsze gdy system otrzymywał informację, że jest on gotowy do rozpoczęcia obliczeń w nowym cyklu, pozostałe inne wątki były zawieszane i czas procesora był mu przekazywany. Wątkiem o średnim priorytecie był W2. Pozwalał on na przerwanie swoich obliczeń (wywłaszczanie) przez wątek W1, równocześnie powodował odroczenie rozpoczęcia wykonywania wątku W3.

Warunkiem szeregowalności zbioru wątków czasu rzeczywistego uruchomionych na pojedynczym rdzeniu obliczeniowym jest spełnienie następującej zależności dla współczynnika wykorzystania procesora:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right).$$

Okazuje się więc, że dysponując jedynie liczbą *n* wątków, które będą wykonywane na danym rdzeniu procesora, oraz współczynnikiem wykorzystania procesora, można wykazać, czy wszystkie wątki zawsze będą spełniać swoje wymagania czasowe.

Jeśli w danym podsystemie czasu rzeczywistego planuje się uruchomienie dużej liczby wątków, to można się posłużyć graniczną wartością współczynnika wykorzystania procesora (U_{gr}):

$$U_{gr} = \ln 2 \ \approx 0{,}69.$$

Wielowątkowy program z algorytmem planowania ze stałymi priorytetami

Biorąc pod uwagę omówione zagadnienia związane z szeregowaniem zadań czasu rzeczywistego, na listingu 4. pokazano wersję aplikacji z listingu 3. uzupełnioną:

- o priorytety przypisane wątkom i ustalone zgodnie z regułą „rate montonic";
- o ustalenie zasady szeregowania wątków.

Listing 4. Dwa wątki sterowane czasowo z ustalonym algorytmem szeregowania i priorytetami

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)

void thread_attributes_test(void)
{   pthread_attr_t my_attrs;
    int scope;
    int inheritsched;
    int policy;
    struct sched_param param;

    pthread_getattr_np(pthread_self(),&my_attrs);

    pthread_attr_getscope(&my_attrs,&scope);
    printf("Zakres szeregowania watkow");
    switch (scope)
```

```
    {   case PTHREAD_SCOPE_SYSTEM:
         printf("Wszystkie watki i procesy konkuruja");
         printf("w jednej puli priorytetow. \n");
        break;
        case PTHREAD_SCOPE_PROCESS:
         printf("Priorytety watkow sa definiowane");
         printf("na poziomie procesow.");
         printf("Procesy konkuruja na poziomie systemu\n");
        break;
        default:
         printf("Nieznany uklad przydzialu priorytetow");
         printf("dla watkow.\n");
        break;
    }
    printf("Dziedziczenie polityki szeregowania:  ");
    pthread_attr_getinheritsched(&my_attrs,&inheritsched);
    switch (inheritsched)
    {   case PTHREAD_INHERIT_SCHED:
         printf("Wlaczone jest dziedziczenie polityki");
         printf("szeregowania po rodzicu\n");
        break;
        case PTHREAD_EXPLICIT_SCHED:
         printf("Watki nie dziedzicza polityki");
         printf("szeregowa nia.");
         printf("Same ustalaja polityke szeregowania.\n");
        break;
        default:
          printf("Nierozpoznana polityka szeregowania\n");
                                    break;
    }
    printf("Wybrana polityka szeregowania:  ");
    pthread_attr_getschedpolicy(&my_attrs, &policy);
    switch (policy)
    {   case SCHED_FIFO:    printf("SCHED_FIFO.\n");
                            break;
        case SCHED_RR:      printf("SCHED_RR.\n");
                            break;
        case SCHED_OTHER:   printf("SCHED_OTHER.\n");
                            break;
        default:            printf("Nieznana polityka \n");
                            break;
    }
    pthread_getschedparam(pthread_self(),&policy, &param);
    printf("Priorytet biezacego wotku: ");
    printf("%d\n",param.sched_priority);
    pthread_attr_destroy(&my_attrs);
}
```

```
void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1;
    struct itimerspec its1;
    struct sigevent sev1;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                                 // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 1; // 1-sekundowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    printf("Watek 1 polityka szeregowania:\n");
    thread_attributes_test();

    while (1) {
        sigwait(&set, &sig);
        //printf("Watek 1: obliczenia wznowione\n");
    }
    pthread_exit(NULL);
}

void *thread2(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
    timer_create(CLOCK_REALTIME, &sev2, &timer2);
```

```
    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                              // 2 sekundy opźnienia
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;

    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);

    printf("Watek 2 polityka szeregowania:\n");
    thread_attributes_test();

    while (1) {
        sigwait(&set, &sig);
        //printf("Watek 2: obliczenia wznowione\n");
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_attr_t attr1, attr2;
    struct sched_param param1, param2, param_main;
    cpu_set_t cpuset;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);
    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset); // Wybierz procesor nr 2
                            w systemie
    printf("Liczba dostepnych CPU: ");
    printf("%d\n", (int) sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t), &cpuset);

    printf("Glowny watek, polityka szeregowania:\n");
    thread_attributes_test();
```

```
    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
                                 PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                                sizeof(cpu_set_t),
                                cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_attr_init(&attr2);
    pthread_attr_setscope(&attr2, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr2,
                                 PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
    param2.sched_priority =
                  sched_get_priority_max(SCHED_FIFO) - 2;
    pthread_attr_setschedparam(&attr2, &param2);
    pthread_create(&t2, &attr2, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);

    return 0;
}
```

Porównując funkcje wątków z listingu 3. i 4., można zauważyć jedną modyfikację. Polega na dodaniu do kodu każdej funkcji wątku uruchomienia funkcji thread_attributes_test. Jest to jawnie napisana funkcja, która wyświetla w terminalu aplikacji informacje dotyczące priorytetu i algorytmu szeregowania zastosowanego w danym wątku programu. Pozwala ona po uruchomieniu systemu sprawdzić, czy założone parametry zostały włączone.

Kluczowych modyfikacji dokonano w głównej funkcji programu (main). Powiększony został zestaw pomocniczych zmiennych. Zmienne typu pthread_attr_t będą zawierać tak zwane atrybuty wątków. Atrybuty te są przekazywane jako argument wywołania funkcji pthread_create i były dotąd pomijane. W obecnym przykładzie mają decydujące znaczenie, ponieważ pozwalają na zdefiniowanie: algorytmu szeregowania, priorytetu oraz numeru procesora, do którego zostanie przypisany dany wątek. W ustalaniu parametrów pracy wątków pomocnicze znaczenie odgrywają również zmienne typu struct

sched_param oraz cpu_set_t. Pierwsze z nich będą zawierać informacje pozwalające wprowadzić do atrybutów priorytet, a drugie – numer rdzenia procesora dla danego wątku.

Przyłączanie wątków do rdzenia mikroprocesora

Wprowadzone wcześniej założenia dotyczące szeregowania dotyczyły aplikacji, dla których wszystkie wątki są uruchomione na tym samym rdzeniu procesora. Wywołanie funkcji CPU_ZERO przygotowuje zmienną cpu_set_t do przyjęcia informacji o liczbie rdzeni procesora. Z kolei funkcja CPU_SET przyłącza do wspomnianej struktury wybrany numer rdzenia. Sposób wywołania tych dwóch funkcji w przykładowej aplikacji powoduje dołączenie do zbioru rdzeni tylko tego o numerze 1 (czyli drugiego z ewentualnie dostępnych). W dalszej części programu odpowiednia funkcja będzie mogła wymusić uruchomienie danego wątku właśnie na wskazanym rdzeniu. Tym sposobem będzie można przygotować aplikację czasu rzeczywistego i wykazać dla niej szeregowalność, ponieważ wszystkie jej wątki będą na stałe powiązane ze wskazanym rdzeniem. W tym miejscu programu następuje również wypisanie na konsoli informacji, ile rdzeni jest dostępnych w systemie mikroprocesorowym, na którym jest uruchomiona aplikacja.

Ustalanie dla wątków parametrów szeregowania

Z punktu widzenia standardu POSIX można wymusić algorytm szeregowania, a także przypisanie określonych priorytetów dla wątków. Podstawowym algorytmem czasu rzeczywistego, jaki można ustalić w systemach zgodnych z POSIX, jest SCHED_FIFO. Wątki są wtedy szeregowane zgodnie z przypisanymi priorytetami. Jeśli do puli wykonywanych wątków wejdzie wątek o wyższym priorytecie, następuje natychmiastowe zawieszenie wykonywanego wątku i uruchomienie tego nowego. Aby zapewnić w takim systemie poprawne szeregowanie zgodnie z algorytmem planowania ze stałymi priorytetami, należy równocześnie z uruchomieniem algorytmu szeregującego powiązać z wątkami odpowiednie priorytety.

Główna funkcja programu (main) sama jest jego pierwszym wątkiem. W omawianej aplikacji pełni ona rolę farmera, który uruchamia pozostałe wątki (workery). Przyjęto, że będzie szeregowana jak inne wątki, zostanie włączona do aplikacji czasu rzeczywistego z najniższym priorytetem i zostanie przywiązana do drugiego rdzenia systemu mikroprocesorowego. Dokonano tego, uruchamiając następujące instrukcje:

```
param_main.sched_priority =
                        sched_get_priority_max(SCHED_FIFO) - 3;
```

```
pthread_setschedparam(pthread_self(), SCHED_FIFO,
                      &param_main);
pthread_setaffinity_np(pthread_self(),
                       sizeof(cpu_set_t), &cpuset);
```

Algorytm szeregowania `SCHED_FIFO` w systemie operacyjnym ma zarejestrowaną osobną pulę priorytetów dla wątków mieszczących się w zakresie od `sched_get_priority_max(SCHED_FIFO)` i `sched_get_priority_min(SCHED_FIFO)`. Przyjęto, że główny wątek programu będzie miał priorytet `sched_get_priority_max(SCHED_FIFO) - 3`. Samo wymuszenie dla funkcji `main` ustalonego algorytmu szeregowania i priorytetu odbywa się przez wywołanie funkcji `pthread_setschedparam`. Podanie jako pierwszego parametru jej wywołania wartości `pthread_self()` oznacza, że zmiany dokonujemy dla siebie samego, czyli głównego wątku programu. Z kolei wywołanie funkcji `pthread_setaffinity_np` przywiązuje główny wątek aplikacji do procesora o numerze 1.

Dwie kolejne sekcje programu wewnątrz funkcji `main` dotyczą ustalenia parametrów wątków potomnych i obejmują wywołanie identycznych sekcji instrukcji przypisanych odpowiednio atrybutom wątku `pthread1` (`attr1`) i wątku `pthread2` (`attr2`). Jedna z tych sekcji wygląda następująco:

```
pthread_attr_init(&attr1);
pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
pthread_attr_setinheritsched(&attr1,
                             PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
param1.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 1;
pthread_attr_setschedparam(&attr1, &param1);
pthread_attr_setaffinity_np(&attr1,
                            sizeof(cpu_set_t), &cpuset);
pthread_create(&t1, &attr1, thread1, NULL);
```

Struktury danych zawierające atrybuty wątków muszą przed modyfikacjami zostać zainicjalizowane (por. wywołanie funkcji `pthread_attr_init`).

Dla systemów zgodnych z POSIX przewidziano dwie możliwości zarządzania priorytetami wątków. Pierwsza, określona identyfikatorem `PTHREAD_SCOPE_SYSTEM` zakłada, że wszystkie wątki w systemie tworzą „płaską” strukturę i na równi współzawodniczą o zasoby. Druga, określona identyfikatorem `PTHREAD_SCOPE_PROCESS` zakłada, że na podstawowym poziomie o zasoby współzawodniczą procesy, a wątki ustalają swoje priorytety osobno w ramach każdego procesu. W omawianej aplikacji za pomocą wywołania funkcji

pthread_attr_setscope „zasugerowano” systemowi zarządzanie priorytetami wątków zgodnie ze strategią PTHREAD_SCOPE_SYSTEM (jest to zresztą jedyna strategia dostępna w systemach Linux).

W standardzie POSIX zaproponowano dwie strategie otrzymywania parametrów szeregowania. Pierwsza, identyfikowana przez stałą PTHREAD_INHERIT_SCHED ustala, że właściwości szeregowania wątków potomnych są dziedziczone po rodzicu. Druga, wybrana w przykładowej aplikacji, identyfikowana przez stałą PTHREAD_EXPLICIT_SCHED ustala, że parametry szeregowania nie są dziedziczone i powinny być wprost zdefiniowane przez programistę. Wybranie strategii otrzymywania parametrów szeregowania polega na uruchomieniu funkcji pthread_attr_setinheritsched z parametrami wskazującymi atrybuty wątku i identyfikator strategii.

Uruchomienie funkcji pthread_attr_setschedpolicy w przykładowym programie proponuje aplikacji szeregowanie wątków zgodnie z omówioną wcześniej metodą SCHED_FIFO. Z kolei uruchomienie funkcji pthread_attr_setschedparam pozwala na przydzielenie przyszłemu wątkowi wybranego priorytetu. Ostatnią fazą przygotowania atrybutów wątku jest wskazanie, na którym rdzeniu ma być wykonywany (por. wywołanie funkcji pthread_attr_setaffinity_np).

Ostatecznie, dla obu wątków potomnych omawianej aplikacji zostają odpowiednio zmodyfikowane atrybuty, tak żeby każdy z nich podlegał regule SCHED_FIFO, miał ustalony własny, statyczny priorytet i był uruchomiony na wskazanym rdzeniu mikroprocesora. Funkcja pthread_create „przejmuje” atrybuty wątków i uruchamia je z ustalonymi właściwościami.

Technika uruchamiania aplikacji

W odróżnieniu od wcześniej pokazanych przykładowych aplikacji, które można było uruchomić z uprawnieniami zwyczajnego użytkownika, do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Bez tych uprawnień nie będzie można uzyskać dostępu do omawianych strategii szeregowania, priorytetów i metod ustalania właściwości atrybutów wątków. Przykładowe polecenie uruchamiające program może wyglądać następująco: *„sudo ./program”*. System operacyjny może wymusić wtedy podanie hasła administratora.

Funkcje spoza standardu POSIX zastosowane w aplikacji

W przykładowym programie zastosowano dwie funkcje z biblioteki pthreads, które nie należą do standardu POSIX: pthread_getattr_np oraz

`pthread_setaffinity_np`. Przyrostek „np” oznacza w nich z języka angielskiego „**n**on **p**ortable”, czyli „nieprzenaszalne”. Pierwszą z nich posłużono się w pomocniczej funkcji `thread_attributes_test`, a drugą – do wskazania atrybutom wątku numeru procesora, gdzie ma być dany wątek uruchomiony. Aplikacja napisana w taki sposób nadaje się tylko do uruchomienia na rodzinie systemów Linux. Dla zachowania przenośności aplikacji można zrezygnować z funkcji pomocniczej, a wskazanie rdzenia, na którym ma być uruchomiona aplikacja, można zdefiniować w skrypcie powłoki systemu operacyjnego. System operacyjny Linux można tak skonfigurować, aby na wskazanym rdzeniu nie wykonywały się żadne inne aplikacje, oprócz tej wybranej przez użytkownika. Postępowanie takie zwykle obniża ogólną wydajność systemu, pozwala jednak na uruchomienie przewidywalnego pod względem czasowym podsystemu czasu rzeczywistego.

Literatura uzupełniająca

W rozdziale pokazano tylko niewielki wycinek bardzo obszernego zagadnienia dotyczącego szeregowania wątków we współbieżnych aplikacjach czasu rzeczywistego. Publikacją, która uszczegóławia technikę projektowania aplikacji pracujących zgodnie ze strategią Rate Monotonic jest [Kle1993]. Czytelnik zainteresowany poszerzeniem swojej wiedzy o inne strategie szeregowania wątków oraz przydziału zasobów dla pojedynczych rdzeni procesora powinien zainteresować się publikacjami: [But2011], [SSRB1998], [BW2009]. W wymienionych pracach warto zwrócić uwagę na algorytm szeregowania Earliest Deadline First (EDF), który jest wspierany między innymi w języku programowania systemów czasu rzeczywistego Ada. Zakłada on możliwość dynamicznego przydziału priorytetów dla wątków czasu rzeczywistego i umożliwia lepsze wykorzystanie procesora. Oprócz strategii statycznego przydziału zadań do pojedynczego rdzenia, którą zaproponowano w tym rozdziale, opracowano wiele innych. Czytelnicy chcący poszerzyć swoje wiadomości z zakresu szeregowania wątków czasu rzeczywistego w systemach wieloprocesorowych mogą przestudiować między innymi książki [CDKM2002] i [BBB2015].

Ćwiczenia

1. Dany jest zbiór 3 niezależnych zadań czasu rzeczywistego o następujących parametrach: {(T_1 = 100 µs, C_1 = 10 µs), (T_2 = 30 µs, C_2 = 10 µs), (T_3 = 25 µs, C_3 = 10 µs)}. Sprawdź, czy jest on szeregowalny, jeśli zastosuje się dla niego strategię szeregowania Rate Monotonic.
2. Jak działa reguła szeregowania `SCHED_FIFO`?

3. Podstawowe zasady wiązania priorytetów do wątków są opisane identyfikatorami `PTHREAD_SCOPE_SYSTEM` i `PTHREAD_SCOPE_PROCESS`. Jak należy rozumieć te zasady?
4. W jaki sposób można wymusić w programie przyłączenie wątków do wykonywania na konkretnym rdzeniu procesora?
5. Znajdź w Internecie informację, jak można w czasie uruchamiania systemu Linux zabronić mu uruchamiania jakichkolwiek procesów na wybranym rdzeniu procesora? Dowiedz się, czy można uruchomić wtedy własny program, który korzysta właśnie z tego wydzielonego rdzenia.

Receptura 4.
– techniki lokalnej komunikacji

Dotąd rozważania na temat konstruowania aplikacji czasu rzeczywistego zakładały, że wszystkie wątki są niezależne i nie wymieniają pomiędzy sobą informacji. Ta część skryptu jest poświęcona wprowadzaniu do systemu wybranych mechanizmów komunikacji. Samo wprowadzenie takich mechanizmów w systemach współbieżnych ma istotne konsekwencje (możliwość pojawienia się zjawisk zakleszczenia i głodzenia). W aplikacjach czasu rzeczywistego należy ponadto uwzględnić dodatkowe opóźnienia w czasie wykonywania wątków, powstające na skutek komunikacji.

Techniki komunikacji lokalnej stosowane w systemach czasu rzeczywistego

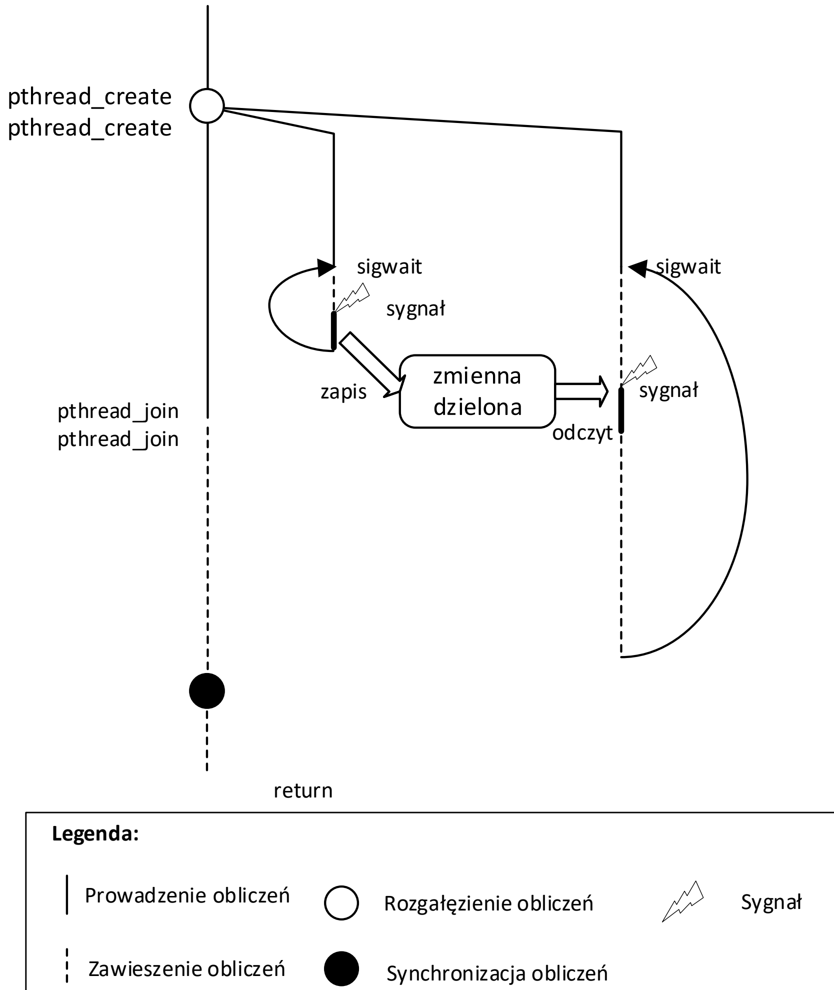
Przedmiotem tego skryptu są aplikacje czasu rzeczywistego składające się z cyklicznych, współbieżnie wykonywanych wątków. Techniki komunikacji, które zostaną omówione, są zatem wprost zapożyczone z metod komunikowania w systemach współbieżnych. W pewnym uproszczeniu w systemach współbieżnych można wyróżnić dwie podstawowe metody komunikacji: współdzielenie pamięci oraz przekazywanie komunikatów.

Współdzielenie pamięci

W przypadku systemów czasu rzeczywistego szczególną uwagę zwraca się na organizację pamięci służącą do przekazywania informacji. Bardzo częstym środkiem wymiany informacji jest pojedyncza zmienna lub zbiór zmiennych przechowujących tylko bieżącą informację o stanie lub sygnale (wariant pamięci współdzielonej). Przykład takiej wersji komunikacji pokazano na rys. 8.

Wybór takiego podejścia do wymiany informacji ma istotne konsekwencje, zwłaszcza jeśli się założy, że wątki wykonują się cyklicznie. Gdy taka współdzielona zmienna jest aktualizowana często przez jeden wątek i odczytywana co jakiś czas przez inny, wówczas część informacji przekazywanych przez tę zmienną jest tracona przez wątek odczytujący. Z drugiej strony, jeśli informacja w takiej zmiennej jest aktualizowana co jakiś czas, a wątek odczytujący pobiera informację często, to pobiera on tę samą informację wielokrotnie. Warto jednak zauważyć, że pomimo tracenia danych lub ich wielokrotnego odczytywania wątki zawsze dysponują najnowszą wartością współdzielonej zmiennej. Jak się okazuje,

łatwo jest również przewidzieć opóźnienia wynikające ze współzawodnictwa wątków o dostęp do współdzielonych zmiennych. W dostępie do współdzielonych zmiennych należy oczywiście rozwiązać problem wzajemnego wykluczania i wyeliminować możliwość zakleszczenia.



Rysunek 8. Wymiana informacji przez współdzieloną zmienną

Problem wzajemnego wykluczania

Jeśli komunikacja pomiędzy współbieżnie wykonywanymi wątkami odbywa się z zastosowaniem współdzielonej zmiennej, to w celu prawidłowego działania

systemu dostęp do niej musi być chroniony. Oznacza to, że proces zapisu lub odczytu tej zmiennej należy tak zorganizować, aby w chwili wykonywania tych operacji tylko jeden spośród wszystkich wątków miał tę możliwość. Jeśli program spełnia takie założenia, to rozwiązuje jeden z podstawowych problemów systemów współbieżnych: problem wzajemnego wykluczania. Fragment programu wątku, który jako jedyny spośród wszystkich wątków ma być w danej chwili wykonywany, jest nazywany sekcją krytyczną.

Do rozwiązania problemu wzajemnego wykluczania w dostępie do współdzielonych zmiennych w standardzie POSIX są zalecane specjalne obiekty systemu operacyjnego o nazwie mutexy. Na tych obiektach można wykonać operacje: inicjalizacji, zamknięcia, otwarcia oraz usunięcia. Wymienione operacje zestawiono w tab. 1.

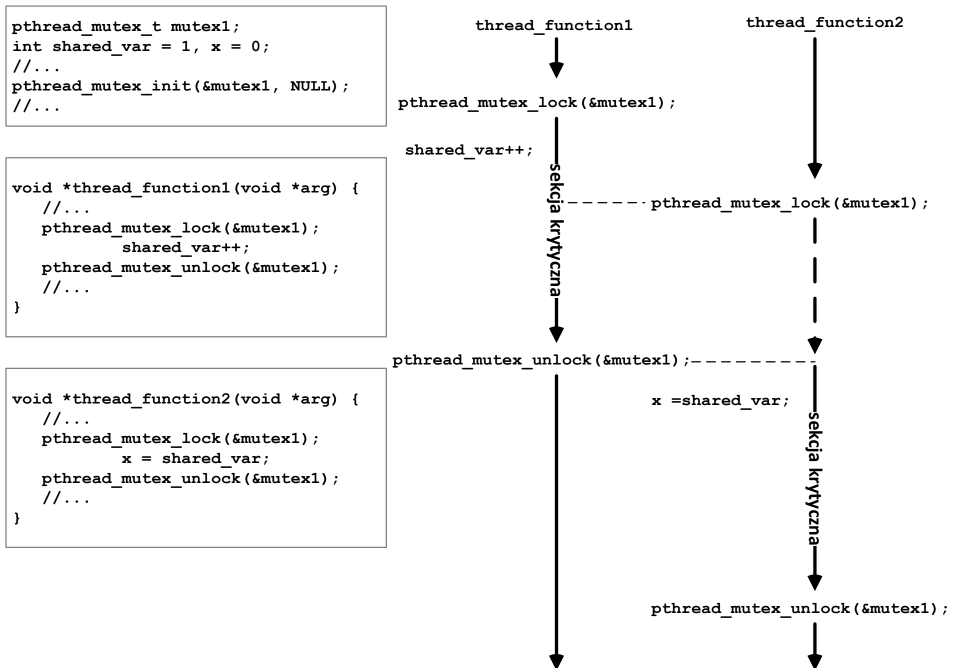
Tabela 1. Operacje na mutexach POSIX

Operacja	Oznaczenie	Opis
Inicjalizacja mutexa	`pthread_mutex_init(&mutex1, &attrs);`	Przyłącz mutex do aplikacji, przyjmij ustalone atrybuty.
Zamknięcie mutexa	`pthread_mutex_lock(&mutex1);`	Gdy mutex nie jest przejęty (zamknięty) przez inny wątek, to go przejmij (zamknij). Gdy mutex jest już przejęty (zamknięty) przez inny wątek, to zablokuj wykonywanie swojego wątku.
Otwarcie mutexa	`pthread_mutex_unlock(&mutex1);`	Gdy istnieje jakiś wątek zablokowany na mutexie, to go odblokuj. Gdy brak jest zablokowanych wątków na danym mutexie, to zwolnij (otwórz) mutex.
Usunięcie mutexa	`pthread_mutex_destroy(&mutex1);`	Odłącz mutex od aplikacji, zwróć go do systemu operacyjnego.

Operacja inicjalizacji (`pthread_mutex_init`) aktywuje mutex w systemie i przekazuje zarządzanie nim bieżącej aplikacji. Pozwala również ustalić jego atrybuty. Ich dobór na potrzeby systemów czasu rzeczywistego zostanie omówiony później. Mutex jest obiektem systemu operacyjnego. Prawidłowe zakończenie korzystania z tego obiektu powinno się wiązać z wywołaniem odpowiedniej funkcji (`pthread_mutex_destroy`), która odłączy go od aplikacji i „zwróci" go do

systemu operacyjnego. Operacje zamknięcia (pthread_mutex_lock) i otwarcia (pthread_mutex_unlock) mutexa są kluczowe dla rozwiązania problemu wzajemnego wykluczania. Ich prawidłowe zastosowanie pokazano na przykładowym scenariuszu zilustrowanym na rys. 9.

Po lewej stronie rysunku znajdują się trzy fragmenty kodu. Pierwszy przedstawia metodę inicjalizacji mutexa. Drugi i trzeci pokazują, jak sekcja krytyczna powinna zostać zaprogramowana. Jeśli mutex ma chronić dostęp do współdzielonej zmiennej, to należy go z nią „skojarzyć" (por. zmiena shared_var i mutex mutex1 na rys. 9.). Samo skojarzenie nie oznacza wywołania specjalnej funkcji, ale podczas przygotowywania oprogramowania warto w ten sposób o zmiennej i mutexie myśleć. Dla takiej zmiennej każda operacja zapisu lub odczytu powinna zostać poprzedzona operacją zamknięcia mutexa (pthread_mutex_lock). Z kolei po dokonaniu operacji należy wywołać operację otwarcia mutexa (pthread_mutex_unlock).



Rysunek 9. Przykładowy scenariusz wzajemnego wykluczania w dostępie do zmiennej

Po prawej stronie rys. 9. zilustrowano przykładowy scenariusz, w którym dwa wątki konkurują o dostęp do współdzielonej zmiennej. W scenariuszu wątek thread_function1 jako pierwszy rozpoczął wykonywanie sekcji krytycznej,

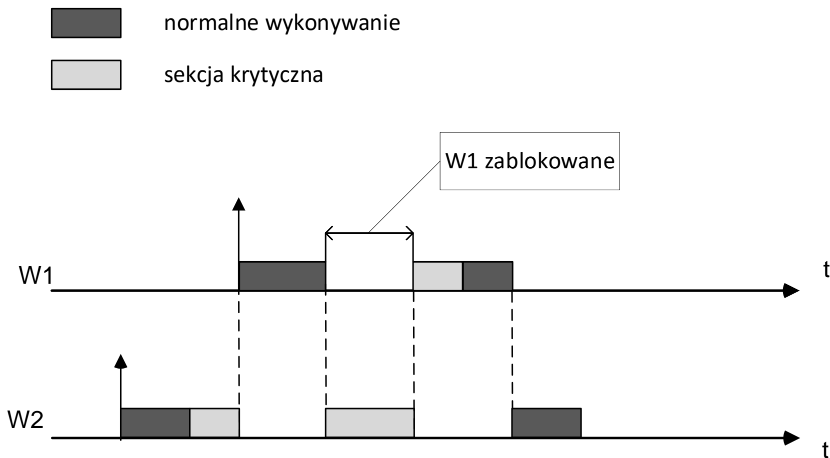
wywołując operację zamknięcia mutexa mutex1 (pthread_mutex_lock). Zgodnie z zasadą działania mutexa pokazaną w tab. 1., ponieważ mutex o nazwie mutex1 nie był wcześniej przejęty (zamknięty) przez jakikolwiek inny wątek, następuje jego przejęcie (zamknięcie). Wątek thread_function1 kontynuuje swoje obliczenia, modyfikując zmienną shared_var. W czasie wykonywania sekcji krytycznej przez wątek thread_function1 drugi wątek (thread_function2) chce rozpocząć wykonywanie swojej sekcji krytycznej. W tym celu wątek thread_function2 również wywołuje funkcję pthread_mutex_lock na tym samym mutexie. Ponieważ mutex mutex1 jest przejęty przez inny wątek, rezultatem wykonania funkcji pthread_mutex_lock jest zablokowanie wykonywania wątku thread_function2. Kiedy wątek thread_function1 kończy sekcję krytyczną, to wywołuje funkcję pthread_mutex_unlock dla mutexa mutex1. Ponieważ w chwili jej wykonywania (zgodnie z regułą wykonywania funkcji) istniał inny wątek zablokowany na tym mutexie, to powoduje ona odblokowanie tego wątku (thread_function2). Zakończenie wykonywania sekcji krytycznej dla wątku thread_function2 oznacza również wywołanie funkcji pthread_mutex_unlock dla mutexa mutex1. Ponieważ nie ma żadnego wątku zablokowanego na tym mutexie, następuje jego zwolnienie (otwarcie).

Scenariusz przedstawiony na rys. 9. ukazuje zasadę sterowania dostępem do zmiennej dzielonej, kiedy konkurują o nią dwa wątki. Okazuje się, że pokazana reguła programowania jest uniwersalna i działa dla systemów złożonych z dowolnej liczby współbieżnie wykonywanych wątków. Gwarantuje ona, że zawsze tylko jeden z wątków w danej chwili może wykonać sekcję krytyczną (np. operację odczytu lub zapisu do współdzielonej zmiennej).

Inwersja priorytetów

Wprowadzenie do aplikacji czasu rzeczywistego mechanizmu wymiany informacji przez chronione współdzielone zmienne może mieć istotny wpływ na czas wykonywania się pojedynczego cyklu obliczeniowego wątku. Okazuje się, że w takiej konfiguracji systemu wątki o niższych priorytetach mogą opóźniać obliczenia wątków posiadających wyższe priorytety. Taki przypadek zilustrowano na rys. 10., gdzie wątek W1 ma przydzielony wyższy priorytet niż wątek W2. W pokazanym scenariuszu wątek W2 otrzymuje jako pierwszy czas procesora, wykonuje obliczenia niewymagające odwoływania się do współdzielonej zmiennej, a następnie rozpoczyna wykonywanie sekcji krytycznej. Podczas trwania sekcji krytycznej system operacyjny otrzymuje informację, że gotowy do wykonywania jest wątek W1. Ma on wyższy priorytet od W2, otrzymuje więc czas procesora. On także wykonuje część obliczeń niewymagających odwoływania się do współ-

dzielonej zmiennej, ale w pewnym momencie zgłasza chęć uzyskania do niej dostępu. Niestety, dostęp do tej zmiennej jest dla tego wątku zablokowany, ponieważ inny wątek (W2) już wcześniej rozpoczął swoją sekcję krytyczną. System operacyjny, aby zachować regułę rozwiązywania problemu wzajemnego wykluczania, musi „oddać" czas procesora wątkowi W2, aby dokończył fragment obliczeń wykonywanych w sekcji krytycznej. Po wykonaniu tego fragmentu obliczeń przez wątek W2, ponownie wątek W1 otrzymuje czas procesora, aż do dokończenia swoich obliczeń. Ostatecznie wątek W2 również kończy swoje wykonywanie.



Rysunek 10. Inwersja priorytetów

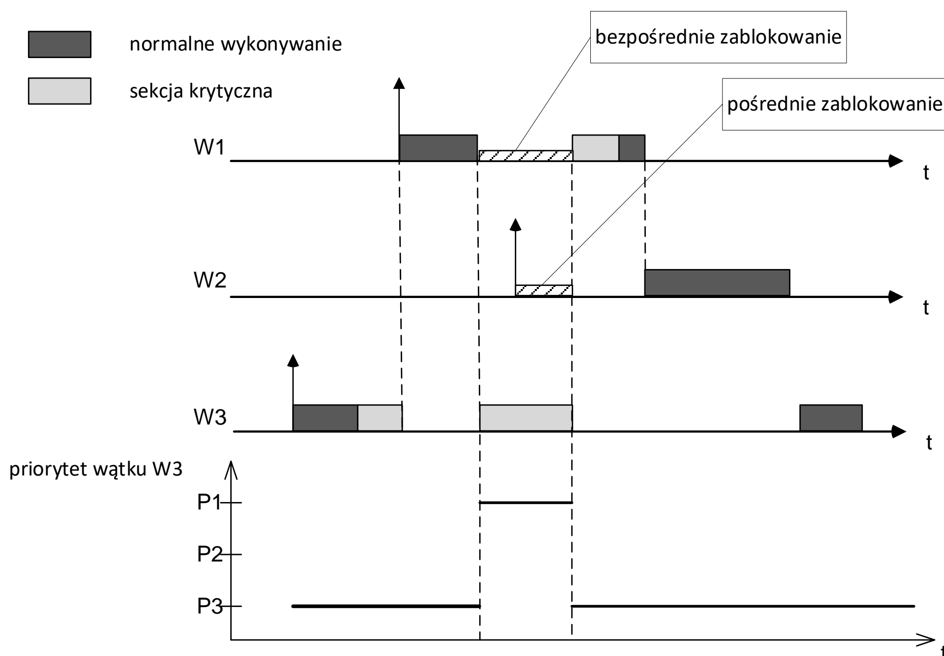
W pokazanym scenariuszu wątek W2 o niższym priorytecie blokował wątek W1 o wyższym priorytecie. Takie zjawisko nosi nazwę inwersji priorytetów. Jeśli w danym systemie następuje komunikacja ze wsparciem algorytmu wzajemnego wykluczania, to takie zjawisko blokowania zawsze w nim istnieje. Można je jednak ograniczyć, stosując protokoły przydziału zasobów sformułowane specjalnie dla aplikacji czasu rzeczywistego.

Protokół dziedziczenia priorytetów

Dla wątków czasu rzeczywistego, dla których pierwotnie przydzielano statyczne priorytety zgodnie z algorytmem planowania ze stałymi priorytetami, opracowano kilka technik przydziału priorytetów w taki sposób, aby minimalizować opóźnienia związane z komunikacją przez współdzieloną pamięć. Metodą wspieraną przez POSIX i zalecaną dla aplikacji czasu rzeczywistego w systemach

Linux jest protokół dziedziczenia priorytetów (ang. Priority Inheritence Protocol). Protokół zakłada, że priorytety procesów mogą ulec zmianie, ale tylko na wyższe niż pierwotnie i tylko w czasie, kiedy dany proces wykonuje sekcję krytyczną. Protokół można sformułować w następujący sposób:

- pierwotnie każdy wątek wykonuje się zgodnie z ustalonym na początku priorytetem;
- kiedy pewien wątek W_I chce wejść do sekcji krytycznej, ale zasób (zmienna dzielona) jest już przejęta przez inny wątek W_J o niższym priorytecie, to następuje jego wstrzymanie;
- kiedy wątek W_I jest zablokowany, to przekazuje swój priorytet wątkowi W_J, który przejął wcześniej zasób. Wątek W_J kończy więc wykonywanie sekcji krytycznej z wyższym priorytetem. Mówi się, że W_J odziedziczył priorytet pa W_I;
- po zakończeniu sekcji krytycznej wątek W_J powraca do pierwotnego priorytetu lub dziedziczy priorytet innego wątku, z którym „wygrał" współzawodnictwo o współdzielony zasób.



Rysunek 11. Przykład działania protokołu dziedziczenia priorytetów

Przykładowy scenariusz ilustrujący zasadę działania protokołu dziedziczenia priorytetów pokazano na rys. 11. Wątek W3 jako pierwszy rozpoczyna wykonywanie swoich obliczeń, przejmuje również współdzielony zasób. Podczas wykonywania sekcji krytycznej zostaje zablokowany, a czas procesora przekazany wątkowi W1. Kiedy wątek W1 żąda udostępnienia mu współdzielonego zasobu, zostaje zawieszony, a czas procesora z powrotem zostaje przekazany wątkowi W3. Wątek W3 jest wykonywany wtedy z wyższym priorytetem, takim jaki ma wątek W1. Nawet pojawienie się w zbiorze wątków gotowych do wykonywania wątku W2 nie przerywa wykonywania wątku W3, gdy ten wykonuje sekcję krytyczną. Kiedy wątek W3 kończy sekcję krytyczną, otrzymuje z powrotem swój pierwotny priorytet. Kolejne obliczenia poszczególnych wątków są wykonywane zgodnie z pierwotnymi priorytetami. Jak już wspomniano, protokół dziedziczenia priorytetów nie eliminuje dodatkowych opóźnień w wykonywaniu obliczeń. Ma on za zadanie skrócenie czasów odpowiedzi procesów o wyższych priorytetach przez przyspieszenie zwalniania zasobów (zmiennych dzielonych) przejmowanych przez wątki o niższych priorytetach.

Na rysunku 11. zwrócono uwagę na ciekawe zjawisko, które może wystąpić podczas wykonywania protokołu dziedziczenia priorytetów. Wątek W3, który na pewien etap wykonywania swoich obliczeń odziedziczył priorytet wątku W1, pośrednio wstrzymał wykonywanie wątku W2, chociaż nie współzawodniczył z nim o dostęp do współdzielonych zasobów (zmiennych).

Warunek szeregowalności wątków komunikujących się za pomocą zmiennej dzielonej

Jeśli poszczególnym wątkom przydzielono priorytety zgodnie z regułą „rate monotonic”, wprowadzono do aplikacji metodę komunikacji przez współdzieloną zmienną, ale ograniczoną przez protokół dziedziczenia priorytetów, to można wykazać, że dany zbiór wątków jest szeregowalny, jeśli spełnia zależność:

$$\forall i, 1 \leq i \leq n, \qquad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i\left(2^{\frac{1}{i}} - 1\right).$$

Osobno dla każdego wątku od 1 do n należy sprawdzić, czy dany wątek i wątki o wyższych priorytetach są szeregowalne zgodnie z regułą „rate monotonic”. Ponadto trzeba uwzględnić największe z możliwych zablokowanie, jakiego dany wątek może doświadczyć z powodu przejmowania współdzielonych zmiennych przez wątki o niższych priorytetach (składnik B_i/T_i w nierówności).

Wielowątkowy program czasu rzeczywistego z wymianą danych przez zmienną dzieloną

Program z listingu 4. (bez definicji i wywołania funkcji `thread_attributes_test`) uzupełniony o komunikację z zastosowaniem współdzielonej zmiennej, chronionej przez mutex z włączonym protokołem dziedziczenia priorytetów pokazano na listingu 5.

Listing 5. Wielowątkowy program z komunikacją przez zmienną współdzieloną

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mutex_attr;
int shared_var = 0;

void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1;
    struct itimerspec its1;
    struct sigevent sev1;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                                 // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 1; // 1-sekundowy interwał
    its1.it_interval.tv_nsec = 0;
```

```
    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&mutex);
    printf("Watek 1: obliczenia wznowione, ");
    printf("shared_var = %d\n", shared_var);
    pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}

void *thread2(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
    timer_create(CLOCK_REALTIME, &sev2, &timer2);

    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                                 // 2 sekundy opoźnienia
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;

    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);

    while (1) {
    sigwait(&set, &sig);
    pthread_mutex_lock(&mutex);
    printf("Watek 2:obliczenia wznowione, ");
    printf("shared_var=%d\n", ++shared_var);
    pthread_mutex_unlock(&mutex);
    }

    pthread_exit(NULL);
}
```

```
int main() {
    pthread_t t1, t2;
    pthread_attr_t attr1, attr2;
    struct sched_param param1, param2, param_main;
    cpu_set_t cpuset;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);

    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset); //Wybierz procesor nr 2 w systemie
    printf("Liczba dostepnych CPU: ");
    printf("%d\n", (int) sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t), &cpuset);

    pthread_mutexattr_init(&mutex_attr);
    pthread_mutexattr_setprotocol(&mutex_attr,
                               PTHREAD_PRIO_INHERIT);
    pthread_mutex_init(&mutex, &mutex_attr);

    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
                              PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                              sizeof(cpu_set_t),
                              cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_attr_init(&attr2);
    pthread_attr_setscope(&attr2, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr2,
                              PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
```

```
    param2.sched_priority =
                  sched_get_priority_max(SCHED_FIFO) - 2;
    pthread_attr_setschedparam(&attr2, &param2);
    pthread_create(&t2, &attr2, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

W programie zdefiniowano trzy nowe zmienne globalne:

- zmienną do identyfikacji mutexa (typ: `pthread_mutex_t`);
- zmienną do przechowywania atrybutów mutexa (typ: `pthread_mutexattr_t`);
- zmienną do przechowywania współdzielonej wartości.

Zgodnie ze specyfikacją języka C zmienne globalne są widoczne dla innych składników programu od miejsca utworzenia do końca pliku. Wymienione zmienne (zadeklarowane na początku pliku) są więc widoczne dla wszystkich funkcji przykładowego programu. Pozwala to w naturalny sposób zorganizować wymianę informacji zgodną ze schematem zmiennej dzielonej. W wątku `thread1` współdzielona zmienna jest odczytywana i wypisywana na ekranie, natomiast w wątku `thread2` zmienna jest inkrementowana i wyświetlana na ekranie. W obu wątkach dostęp do zmiennej jest chroniony z zastosowaniem mutexa (odpowiednie wywołanie pary funkcji `pthread_mutex_lock` i `pthread_mutex_unlock`). Inicjalizacja mutexa jest wykonywana przed utworzeniem wątków w funkcji `main`. Przed wywołaniem funkcji inicjalizującej mutex (`pthread_mutex_init`) inicjalizowana jest struktura do przechowywania atrybutów mutexa (`pthread_mutexattr_init`). Atrybuty wątku są z kolei ustawiane na takie, które spowodują, że będzie on wypełniał protokół dziedziczenia priorytetów (`PTHREAD_PRIO_INHERIT`).

W omawianym programie zadaniem programisty jest tylko uruchomienie protokołu dziedziczenia priorytetów, a nie manipulowanie priorytetami wątków. Tą sferą wykonywania programu zajmuje się system operacyjny.

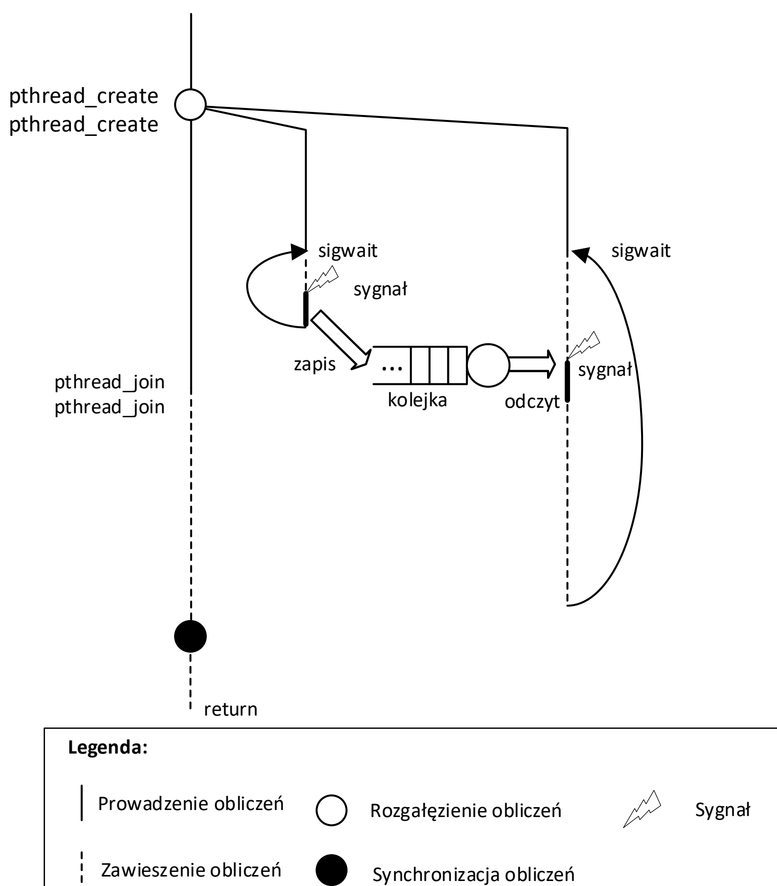
Technika uruchamiania aplikacji

Do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Przykładowe polecenie uruchamiające program może

wyglądać następująco: *„sudo ./program”*. System operacyjny może wymusić wtedy podanie hasła administratora.

Kolejki komunikatów

Podczas projektowania współbieżnych systemów czasu rzeczywistego warto również rozważyć taką technikę komunikacji międzywątkowej, która pozwala niezawodnie przekazać strumienie danych (model komunikacji w postaci przekazywania komunikatów). Niezawodnie i bezstratnie przekazywane strumienie (pliki z danymi) pozwalają np. na instalacje nowych konfiguracji systemu. Do takiego wariantu komunikacji służą mechanizmy ogólnie nazywane kolejkami komunikatów. Schematycznie taki wariant komunikacji przedstawiono na rys. 12.



Rysunek 12. Komunikacja z zastosowaniem kolejki komunikatów

Przekazywanie danych przez kolejki komunikatów w systemie współbieżnym, zwłaszcza takim, który wykonuje cykliczne obliczenia, ma swoją specyfikę. Po pierwsze, w odróżnieniu od zmiennych dzielonych kolejka komunikatów „pamięta” kolejne komunikaty, które są do niej wysyłane, dopóki inny składnik systemu danego komunikatu nie odbierze. Pozwala zatem niezawodnie przekazać strumień danych bez utraty informacji. Po drugie, operacje na kolejkach są wykonywane z zastosowaniem wzajemnego wykluczania. W jednym czasie tylko jeden wątek dokonuje zapisu lub odczytu danych z kolejki. Ma to jednak swoje konsekwencje. W praktyce kolejka zawsze ma ograniczoną pojemność. Jeśli kolejka zostanie utworzona w standardowych ustawieniach, to próba odczytu danych z pustej kolejki blokuje wątek odczytujący, a próba zapisu do przepełnionej kolejki blokuje wątek zapisujący. Jeśli wątki mają się wykonywać cyklicznie i w ustalonych reżimach czasowych, to stosowanie kolejek w standardowych ustawieniach może powodować nieprzewidywalne opóźnienia i utratę założonej terminowości pracy systemu.

Aby uniknąć nieprzewidzianych blokad na kolejkach podczas wykonywania aplikacji czasu rzeczywistego, dostęp do kolejek musi zostać ustawiony na „nieblokujący” (ang. nonblocking). W takim trybie pracy kolejki próba zapisu danych do przepełnionej kolejki lub próba odczytu danych z pustej kolejki zwróci informację o błędzie, ale nie zablokuje wykonywania wątku.

Wielowątkowy program czasu rzeczywistego z wymianą danych przez kolejkę

Program z listingu 4. (bez definicji i wywołania funkcji `thread_attributes_test`) uzupełniony o komunikację z zastosowaniem nieblokującej kolejki pokazano na listingu 6.

Listing 6. Wielowątkowy program z komunikacją przez nieblokującą kolejkę komunikatów

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <mqueue.h>
#include <errno.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)
```

```
mqd_t mq;
struct message{
    int counter;
};

void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1;
    struct itimerspec its1;
    struct sigevent sev1;
    struct timespec now;
    struct message msg;
    msg.counter = 1;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                                    // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 1; // 1-sekundowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
        sigwait(&set, &sig);
        if(mq_send(mq, (char*)&msg, sizeof(msg), 0) == -1)
{
            if(errno == EAGAIN) {
                printf("Kolejka jest przepełniona\n");
            }
        } else {
            struct mq_attr attr;
            mq_getattr(mq,&attr);
            printf("Liczba wiad. w kolejce: ");
            printf(" %d\n",(int) attr.mq_curmsgs);
            msg.counter++;
        }
    }
    pthread_exit(NULL);
}
```

```
void *thread2(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;
    struct message received;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
    timer_create(CLOCK_REALTIME, &sev2, &timer2);

    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                                  // 2 sekundy opóźnienia
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;

    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);

    while (1) {
        sigwait(&set, &sig);
          if(mq_receive(mq, (char*)&received,
                        sizeof(received), 0) == -1) {
                if(errno == EAGAIN) {
                    printf("Kolejka jest pusta\n");
                    break;
                }
          } else {
                printf("Odebrana wartosc: ");
                printf("%d\n", received.counter);
            }
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_attr_t attr1, attr2;
    struct sched_param param1, param2, param_main;
    cpu_set_t cpuset;
```

```
    struct mq_attr attr;
    sigset_t set;

    attr.mq_maxmsg = 10;
    attr.mq_msgsize = sizeof(struct message);
    attr.mq_curmsgs = 0;
    mq = mq_open("/queue", O_CREAT | O_RDWR | O_NONBLOCK,
0644, &attr);

    if (mq != (mqd_t) -1) {
        mq_close(mq);
        mq_unlink("/queue"); // Usuń istniejącą kolejkę
    } else if (errno != ENOENT) {
        fprintf(stderr, "Nie można otworzyć kolejki");
        fprintf(stderr, "komunikatów.\n");
        exit(EXIT_FAILURE);
    }
    // Utwórz nową kolejkę:
    mq = mq_open("/queue", O_CREAT | O_RDWR | O_NONBLOCK,
                 0644, &attr);

    if (mq == (mqd_t) -1) {
        fprintf(stderr, "Nie można otworzyć kolejki");
        fprintf(stderr, "komunikatów.\n");
        exit(EXIT_FAILURE);
    }

    sigemptyset(&set);
    sigaddset(&set, SIG);
    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset); //Wybierz procesor nr 2 w systemie
    printf("Liczba CPU: %d\n",
           (int)sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t),
                           &cpuset);

    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
```

```
                                    PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                  sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                                sizeof(cpu_set_t),
                                &cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_attr_init(&attr2);
    pthread_attr_setscope(&attr2, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr2,
                                  PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
    param2.sched_priority =
                   sched_get_priority_max(SCHED_FIFO) - 2;
    pthread_attr_setschedparam(&attr2, &param2);
    pthread_create(&t2, &attr2, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    mq_close(mq);
    mq_unlink("/queue");

    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);

    return 0;
}
```

Każda kolejka komunikatów zgodna ze standardem POSIX jest dostępna z punktu widzenia programu przez zmienną typu `mqd_t` (udostępniającą po utworzeniu kolejki „uchwyt” do niej). Wiadomości przekazywane z zastosowaniem kolejek można podzielić na pola i zdefiniować jako strukturę języka C (por. struktura `struct message`). Zdefiniowanie podstawowych właściwości każdej kolejki polega na przygotowaniu zmiennej do przechowywania atrybutów kolejki (typu `struct mq_attr`) i wypełnieniu jej pól założonymi wartościami. W programie z listingu 6. ustalono długość kolejki na 10 komunikatów (`attr.mq_maxmsg = 10`), a rozmiar pojedynczego komunikatu na rozmiar struktury `struct message` (`attr.mg_msgsize = sizeof(struct message)`).

Utworzenie kolejki odbywa się przez wywołanie funkcji `mq_open`. Parametrami wywołania funkcji są: nazwa kolejki, flagi decydujące o trybie pracy kolejki, uprawnienia nadawane dla kolejki oraz atrybuty. Kolejka komunikatów

POSIX jest widoczna w systemie jako pseudoplik, który ma swoją nazwę w drzewie katalogów (w programie: „\queue”). Flagi zastosowane w przykładowym programie decydują o tym, że kolejka ma być utworzona. Jeśli nie istniała wcześniej w systemie, ma być otwarta zarówno do operacji zapisu, jak i odczytu, dostęp do niej ma być nieblokujący (O_NONBLOCK). Właściciel kolejki (program) ma prawa do zapisu i odczytu danych, natomiast grupa i inni mogą tylko odczytywać dane. Można zauważyć, że funkcja mq_open jest w przykładowym programie uruchomiona dwukrotnie. Fragment programu pomiędzy tymi uruchomieniami ma na celu usunięcie z systemu tej samej kolejki utworzonej przez wcześniejsze działanie tego samego programu. W założeniu poprzednia kolejka o tej samej nazwie jest niszczona i na jej miejsce powoływana jest nowa. Prawidłowe zamknięcie dostępu do kolejki oraz usunięcie jej z systemu odbywa się przez wywołanie odpowiednio funkcji mq_close i mq_unlink.

Przykładowa aplikacja, potencjalnie w nieskończoność, wysyła komunikaty do kolejki (por. kod funkcji thread1) i odbiera je w innym wątku (por. kod funkcji thread2). Wysyłanie komunikatów polega na wywoływaniu funkcji mq_send. Jej parametrami są uchwyt do kolejki, wskazanie na miejsce w pamięci, gdzie znajduje się komunikat do wysłania, rozmiar komunikatu oraz priorytet komunikatu. Komunikacja przez kolejkę jest ustawiona w tryb nieblokujący. W tym trybie próba wysłania komunikatu do przepełnionej kolejki powoduje, że funkcja mq_send zwraca wartość –1, a w zmiennej do przechwytywania ostatnich błędów (errno) ustawiana jest wartość EAGAIN. Wątek nadający komunikaty jest w stanie wychwycić takie zdarzenie i wyświetlić komunikat na konsoli. W przypadku pomyślnego zapisu do kolejki wątek, przez bieżący odczyt atrybutów kolejki, bada liczbę wiadomości w niej zapisanych i wyświetla o tym informację. Odbieranie komunikatów z kolejki odbywa się przez wywołanie funkcji mq_receive. Parametrami funkcji są: uchwyt do kolejki, wskazanie na miejsce pamięci, gdzie ma zostać pobrany komunikat, rozmiar pobranego komunikatu oraz priorytet odebranego komunikatu. Podobnie jak w przypadku nadawania danych, próba odczytu z pustej kolejki powoduje zwrócenie przez funkcję mg_receive wartości –1. Zbadanie stanu zmiennej errno pozwala na potwierdzenie przyczyny błędu.

Technika uruchamiania aplikacji

Do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Przykładowe polecenie uruchamiające program może wyglądać następująco: *„sudo ./program”*. System operacyjny może wtedy wymusić podanie hasła administratora. Uruchomienie aplikacji z założonymi parametrami czasowymi jak w programie z rys. 11. spowoduje w krótkim czasie przepełnienie kolejki. Wątek nadający będzie zgłaszał o tym komunikaty.

Techniki komunikacji międzyprocesowej – uzupełnienie

Przykładowe aplikacje pokazane na listingach 5. i 6. ilustrują techniki komunikacji pomiędzy wątkami należącymi do pojedynczej aplikacji. Warto wiedzieć, że można je efektywnie stosować do wymiany informacji w systemach złożonych z wielu osobno uruchamianych programów.

Aby wprowadzić w takich systemach model komunikacji przez współdzieloną zmienną, należy zdefiniować z zastosowaniem odpowiednich funkcji obszar pamięci dzielonej utworzony poza procesami w pamięci systemu operacyjnego. Do takiego obszaru pamięci może się „przyłączyć" dowolna liczba procesów i traktować go jako medium zmienną współdzieloną. Warto zwrócić uwagę, że dostęp do takiego współdzielonego zasobu powinien być chroniony z zastosowaniem mutexa zdolnego do rozwiązywania problemu wzajemnego wykluczania na poziomie komunikacji międzyprocesowej. Można wykorzystać wspomnianą pamięć dzieloną do przechowywania stanu tego mutexa.

Wdrożenie międzyprocesowej komunikacji przez kolejki komunikatów nie wymaga dodatkowych zabiegów. Wzorując się na listingu 6., można w jednym z procesów wykonać operację utworzenia kolejki, a następnie pozwolić zainteresowanym stronom komunikacji na „przyłączenie" się do kolejki w charakterze producentów lub konsumentów.

Literatura uzupełniająca

Pozycjami literatury pokazującymi zasady konstruowania komunikacji między wątkami czasu rzeczywistego są prace: [But2011], [BW2009], [CDKM2002], [SSRB1998], [Uła2007]. Techniki programowania komunikacji międzyprocesowej oraz międzywątkowej w ujęciu systemów czasu rzeczywistego oraz w ujęciu systemów ogólnego przeznaczenia w standardzie przystępnie pokazano w publikacjach: [Gal1995] , [BW2009], [Uła2007], [MS2007], [Fus2007], [Ami2021].

Podczas wymiany informacji pomiędzy współbieżnie wykonywanymi wątkami może zajść kilka zjawisk, które mogą wpływać na jakość wykonywanej aplikacji. Pierwsze, już wcześniej zasygnalizowane, polegało na rywalizacji o dostęp do współdzielonej zmiennej. Jeśli kilka wątków zapisuje do jednej zmiennej dane, to ostateczna jej wartość zależy od sekwencji tych zapisów. Na tę sekwencję nie mamy niestety wpływu. Warto więc opracować czytelne scenariusze wymiany informacji przez zmienne dzielone, np. przyjmując, że wskazany wątek w systemie modyfikuje daną zmienną, a pozostałe tylko ją odczytują. Wprowadzenie do systemu współbieżnego rozwiązania problemu wzajemnego wykluczania może spowodować dwa kolejne niekorzystne zjawiska: głodzenie oraz zakleszczenie. W przypadku głodzenia w systemie mogą istnieć ważniejsze wątki, które nigdy nie dopuszczą mniej ważnego wątku do współdzielonego zasobu. Potencjalnie

taki mniej ważny wątek może w nieskończoność oczekiwać na wejście do sekcji krytycznej, ale nigdy do niej nie wejść. Zakleszczenie jest jeszcze groźniejszym zjawiskiem, ponieważ może, przy niefortunnie zaplanowanej strategii dostępu do współdzielonych zasobów, doprowadzić do nieodwracalnego zablokowania wykonywania wątków. Zasygnalizowane problemy występujące w wytwarzaniu systemów współbieżnych w przystępny sposób omówiono w pracach [Sta2018] oraz [Ami2021].

W przykładowej aplikacji prezentowanej w tym rozdziale przedstawiono zaimplementowany i sugerowany do stosowania we wszystkich dystrybucjach Linuxa protokół dziedziczenia priorytetów (ang. Priority Inheritance Protocol). Ogranicza on inwersję priorytetów w dostępie do współdzielonych zasobów i jest opisany w pracach [But2011] oraz [BW2009]. Dla zadań szeregowanych zgodnie ze strategią Rate Monotonic został opracowany jeszcze lepszy protokół o nazwie pułapowy protokół priorytetów (ang. Priority Ceiling Protocol). W odróżnieniu od protokołu dziedziczenia priorytetów zapobiega on pojawianiu się zakleszczeń oraz dodatkowego blokowania tych wątków, które nie współzawodniczą o zasób (por. komentarz do rys. 15.). W standardzie POSIX jest przygotowany zestaw funkcji pozwalający na włączenie tego protokołu. W chwili przygotowywania skryptu nie znaleziono jednak dystrybucji systemu Linux oficjalnie wspierającej to rozwiązanie. Formalną definicję protokołu i konsekwencje wynikające z jego stosowania zawierają między innymi prace [But2011] oraz [BW2009].

Dla wspomnianej w poprzednim rozdziale strategii dynamicznego przydziału priorytetów dla wątków czasu rzeczywistego (ang. Earliest Deadline First) również opracowano strategię przydziału zasobów o nazwie Stack Resource Policy, omówioną między innymi w książkach [But2011], [SSRB1998]. W przypadku systemów wieloprocesorowych, dla których szeregowanie wątków odbywa się w sposób globalny, również opracowano protokoły przydziału zasobów. Na uwagę zasługują wieloprocesorowe wersje protokołów wspomnianych wcześniej: Multiprocessor Priority Ceiling Protocol, Distributed Priority Ceiling Protocol czy Multiprocessor Stack Resource Policy. Studia nad tymi protokołami warto rozpocząć od artykułu [RCh2009].

Zadania

1. Na czym polega inwersja priorytetów?
2. Jak działa protokół dziedziczenia priorytetów?
3. Czym różni się dostęp blokujący od nieblokującego dostępu do kolejki komunikatów?
4. Sprawdź szeregowalność dla następującego zbioru zadań czasu rzeczywistego: {(T_1 = 70 μs, C_1 = 10 μs), (T_2 = 100 μs, C_2 = 20 μs), (T_3 = 150 μs, C_3 =

= 30 µs)}. Zadanie T_3 może być zablokowane przez zadanie T_2 na 10 µs z powodu współdzielenia przez nie zasobu. Zadania mają być szeregowane z użyciem algorytmu Rate Monotonic, natomiast zastosowanym protokołem przydziału zasobów będzie protokół dziedziczenia priorytetów (ang. Priority Inheritance Protocol).

5. Jakie są zalety zastosowania pułapowego protokołu priorytetów? Czy eliminuje on inwersję priorytetów?

Receptura 5.
– techniki komunikacji sieciowej

Naturalnym rozwinięciem lokalnego systemu czasu rzeczywistego działającego na pojedynczym komputerze jest włączenie go do sieci komputerowej. Oczekuje się, że integracja z siecią umożliwi budowanie rozproszonych aplikacji czasu rzeczywistego. Okazuje się jednak, że typowe sieci klasy Ethernet, najpowszechniej stosowane do lokalnych sieci komputerowych, z uwagi na przyjętą zasadę działania nie mogą być wprost traktowane jako medium, gdzie można zawsze przewidzieć ewentualne opóźnienia w komunikacji. W konsekwencji stworzenie przewidywalnej, rozproszonej aplikacji czasu rzeczywistego wymaga specjalizowanych sieci komputerowych.

W tej części skryptu zaproponowano nieco odmienne podejście. Założono, że nie jest możliwe precyzyjne określenie terminowości przesyłania danych przez sieć, ale na poziomie aplikacji można włączyć dodatkowe mechanizmy, które będą monitorować opóźnienia w transmisji. Okazuje się, że w wielu przypadkach takie podejście wystarcza do efektywnego tworzenia rozproszonych aplikacji czasu rzeczywistego o akceptowalnych właściwościach czasowych.

Dwa podstawowe warianty komunikacji sieciowej

Do komunikacji przez sieć komputerową stosuje się najczęściej dwa protokoły: UDP (ang. User Datagram Protocol) oraz TCP (ang. Transmission Control Protocol). Pierwszy z nich jest bezpołączeniowy, co w konsekwencji oznacza, że nadawca wysyła pakiety (datagramy) z danymi, nie wiedząc, czy w danej chwili odbiorca je odbiera. Protokół nie posiada w zasadzie żadnych mechanizmów regulujących szybkość nadawania danych. Drugi z nich pozwala na przesyłanie informacji tylko wtedy, gdy obie strony potwierdzą swoją gotowość do współpracy. Zapewnia również pewien poziom niezawodności, ponieważ samodzielnie może „poprosić" o ponowne przesłanie brakujących danych, a następnie włączyć je w odpowiednie miejsce przesyłanego strumienia. Ma również wbudowane mechanizmy regulacji tempa przesyłanych informacji (np. zażądanie rzadszego przesyłania porcji informacji z uwagi na przeciążenie kanału komunikacyjnego).

Pewnym zaskoczeniem może być fakt, że mimo wielu zalet protokołu TCP, to protokół UDP dominuje w aplikacjach czasu rzeczywistego. Łatwiej jest się z tą wiadomością oswoić, jeśli odniesie się ją do preferowanego w systemach czasu rzeczywistego modelu komunikacji, czyli współdzielenia zmiennych. Dane prze-

syłane przez protokół UDP łatwo sobie wyobrazić jako kolejne komunikaty aktualizujące stan (zbiór zmiennych) systemu. Preferowaną sytuacją jest tu otrzymywanie najbardziej aktualnych informacji o stanie wybranych zmiennych w postaci regularnie nadsyłanych pakietów. Do zaakceptowania wydaje się utrata kilku pakietów z informacją, natomiast mechanizm ponownego nadsyłania przestarzałych danych, które zostały kiedyś „zgubione" (jest to standardowa właściwość protokołu TCP) może być postrzegany jako niepotrzebne zakłócenie linii czasowej pozyskiwania informacji.

Podsumowując, preferowanym protokołem sieciowym stosowanym w systemach czasu rzeczywistego jest UDP. Protokół TCP jest również stosowany, ale przede wszystkim do niezawodnego przesyłania strumieni danych, np. plików konfiguracyjnych.

Aplikacja sieciowa opierająca się na protokole UDP

Jedną z popularniejszych metod tworzenia oprogramowania sieciowego jest stosowanie tak zwanego interfejsu gniazd (ang. sockets). Z punktu widzenia sieci komputerowych posługujących się rodziną protokołów TCP/UDP/IP gniazdo jednoznacznie konfiguruje kanał komunikacyjny pomiędzy dwoma komputerami. Po każdej stronie kanału komunikacyjnego definiuje się parę danych: adres IP i port, na którym nadawca nadaje dane oraz adres IP i port, na którym odbiorca odbiera dane. Protokół UDP, jak już wspomniano, przekazuje komunikaty/datagramy. Zawartość komunikatu zależy od programisty. Często programom, które komunikują się w sieci, nadaje się wyróżnione role: klienta lub serwera. W założeniu serwer oczekuje na połączenia, a następnie wymienia informacje z klientami. Na listingu 7. pokazano przykładowy program klienta.

Listing 7. Przykładowy program klienta w komunikacji z zastosowaniem UDP

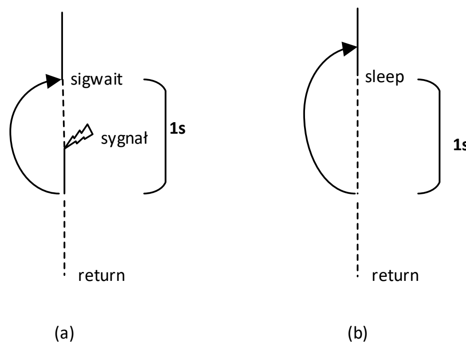
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <time.h>

#define PORT 8080
#define SERVER_IP "127.0.0.1"

typedef struct {
    int number;
    struct timespec timestamp;
```

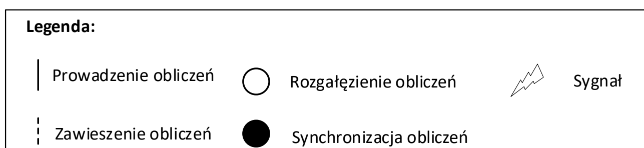
```
} Data;

int main() {
    int sockfd;
    struct sockaddr_in servaddr;
    int n = 100;
    Data data;

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr(SERVER_IP);

    while (1) {
        struct timespec ts;
        data.number = n++;
        clock_gettime(CLOCK_REALTIME, &data.timestamp);

        sendto(sockfd, &data, sizeof(Data),
            MSG_CONFIRM, (const struct sockaddr*)&servaddr,
            sizeof(servaddr));

        ts.tv_sec = 1;
        ts.tv_nsec = 0;
        nanosleep(&ts, NULL);
    }
    close(sockfd);
    return 0;
}
```

Program jest dostosowany do uruchomienia na jednym komputerze. Posługuje się adresem IP lokalnego interfejsu sieciowego (127.0.0.1) oraz wybranym portem 8080, gdzie spodziewa się, że będzie mógł przekazywać komunikaty. Kluczowym mechanizmem zastosowanym do komunikacji w programie jest interfejs gniazd. Rozpoczęcie połączenia zaczyna się od zdefiniowania własnego gniazda (por. wywołanie funkcji `socket`). Podane parametry funkcji ustalają, że komunikacja będzie się odbywała przez sieć komputerową z zastosowaniem protokołów IPv4 oraz UDP. Przesłanie komunikatu do wskazanego adresu IP i portu wymaga wypełnienia struktury typu `struct sockaddr_in`. Przekazuje się do niej wybrany protokół, port i adres IP serwera. Ostatecznie przesłanie pojedynczego

komunikatu odbywa się przez wykonanie funkcji `sendto`. Funkcja ta z zastosowaniem zdefiniowanego gniazda wysyła wiadomość (por. zmienna `data`) o pewnej długości wraz z zawartością struktury danych informującej o wybranych parametrach komunikacji (por. zmienna `servaddr`).

W przykładowym programie klienta wysyłana jest struktura danych złożona z licznika (pole `nuber`) oraz pieczątki czasowej (pole `timestamp`). Wartością początkową licznika jest 100 i po każdym wysłaniu komunikatu jest ona zwiększana o 1. Pieczątka czasowa zawiera informację o stanie czasu systemowego tuż przed wysłaniem danych przez interfejs sieciowy. Wysłanie pojedynczego komunikatu odbywa się co 1 s. Tym razem w odróżnieniu od wcześniejszych programów zastosowano inną metodę cyklicznego wykonywania programu, w przybliżeniu co 1 s. W pętli, po wykonaniu instrukcji przekazania danych przez interfejs gniazd wywoływana jest funkcja `nanosleep`, która zawiesza wykonywanie programu na zadany przedział czasu. Rezultat działania programu może się wydawać podobny do wcześniejszych przykładów, gdzie również co 1 s program się budził i wznawiał swoje obliczenia. Należy jednak zwrócić uwagę, że tym razem 1-sekundowe opóźnienie jest doliczane do czasu wykonywania obliczeń w pętli. W poprzednich przykładach struktura programu, niezależnie od długości wykonywania obliczeń w pętli, za każdym razem co 1 s wznawiała obliczenia i to ta wcześniejsza metoda sterowania zdarzeniami czasowymi jest zalecana do precyzyjnego cyklicznego wznawiania obliczeń. Różnicę w zasadzie wykonywania cyklicznych wątków zaprogramowanych pierwszym i drugim sposobem pokazano na rys. 13.





Rysunek 13. Sterowanie cyklem wykonywania wątku z zastosowaniem sygnału (a) lub z zastosowaniem opóźnienia (b)

Na listingu 8. pokazano program serwera przykładowej aplikacji sieciowej stosującej wymianę komunikatów z zastosowaniem protokołu UDP.

Listing 8. Przykładowy program serwera w komunikacji z zastosowaniem UDP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <time.h>

#define PORT 8080
#define SERVER_IP "127.0.0.1"

typedef struct {
    int number;
    struct timespec timestamp;
} Data;

Data shared_data;
pthread_mutex_t lock;

void* receive_data(void* arg) {
    int sockfd = *((int*)arg);
    struct sockaddr_in cliaddr;
    socklen_t len;
    char buffer[sizeof(Data)];

    while (1) {
        recvfrom(sockfd, (char*)buffer, sizeof(Data),
            MSG_WAITALL, (struct sockaddr*)&cliaddr,
            &len);

        Data* data = (Data*)buffer;

        pthread_mutex_lock(&lock);
        shared_data = *data;
        pthread_mutex_unlock(&lock);
    }
```

```
    return NULL;
}

int main() {
    int sockfd;
    struct sockaddr_in servaddr;

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr(SERVER_IP);

    if (bind(sockfd, (const struct sockaddr*)&servaddr,
        sizeof(servaddr)) < 0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    pthread_t thread_id;
    pthread_create(&thread_id, NULL, receive_data,
&sockfd);

    while (1) {
        struct timespec ts = {2,0};
        pthread_mutex_lock(&lock);
        Data data = shared_data;
        pthread_mutex_unlock(&lock);

        struct timespec now;
        clock_gettime(CLOCK_REALTIME, &now);
        double delay =
        (now.tv_sec - data.timestamp.tv_sec)
        +
        (now.tv_nsec - data.timestamp.tv_nsec) / 1e9;


        printf("Received number: %d\n", data.number);
        printf("Delay: %.9f seconds\n", delay);
```

```
        nanosleep(&ts,NULL);
    }

    close(sockfd);
    return 0;
}
```

Wykonywanie głównej funkcji programu z listingu 8. rozpoczyna się od zdefiniowania gniazda. Parametry wywołania funkcji `socket` są identyczne jak w programie z listingu 7. Podobną zawartość uzyskuje również struktura typu `struct sockaddr_in`. Po stronie serwera w chwili wywołania funkcji `bind` następuje przypisanie do gniazda wybranego portu i adresu IP.

Za odbieranie komunikatów jest odpowiedzialny osobny wątek sterowany funkcją `receive_data`. W czasie uruchomienia otrzymuje uchwyt do gniazda, przez które będą dostarczane komunikaty. W funkcji wątku zostaje uruchomiona pętla, gdzie serwer, wywołując funkcję `recvfrom`, może odbierać komunikat, który został do niego wysłany. W przykładowej aplikacji, po otrzymaniu komunikatu odebrana struktura jest przekazywana do współdzielonej zmiennej. Główny wątek (funkcja `main`) co 2 s odczytuje stan licznika oraz oblicza opóźnienie pomiędzy czasem wysłania komunikatu a jego odebraniem, następnie wypisuje na ekranie te informacje. Dostęp do współdzielonej zmiennej jest chroniony z zastosowaniem mutexa.

Warto zwrócić uwagę na strukturę aplikacji serwera. Jest ona wielowątkowa. Jeden wątek zajmuje się obsługą komunikacji sieciowej, drugi ewentualnym przetwarzaniem. Wątek sterowany funkcją `receive_data` ma w sobie wbudowaną nieskończoną pętlę odbierającą komunikaty z interfejsu sieciowego. Funkcja `recvfrom` i gniazdo posiadają standardowe ustawienia i w konsekwencji funkcja `recvfrom`, jeśli nie nadesłano do niej komunikatu, zawiesza wykonywanie tego wątku. Wątek ten przeważnie więc oczekuje na nadejście kolejnych datagramów z interfejsu sieciowego. W standardowych ustawieniach funkcja `recvfrom` pracuje w trybie blokującym, co oznacza blokowanie wykonywania wątku, w którym jest wywołana do chwili otrzymania kolejnego komunikatu. Można skonfigurować działanie tej funkcji i interfejsu gniazd do działania w trybie nieblokującym. Rezultat uruchomienia funkcji może być wówczas dwojaki. Jeśli w buforze komunikacyjnym jest komunikat, to funkcja go odbiera. Jeśli nie ma komunikatu, to zgłasza błąd i kończy wykonywanie. Jak już wspomniano, w kodzie serwera drugi (główny) wątek może prowadzić obliczenia, gdy wątek obsługujący komunikację sieciową oczekuje na kolejne komunikaty.

Badanie opóźnień w komunikacji

W skrypcie przyjęto, że do dyspozycji twórcy oprogramowania czasu rzeczywistego jest „standardowa” lokalna sieć komputerowa klasy Ethernet. Warto jednak pamiętać, że w typowych ustawieniach zadowalającą jakość transmisji uzyskuje się, gdy sieć tego typu jest obciążona do pewnego poziomu. Podczas przeciążenia sieci mogą się w niej pojawiać nieprzewidywalne opóźnienia, prowadzące do degradacji jakości działania niektórych aplikacji. Szczególnie wrażliwe na nieprzewidywalne opóźnienia są aplikacje wykonujące algorytmy sterowania, które w czasie rzeczywistym rozwiązują równania różniczkowe. Dla takich systemów zamiast typowych sieci Ethernet stosuje się ich specjalne warianty poprawiające lub zapewniające terminowość komunikacji. Istnieje również znacząca grupa aplikacji czasu rzeczywistego, które stosują „standardowe” sieci Ethernet. Ich twórcy przyjmują, że sieć może wprowadzić nieprzewidywalne opóźnienia w komunikacji, ale jest to traktowane jako stan awaryjny. W „normalnych” warunkach pracy sieć zapewnia wystarczającą przepustowość. W takich aplikacjach warto kontrolować opóźnienia w transmisji pomiędzy składnikami systemu. W tym celu, tak jak w aplikacji pokazanej na listingach 7. i 8., można przechwytywać czas systemowy tuż przed wysłaniem komunikatu i dołączyć tę wartość do przesyłanej struktury danych. Często taka zapamiętana wartość czasu jest nazywana pieczątką czasową. Odbiorca komunikatu może, posługując się wartością swojego lokalnego czasu systemowego, obliczyć wartość opóźnienia w transmisji. Warunkiem poprawnego obliczenia opóźnienia jest zsynchronizowanie czasu na wszystkich urządzeniach w sieci oraz gwarancja, że sam pomiar czasu (wywołanie funkcji `clock_gettime`) dostarcza wartość odczytanego czasu systemowego z nieprzekraczalnym opóźnieniem.

Technika uruchamiania aplikacji

Tym razem przykładowa aplikacja składa się z dwóch programów. Nie wymaga ona uprawnień administratora i pozwala na wymianę informacji z zastosowaniem lokalnego interfejsu sieciowego. Każdy ze skompilowanych programów można uruchomić niezależnie w osobnych terminalach. Nawiążą one połączenie przez gniazdo i zaczną wymieniać informacje.

Aplikacja sieciowa opierająca się na protokole TCP

Jak już wspomniano, komunikację sieciową pomiędzy aplikacjami można zestawić z zastosowaniem protokołu TCP. Nadaje się on szczególnie do niezawodnego przekazywania strumieni danych. W przypadku systemów czasu rzeczywistego można się nim posłużyć do transmisji nowych plików konfiguracyjnych. Na

listingu 9. pokazano przykładowy program klienta komunikującego się z zastosowaniem protokołu TCP ze wskazanym programem serwera. Celem programu klienta jest niezawodne przekazanie zawartości wybranego pliku.

Listing 9. Przykładowy program klienta w komunikacji z zastosowaniem protokołu TCP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define BUF_SIZE 1024
#define IP "127.0.0.1"
#define PORT 1234
void * send_file(void * arg);

int main()
{
    pthread_t snd_thread;
    void * thread_return;

    pthread_create(&snd_thread, NULL, send_file, NULL);
    pthread_join(snd_thread, &thread_return);

    return 0;
}

void * send_file(void * arg)
{
    int sock;
    char buf[BUF_SIZE];
    char *fstr = "konfiguracja1.txt";
    FILE *fp;
    int read_cnt;

    struct sockaddr_in serv_adr;

    sock=socket(PF_INET, SOCK_STREAM, 0);
    if(sock == -1)
    {
        perror("socket() error");
        exit(1);
    }

    memset(&serv_adr, 0, sizeof(serv_adr));
```

```
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=inet_addr(IP);
    serv_adr.sin_port=htons(PORT);

    if(connect(sock, (struct sockaddr*)&serv_adr,
       sizeof(serv_adr)) == -1)
    {
        perror("connect() error");
        exit(1);
    }

    fp=fopen(fstr, "rb");
    if(fp == NULL)
    {
        perror("fopen() error");
        exit(1);
    }

    while(1)
    {
        read_cnt=fread((void*)buf, 1, BUF_SIZE, fp);
        if(read_cnt<BUF_SIZE)
        {
            write(sock, buf, read_cnt);
            break;
        }
        write(sock, buf, BUF_SIZE);
    }

    shutdown(sock, SHUT_WR);
    read(sock, buf, BUF_SIZE);
    printf("Message from server: %s \n", buf);

    fclose(fp);
    close(sock);
    return NULL;
}
```

Program klienta został pomyślany w taki sposób, że przekazaniem danych do serwera zajmuje się osobny wątek. Jako cel komunikacji założono przekopiowanie do serwera zawartości pliku o nazwie „konfiguracja1.txt”. Podobnie jak w przypadku programu klienta z listingu 7., do komunikacji powołuje się nowe gniazdo z zastosowaniem funkcji `socket`. Podanie w tej funkcji parametru wywołania `SOCK_STREAM` oznacza chęć zbudowania połączenia z zastosowaniem protokołu TCP. Komunikacja pomiędzy serwerem a klientem będzie się odbywać przez kanał komunikacyjny identyfikowany przez wartość zapisaną w zmiennej `sock`. Wypełnienie struktury `serv_adr` pozwala na wskazanie adresu IP oraz

portu, na którym ma nasłuchiwać serwer na połączenie z klientem. Wywołanie funkcji `connect` uruchamia proces połączenia z serwerem. Jeśli udało się nawiązać połączenie, klient otwiera wybrany plik i przesyła go w porcjach po 1024 bajty za pomocą polecenia `write` przez gniazdo (identyfikowane przez zmienną `sock`). Ostatnim etapem komunikacji jest oczekiwanie na komunikat od serwera i wypisanie go na ekranie, a następnie pozamykanie otwartych plików i strumieni danych.

Listing 10. zawiera kod źródłowy przykładowego programu serwera przygotowanego na odbieranie kolejnych wersji pliku konfiguracyjnego od klientów z zastosowaniem protokołu TCP.

Listing 10. Przykładowy program serwera w komunikacji z zastosowaniem protokołu TCP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <pthread.h>

#define BUF_SIZE 1024
#define PORT 1234
void * recv_file(void * arg);

int main()
{
    pthread_t rcv_thread;
    void * thread_return;

    pthread_create(&rcv_thread, NULL, recv_file, NULL);
    pthread_join(rcv_thread, &thread_return);

    return 0;
}

void * recv_file(void * arg)
{
    int serv_sock;
    char buf[BUF_SIZE];
    char *fstr = "konfiguracja1_odebrana.txt";
    FILE *fp;
    int read_cnt;

    struct sockaddr_in serv_adr, clnt_adr;
    socklen_t clnt_adr_sz;
```

```
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);

    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(PORT);
    bind(serv_sock, (struct sockaddr*)&serv_adr,
         sizeof(serv_adr));
    listen(serv_sock, 5);

    while(1)
    {
        clnt_adr_sz=sizeof(clnt_adr);
        int clnt_sock=accept(serv_sock,
                             (struct sockaddr*)&clnt_adr,
                             &clnt_adr_sz);

        fp=fopen(fstr, "wb");
        while((read_cnt=read(clnt_sock, buf, BUF_SIZE))!=0)
            fwrite((void*)buf, 1, read_cnt, fp);

        write(clnt_sock, "Thank you", 10);
        fclose(fp);
        close(clnt_sock);
    }

    close(serv_sock);
    return NULL;
}
```

Serwer jest uruchomiony w osobnym wątku. W przypadku komunikacji z zastosowaniem protokołu TCP interfejs gniazd przekierowuje komunikację pomiędzy klientem a serwerem na nowe gniazdo. Okazuje się więc, że gniazdo (adres IP oraz port) jawnie opisane w programie klienta i serwera służy tylko do nawiązania połączenia. Sama komunikacja odbywa się poprzez inny port „wynegocjowany” podczas nawiązywania połączenia z danym klientem. Takie podejście umożliwia utworzenie oprogramowania serwera równocześnie prowadzącego wymianę informacji z wieloma klientami. Wystarczy tylko w momencie nawiązywania kolejnego połączenia przekazać sterowanie po stronie klienta do nowego wątku. Przykładowa aplikacja skonfigurowana jest jednak do innych celów. Serwer w czasie swojego działania oczekuje na kolejne połączenia z tym samym klientem. Za każdym razem tworzy on połączenie przez gniazdo i przenosi zawartość przesyłanego pliku do pliku lokalnego o nazwie „konfiguracja1_odebrana.txt”.

Zmienna serv_adr, podobnie jak w przypadku klienta, zawiera konfigurację gniazda, na którym nasłuchuje serwer. Funkcja bind łączy gniazdo nasłuchujące (serv_sock) ze zdefiniowanym adresem i portem. Funkcja listen przygotowuje kolejkę nadchodzących do gniazda połączeń. Maksymalna liczba połączeń w kolejce wynosi 5. Jeśli kolejka jest pełna i nadchodzą kolejne połączenia, to są odrzucane.

Funkcja accept zawiesza wykonywanie wątku w oczekiwaniu na połączenie. Pojawienie się połączenia do klienta powoduje odblokowanie wątku oraz utworzenie nowego gniazda do komunikacji pomiędzy klientem a serwerem (clnt_sock). W przykładowym programie serwer otwiera wskazany plik („konfiguracja1_odebrana.txt”) i przekopiowuje do niego strumień otrzymanych danych. Odczyt ze strumienia odbywa się z zastosowaniem funkcji read. Na zakończenie pobieranych danych serwer odsyła do klienta łańcuch tekstowy „Thank you”. Komunikacja przez interfejs gniazd jest więc dwukierunkowa. Zakończenie sesji pobierania danych kończy zamknięcie pliku oraz gniazd. Serwer wykonujący swoje operacje w nieskończonej pętli ponownie zawiesza swoje wykonywanie na funkcji accept lub od razu obsługuje kolejne połączenie oczekujące w kolejce.

Technika uruchamiania aplikacji

Przykładowa aplikacja składa się z dwóch programów. Nie wymaga ona uprawnień administratora i pozwala na wymianę informacji z zastosowaniem lokalnego interfejsu sieciowego. Każdy ze skompilowanych programów można uruchomić niezależnie w osobnych terminalach. Ponieważ w chwili uruchomienia klient oczekuje, że serwer będzie gotowy na połączenie, warto najpierw uruchomić program serwera, a potem klienta. W katalogu programu klienta należy utworzyć plik o nazwie „konfiguracja1.txt”. Rezultatem jednokrotnego uruchomienia klienta będzie odebranie zawartości pliku i umieszczenie jej w katalogu serwera w pliku o nazwie „konfiguracja1_odebrana.txt”.

Uwagi o interfejsie sieciowym jako medium komunikacji

We wcześniejszych rozdziałach skryptu pokazywano techniki komunikacji pomiędzy współbieżnymi wątkami, stosujące mechanizmy współdzielonych zmiennych i kolejek komunikatów. W odróżnieniu od wcześniejszych przykładów systemy pokazane na listingach 7., 8., 9. i 10. składają się z oddzielnych programów. Tym razem medium komunikacji jest interfejs sieciowy. Zaproponowane przykłady programów skonstruowano w taki sposób, aby można je było uruchomić na jednym komputerze. Mogą one zostać uruchomione w lokalnej sieci komputerowej na osobnych komputerach, jeśli w kodzie programu odpowiednio

skonfiguruje się adresy IP serwera i nadawcy. Technika stosowania interfejsu sieciowego do komunikacji między programami uruchomionymi na pojedynczym komputerze lub na wielu komputerach jest bardzo popularnym sposobem programowania współbieżnych i rozproszonych systemów informatycznych.

Literatura uzupełniająca

Standard programowania z zastosowaniem interfejsu gniazd jest bardzo dobrze opisany w wielu publikacjach. Wartościowe przewodniki dla programistów z tej dziedziny opierające się na przykładach kodów to [MS2007], [Fus2007], [Ami2021]. Czytelników zainteresowanych szczegółową znajomością zagadnień programowania sieciowego na poziomie gniazd mogą zainteresować książki [Ste2000] i [WS1998].

W rozdziale zaproponowano metodę komunikacji sieciowej, w której medium stanowi standardowa lokalna sieć komputerowa. Zaznaczono, że jeśli nie jest ona przeciążona, posiada odpowiednią przepustowość, oraz gdy wdroży się metody kontroli opóźnień w komunikacji, to może stanowić wystarczająco przewidywalne medium komunikacji w systemach czasu rzeczywistego. Okazuje się również, że właśnie takie podejście jest stosowane w wielu rzeczywistych rozwiązaniach technicznych.

Warto również wspomnieć, że znaczna liczba organizacji opracowała wersje sieci lokalnych opierające się na standardach Ethernet oraz rodzinie protokołów TCP/IP, oferujące zwiększoną lub pełną przewidywalność opóźnień w lokalnych sieciach komputerowych. W dziedzinie automatyki przemysłowej warto zwrócić uwagę na rozwiązanie EtherCAT (www.ethercat.org). Przemysł lotniczy pracuje nad wdrożeniem standardów ARINC 664P7 [ARINC2005] oraz Time-Triggered Ethernet (https://www.tttech.com/explore/time-triggered-protocol-ttp), z czego drugi ze standardów ma zastosowanie również w przemyśle samochodowym.

Zadania

1. Przygotuj modyfikacje przykładowych programów tak, aby klient i serwer były uruchomione na różnych komputerach.
2. Dlaczego w komunikacji w przypadku przesyłania zmiennych pomiarowych w systemach czasu rzeczywistego preferowany jest protokół UDP?
3. Na podstawie dostępnej literatury omów zasadę przyjmowania komunikacji przez serwer w protokole TCP.
4. Na podstawie publikacji zamieszczonych w Internecie przeanalizuj rozwiązania komunikacji sieciowej zaproponowane w standardzie ARINC 664P7. W jaki sposób poprawiają one przewidywalność opóźnień w komunikacji sieciowej?

5. Znajdź w Internecie urządzenia i metody synchronizacji czasowej komputerów w lokalnej sieci komputerowej z zastosowaniem sygnałów radiowych lub GPS.

Receptura 6.
– techniki wykrywania uszkodzeń

W poprzednich rozdziałach tego skryptu zakładano, że gwarantem poprawnego działania systemu czasu rzeczywistego jest jego odpowiednie zaprojektowanie. Oznaczało to analizę czasową wykonywania poszczególnych wątków, precyzyjne ustalenie priorytetów i algorytmu szeregowania, a także przydzielenie określonych zbiorów wątków do jednostek obliczeniowych. Może się jednak okazać, że mimo przestrzegania wcześniej omówionych zaleceń system będzie przekraczał swoje ograniczenia czasowe. Przyczyną pojawienia się takich problemów mogą być między innymi:

- nieprecyzyjne wyznaczenie najdłuższego czasu wykonywania pojedynczego cyklu wątku;
- niedoszacowane czasy zablokowań wątków przez inne wątki (czasy wykonywania się sekcji krytycznych dłuższe niż przewidziano);
- brak uwzględnienia wszystkich możliwych opóźnień w analizie szeregowalności (np. nieuwzględnienie czasów przełączania kontekstów i czasu na wykonywanie samego zadania szeregującego);
- problemy z programem szeregującym z powodu przeciążenia;
- praca systemu poza ustalonymi parametrami (zbyt często się pojawiające zadania wznawiane zewnętrznymi zdarzeniami).

W związku z tym warto rozważyć rozbudowę wcześniej pokazanych schematów tworzenia aplikacji czasu rzeczywistego o techniki umożliwiające swego rodzaju samoocenę stanu wykonywania systemu. Takie automatyczne wykrywanie nieprawidłowości może z kolei pozwalać na budowanie aplikacji czasu rzeczywistego pracujących w różnych trybach. Tryb „normalny” pracy aplikacji oznacza np. wykonywanie zaawansowanych algorytmów sterujących gwarantujących optymalizację zużycia energii i urządzeń wykonawczych. Z kolei tryb „awaryjny” lub „zdegradowany” może oznaczać wykonywanie przez system takich samych zadań, ale z zastosowaniem algorytmów uproszczonych, obliczanych szybciej i tym samym spełniających ograniczenia czasowe. W kolejnych podrozdziałach zostaną zaprezentowane wybrane techniki wykrywania uszkodzeń, które można wbudować w oprogramowanie, aby umożliwić mu przejście w inną konfigurację lub zgłoszenie błędu.

Wykrywanie przekroczenia ograniczeń czasowych

Podstawową techniką pozwalającą na kontrolę jakości systemu czasu rzeczywistego jest sprawdzanie, czy dany wątek spełnia założone ograniczenia czasowe. Program z listingu 11. pokazuje, jak stosując ponownie interfejs sygnałów, można ten problem rozwiązać. Jest on modyfikacją programu z listingu 4.

Listing 11. Zastosowanie interfejsu sygnałów do wykrywania przekroczenia ograniczeń czasowych

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define SIG SIGRTMIN
#define WATCHDOG_SIG SIGRTMIN+1

void watchdog_handler(int sig, siginfo_t *si, void *uc) {
    printf("Watchdog: przekroczono limit czasu!\n");
}

void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1, watchdog_timer;
    struct itimerspec its1, its_watchdog;
    struct sigevent sev1, sev_watchdog;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    sev_watchdog.sigev_notify = SIGEV_SIGNAL;
    sev_watchdog.sigev_signo = WATCHDOG_SIG;
    timer_create(CLOCK_REALTIME,
                 &sev_watchdog, &watchdog_timer);

    clock_gettime(CLOCK_REALTIME, &now);
```

```
    its1.it_value.tv_sec = now.tv_sec + 2;
                                  // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 3; // 3-sekundowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("Watek 1: obliczenia wznowione\n");

        // Dodajemy tutaj watchdog timer
        clock_gettime(CLOCK_REALTIME, &now);
        its_watchdog.it_value.tv_sec = now.tv_sec + 3;
                                  // 3 sekundy opóźnienia
        its_watchdog.it_value.tv_nsec = now.tv_nsec;
        its_watchdog.it_interval.tv_sec = 0;
                                  // bez interwału
        its_watchdog.it_interval.tv_nsec = 0;

        timer_settime(watchdog_timer, TIMER_ABSTIME,
                      &its_watchdog, NULL);
        // ---------------------------------------------------
        //Czas na wykonywanie obliczeń w pojedynczym cyklu:
        //----------------------------------------------------
        sleep(2);
        its_watchdog.it_value.tv_sec = 0;
        its_watchdog.it_value.tv_nsec = 0;
        its_watchdog.it_interval.tv_sec = 0;
        its_watchdog.it_interval.tv_nsec = 0;
        timer_settime(watchdog_timer, TIMER_ABSTIME,
                      &its_watchdog, NULL);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1;
    pthread_attr_t attr1;
    struct sched_param param1, param_main;
    cpu_set_t cpuset;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);
    pthread_sigmask(SIG_BLOCK, &set, NULL);
```

```
    struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = watchdog_handler;
    sigaction(WATCHDOG_SIG, &sa, NULL);

    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset);
                     // Wybierz procesor nr 2 w systemie
    printf("Liczba dostępnych CPU: ");
    printf(" %d\n", (int) sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t), &cpuset);

    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
                                 PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                                sizeof(cpu_set_t),
                                &cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_join(t1, NULL);

    pthread_attr_destroy(&attr1);

    return 0;
}
```

W funkcji `main` wprowadzono zmodyfikowaną obsługę nowego sygnału identyfikowanego jako `WATCHDOG_SIG`. Jest to jeden z sygnałów czasu rzeczywistego. Nowe podejście od obsługi sygnału jest zawarte we fragmencie kodu:

```
struct sigaction sa;
    sa.sa_flags = SA_SIGINFO;
    sa.sa_sigaction = watchdog_handler;
    sigaction(WATCHDOG_SIG, &sa, NULL);
```

W programie powołana jest zmienna typu struct sigaction. Pola tej struktury zawierają informację, w jaki sposób będzie obsługiwany nowy sygnał. Ustalenie pola sa_flags na wartość SA_SIGINFO oznacza, że w polu sa_sigaction ma się znaleźć wskaźnik do funkcji, która będzie wywoływana automatycznie, po nadejściu wskazanego sygnału do programu. Rejestracja obsługi sygnału odbywa się przez wywołanie funkcji sigaction. Wywołanie funkcji sigaction w omawianym fragmencie kodu oznacza zarejestrowanie pewnej funkcji o nazwie watchdog_handler jako tej, która ma być uruchamiana za każdym razem, gdy do aplikacji zostanie przesłany sygnał WATCHDOG_SIG. Kod funkcji watchdog_handler znajduje się w początkowej sekcji programu. Funkcja komunikuje system, że został przekroczony limit czasu.

W programie powoływany jest jeden wątek potomny czasu rzeczywistego (thread1). W wątku, w sekcji poprzedzającej cykliczne wykonywanie, jest tworzony nowy, dodatkowy budzik (watchdog_timer), który będzie generował sygnał WATCHDOG_SIG, ale jego uruchomienie jest wstrzymane. Podczas każdego wznowienia cyklu wykonywania wątku następuje pobranie bieżącego czasu systemowego, zdefiniowanie przedziału czasowego, a następnie uruchomienie budzika watchdog_timer (wywołanie funkcji timer_settime). Przedział czasowy zdefiniowany dla tego budzika powinien odzwierciedlać czas, przed którego upływem powinien się zakończyć pojedynczy cykl wykonywania obliczeń przez wątek. Jeśli wątek przekroczy zadany czas wykonywania swojego cyklu, w systemie zostanie zgłoszony sygnał i uruchomiona będzie funkcja watchdog_handler. W przeciwnym wypadku budzik zostaje zdezaktywowany przez ponowne uruchomienie funkcji timer_settime z wyzerowanymi wartościami czasu do odliczania.

Przyjęcie nazewnictwa nowego sygnału (WATCHDOG_SIG), budzika (watchdog_timer) oraz nowej funkcji obsługującej sygnał (watchdog_handler) nie jest przypadkowe. W programie zastosowano bowiem schemat programistyczny często nazywany właśnie „watchdog”: powołano układ czasowo licznikowy i nakazano mu zliczanie czasu. Jeśli system przed upływem tego czasu nie zdezaktywuje układu, to oznacza, że z jakichś przyczyn „nie zdążył” tego zrobić, czyli nastąpił w nim błąd.

Technika uruchamiania aplikacji

Do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Przykładowe polecenie uruchamiające program może wyglądać następująco: *„sudo ./program”*. System operacyjny może wymusić wtedy podanie hasła administratora. Uruchomienie aplikacji z założonymi parametrami czasowymi będzie symulowało poprawne zachowanie aplikacji. Warto

rozważyć np. wydłużenie obliczeń przez wątek i sprawdzenie, czy zostanie wykryte przekroczenie ograniczeń czasowych.

Przeterminowanie dostępu do sekcji krytycznej

Przekroczenie ograniczeń czasowych przez wątek czasu rzeczywistego może być spowodowane przedłużonym blokowaniem w dostępie do współdzielonego zasobu. W klasycznym rozwiązaniu pokazanym na listingu 5. założono, że dostęp do współdzielonego zasobu musi być wykonany z zastosowaniem mechanizmu wzajemnego wykluczania, oraz że czas ewentualnego zablokowania zadania w oczekiwaniu na zasób jest przewidywalny i uwzględniony w planowaniu szeregowania zadań. Okazuje się, że „przedłużające się" czasy zablokowań na współdzielonych zasobach można wykrywać z zastosowaniem zmodyfikowanych wersji funkcji obsługujących mutexy. Na listingu 12. zaproponowano zmodyfikowaną wersję programu z listingu 5.

Listing 12. Rozwiązanie komunikacji przez zmienną dzieloną z wykryciem przeterminowania

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)

pthread_mutex_t mutex;
pthread_mutexattr_t mutex_attr;
int shared_var = 0;

void *thread1(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer1;
    struct itimerspec its1;
    struct sigevent sev1;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
```

```
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);
    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                                 // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 3; // 3-sekudowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
        sigwait(&set, &sig);
        pthread_mutex_lock(&mutex);
        printf("Watek 1: obliczenia wznowione, ");
        printf("shared_var = %d\n", shared_var);
        sleep(2);
                // Wydłużenie sekcji krytycznej do 2 sekund
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

void *thread2(void *arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;
    struct timespec timeout;
    int ret;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
    timer_create(CLOCK_REALTIME, &sev2, &timer2);

    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                                  // 2 sekundy opóźnienia
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;

    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);
```

```
    while (1) {
        sigwait(&set, &sig);
        clock_gettime(CLOCK_REALTIME, &timeout);
        timeout.tv_sec += 1;
                    // Czas oczekiwania mutexa to 1 sekunda
        ret = pthread_mutex_timedlock(&mutex, &timeout);
        if (ret == ETIMEDOUT) {
         printf("Watek 2: Nie uzyskano dostępu do ");
         printf("zmiennej współdzielonej ");
         printf("w ciągu jednej sekundy\n");
            continue;
        }
        printf("Watek 2: obliczenia wznowione, ");
        printf("shared_var = %d\n", ++shared_var);
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_attr_t attr1, attr2;
    struct sched_param param1, param2, param_main;
    cpu_set_t cpuset;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);
    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);

    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset);
                        // Wybierz procesor nr 2 w systemie
    printf("Liczba dostepnych CPU: ");
    printf("%d\n", (int) sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t), &cpuset);

    pthread_mutexattr_init(&mutex_attr);
    pthread_mutexattr_setprotocol(&mutex_attr,
                                  PTHREAD_PRIO_INHERIT);
    pthread_mutex_init(&mutex, &mutex_attr);
```

```
    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
                              PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                  sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                             sizeof(cpu_set_t),
                             &cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_attr_init(&attr2);
    pthread_attr_setscope(&attr2, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr2, PTHREAD_EX-
PLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
    param2.sched_priority =
                   sched_get_priority_max(SCHED_FIFO) - 2;
    pthread_attr_setschedparam(&attr2, &param2);
    pthread_create(&t2, &attr2, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);
    pthread_mutex_destroy(&mutex);

    return 0;
}
```

W zmodyfikowanym programie w wątku `thread1` sztucznie wydłużono czas przebywania wątku w sekcji krytycznej. Wątek po przejęciu mutexa „zasypia” na 2 s, a następnie go zwalnia. W zmodyfikowanej cyklicznej części wątku `thread2` próba przejęcia mutexa odbywa się z zastosowaniem funkcji `pthread_mutex_timedlock`. Funkcja ta przyjmuje dwa parametry: uchwyt do mutexa (`mutex`) oraz strukturę typu `struct timespec` (`timeout`) do przechowywania wartości czasowych. Można więc przekazać do niej wartość maksymalnego opóźnienia, z jakim mutex będzie oczekiwał na odblokowanie przez inny wątek wykonujący swoją sekcję krytyczną. W przykładowym programie założono, że mutex będzie oczekiwał do 1 s.

Wynik działania funkcji `pthread_mutex_timedlock` może przyjąć dwa podstawowe rezultaty. Jeśli w ciągu zadanego przedziału czasowego inny ele-

ment systemu zwolni mutex, to funkcja dokonuje przejęcia tego mutexa i rozpoczyna się wykonywanie sekcji krytycznej. Jeśli natomiast do upłynięcia zadanego odcinka czasu mutex w dalszym ciągu będzie przejęty przez inny element systemu, to funkcja kończy swoje działanie i zwraca wartość błędu `ETIMEDOUT`. Istnieje wtedy możliwość zgłoszenia błędu działania systemu.

Technika uruchamiania aplikacji

Do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Przykładowe polecenie uruchamiające program może wyglądać następująco: *„sudo ./program”*. System operacyjny może wymusić wtedy podanie hasła administratora. W przykładowym programie co pewien czas dostęp do współdzielonego zasobu jest blokowany na 2 s. Powoduje to wyświetlenie na konsoli odpowiedniego komunikatu.

Wykrywanie blokujących się sekcji krytycznych

Przykładowy program z listingu 12. poprawnie wykrywa przeterminowanie w dostępie do współdzielonego zasobu i jest prawidłowym rozwiązaniem postawionego problemu, o ile wątek może „pozwolić sobie” na bezczynne oczekiwanie. W dostępie do blokowanego mutexa można jednak przyjąć strategię „odpytywania”, czy mutex jest już zwolniony. Na listingu 13. zaproponowano alternatywną modyfikację programu z listingu 5.

Listing 13. Rozwiązanie komunikacji przez zmienną dzieloną z „odpytywaniem”

```
#define _GNU_SOURCE
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define SIG SIGRTMIN
#define SIG1 (SIGRTMIN+1)

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mutex_attr;
int shared_var = 0;

void* thread1(void* arg) {
    sigset_t set;
    int sig;
```

```
    timer_t timer1;
    struct itimerspec its1;
    struct sigevent sev1;
    struct timespec now;

    sigemptyset(&set);
    sigaddset(&set, SIG);

    sev1.sigev_notify = SIGEV_SIGNAL;
    sev1.sigev_signo = SIG;
    timer_create(CLOCK_REALTIME, &sev1, &timer1);

    clock_gettime(CLOCK_REALTIME, &now);
    its1.it_value.tv_sec = now.tv_sec + 2;
                              // 2 sekundy opóźnienia
    its1.it_value.tv_nsec = now.tv_nsec;
    its1.it_interval.tv_sec = 3; // 3-sekundowy interwał
    its1.it_interval.tv_nsec = 0;

    timer_settime(timer1, TIMER_ABSTIME, &its1, NULL);

    while (1) {
        sigwait(&set, &sig);
        pthread_mutex_lock(&mutex);
        printf("Watek 1: obliczenia wznowione, ");
        printf("shared_var = %d\n", shared_var);
        sleep(2);
        //Wydłużenie sekcji krytycznej do 2 sekund
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

void* thread2(void* arg) {
    sigset_t set;
    int sig;
    timer_t timer2;
    struct itimerspec its2;
    struct sigevent sev2;
    struct timespec now;
    int ret;
    int try_count = 0;

    sigemptyset(&set);
    sigaddset(&set, SIG1);

    sev2.sigev_notify = SIGEV_SIGNAL;
    sev2.sigev_signo = SIG1;
```

```
    timer_create(CLOCK_REALTIME, &sev2, &timer2);

    clock_gettime(CLOCK_REALTIME, &now);
    its2.it_value.tv_sec = now.tv_sec + 2;
                              // 2 sekundy opoznienia
    its2.it_value.tv_nsec = now.tv_nsec;
    its2.it_interval.tv_sec = 5; // 5-sekundowy interwał
    its2.it_interval.tv_nsec = 0;

    timer_settime(timer2, TIMER_ABSTIME, &its2, NULL);

    while (1) {
        sigwait(&set, &sig);
        while ((ret = pthread_mutex_trylock(&mutex)) != 0
               && try_count < 3)
        {
            usleep(500000);
            // Czekaj 0,5 sekundy przed kolejną próbą
            try_count++;
        }
        if (ret != 0) {
            printf("Watek 2: Problem z dostępem do ");
            printf("zmiennej współdzielonej ");
            printf("po trzech próbach\n");
            try_count = 0;
            continue;
        }
        printf("Watek 2: obliczenia wznowione, ");
        printf("shared_var = %d\n", ++shared_var);
        pthread_mutex_unlock(&mutex);
        try_count = 0;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    pthread_attr_t attr1, attr2;
    struct sched_param param1, param2, param_main;
    cpu_set_t cpuset;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIG);

    sigaddset(&set, SIG1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);
```

```
    CPU_ZERO(&cpuset);
    CPU_SET(1, &cpuset); // Wybierz procesor nr 2 w syste-
mie
    printf("Liczba dostępnych CPU: ");
    printf(" %d\n", (int)sysconf(_SC_NPROCESSORS_ONLN));

    param_main.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 3;
    pthread_setschedparam(pthread_self(),
                          SCHED_FIFO, &param_main);
    pthread_setaffinity_np(pthread_self(),
                           sizeof(cpu_set_t), &cpuset);

    pthread_mutexattr_init(&mutex_attr);
    pthread_mutexattr_setprotocol(&mutex_attr,
                                  PTHREAD_PRIO_INHERIT);
    pthread_mutex_init(&mutex, &mutex_attr);

    pthread_attr_init(&attr1);
    pthread_attr_setscope(&attr1, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr1,
                                 PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr1, SCHED_FIFO);
    param1.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 1;
    pthread_attr_setschedparam(&attr1, &param1);
    pthread_attr_setaffinity_np(&attr1,
                                sizeof(cpu_set_t),
                                cpuset);
    pthread_create(&t1, &attr1, thread1, NULL);

    pthread_attr_init(&attr2);
    pthread_attr_setscope(&attr2, PTHREAD_SCOPE_SYSTEM);
    pthread_attr_setinheritsched(&attr2,
                                 PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
    param2.sched_priority =
                    sched_get_priority_max(SCHED_FIFO) - 2;
    pthread_attr_setschedparam(&attr2, &param2);
    pthread_create(&t2, &attr2, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_attr_destroy(&attr1);
    pthread_attr_destroy(&attr2);
```

```
pthread_mutex_destroy(&mutex);

return 0;}
```

Podobnie jak w programie z listingu 12., w wątku `thread1` sztucznie wydłużono czas przebywania wątku w sekcji krytycznej. Wątek po przejęciu mutexa „zasypia” na 2 s, a następnie go zwalnia. W zmodyfikowanej cyklicznej części wątku `thread2` próba przejęcia mutexa odbywa się z zastosowaniem funkcji `pthread_mutex_trylock`.

Wynik działania funkcji `pthread_mutex_trylock` może przyjąć dwa podstawowe rezultaty. Jeśli w chwili wywołania funkcji mutex jest wolny, to funkcja dokonuje przejęcia tego mutexa i rozpoczyna się wykonywanie sekcji krytycznej. W przeciwnym wypadku funkcja zwraca kod błędu. Istnieje wtedy możliwość zgłoszenia błędu działania systemu.

Technika uruchamiania aplikacji

Do uruchomienia obecnie omawianej aplikacji wymagane są uprawnienia konta *root* systemu Linux. Przykładowe polecenie uruchamiające program może wyglądać następująco: *„sudo ./program*”. System operacyjny może wymusić wtedy podanie hasła administratora. W przykładowym programie co pewien czas dostęp do współdzielonego zasobu jest blokowany przez jeden z wątków na 2 s. Drugi wątek co pewien czas próbuje z zastosowaniem funkcji `pthread_mutex_trylock` otrzymać dostęp do mutexa. Trzy nieudane próby dostępu do mutexa powodują zgłoszenie tego faktu na konsoli.

Literatura uzupełniająca

Techniki wykrywania uszkodzeń w systemach czasu rzeczywistego są dobrze skomentowane w pracy [BW2009].

Zadania

1. Na czym polega zasada działania układu czasowego typu watchdog?
2. Omów różnicę w wykonywaniu funkcji `pthread_mutex_lock` oraz `pthread_mutex_timedlock`.
3. Omów różnicę w wykonywaniu funkcji `pthread_mutex_lock` oraz `pthread_mutex_trylock`.
4. Zaproponuj modyfikacje kodów źródłowych aplikacji omawianych we wcześniejszych rozdziałach, tak aby można było w nich wykrywać przekroczenia ograniczeń czasowych oraz wydłużające się oczekiwania na dostąp do współdzielonych zasobów.

5. Zaproponuj projekt rozproszonej aplikacji czasu rzeczywistego, w której bada się przekroczenie ograniczeń czasowych oraz monitoruje opóźnienia w komunikacji sieciowej.

Podsumowanie

Celem przedłożonego Czytelnikowi skryptu było stopniowe wprowadzenie do zagadnień programowania systemów czasu rzeczywistego dla systemu Linux. Przyjęto tu założenie, że platformą do uruchamiania aplikacji będzie system operacyjny czasu rzeczywistego, a więc taka modyfikacja Linuxa, która gwarantuje czasy odpowiedzi systemu na zdarzenia niezależnie od jego obciążenia. Kolejne rozdziały skryptu wprowadzały techniki programistyczne oraz wiedzę teoretyczną niezbędne do zbudowania współbieżnych wielowątkowych aplikacji, gdzie wątki wznawiają swoje obliczenia co pewien przedział czasowy oraz komunikują się ze sobą z zastosowaniem lokalnych i sieciowych mechanizmów wymiany informacji. Zwrócono uwagę na niezbędne techniki projektowania tego typu systemów, zapewniające terminowość wykonywania obliczeń oraz automatyczne wykrywanie przekroczenia ograniczeń czasowych.

Treści omówione w skrypcie są, w opinii autora, ważnym, ale niekompletnym kompendium wiedzy dotyczącym wytwarzania systemów czasu rzeczywistego. W rozważaniach pominięto między innymi:

- projektowanie obsługi tak zwanych zdarzeń sporadycznych (takich, które mogą się pojawić raz na jakiś długi, z punktu widzenia innych wątków, czas);
- szeregowanie zadań na systemach wielordzeniowych, gdzie problem przydziału wątków do rdzeni odbywa się globalnie;
- zaawansowane techniki komunikacji w systemach rozproszonych z zastosowaniem specjalizowanych sieci komputerowych czasu rzeczywistego;
- spojrzenie na systemy czasu rzeczywistego w ogólniejszym ujęciu, gdzie do osadzania aplikacji czasu rzeczywistego stosuje się inne niż systemy operacyjne platformy sprzętowo-programowe oraz inne niż C języki programowania.

Czytelnicy zainteresowani tymi zagadnieniami mogą rozszerzyć swoją wiedzę, studiując między innymi prace: [BBB2015], [But2011], [KS2022], [ARINC2015], [AS6802_2023], [BW2009].

Przedłożony kod źródłowy jest w 99% zgodny ze standardem POSIX dla systemów rodziny UNIX [BFN1996], [But1997], [BW2009], [Gal1995]. Wydaje się więc, że z niewielkimi modyfikacjami (np. dostosowanie odwołań do plików nagłówkowych, pominięcie funkcji nieprzenaszalnych do innych niż Linux systemów operacyjnych, na które zwrócono uwagę w tekście skryptu) można zapropono-

wane aplikacje wdrażać do uruchomienia dla innych niż Linux systemów operacyjnych czasu rzeczywistego, np. WxWorks [VxW2011] czy QNX [KH2003], [Uła2007].

„Pilnowanie się" standardu POSIX podczas tworzenia oprogramowania na potrzeby tego skryptu czasami oznaczało również rezygnację z tworzenia oprogramowania z zastosowaniem najnowszych osiągnięć z dziedziny projektowania systemów czasu rzeczywistego. Jest ono więc przeznaczone na systemy operacyjne z dłuższą historią, postrzegane jako dobrze ugruntowane na rynku informatycznym (np. Linux, QNX, VxWorks). Warto jednak śledzić współczesny rynek systemów mikroprocesorowych oraz systemów operacyjnych czasu rzeczywistego (np. Azure RTOS (ThreadX), LynxOS, embOS, Real-Time Ubuntu), w których najprawdopodobniej wdrożone zostaną nowe osiągnięcia z dziedziny inżynierii oprogramowania systemów czasu rzeczywistego.

Bibliografia

[Ami2021] K. Amini. Ekstremalny kod w języku C. Współbieżność i programowanie zorientowane obiektowo. Helion, 2021.

[ARINC2015] AIRCRAFT DATA NETWORK PART 7 AVIONICS FULL DUPLEX SWITCHED ETHERNET (AFDX) NETWORK, ARINC 2015.

[AS6802_2023] AS6802 Aerospace Standard, Time-Triggered Ethernet, www.sae.org, 2023.

[BBB2015] S. Baruah, M. Bertogna, G. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer Cham, 2015.

[BFN1996] D. Buttlar, J. Farrell, B. Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly Media, 1996.

[But1997] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[But2011] G.C. Buttazzo. *Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications*. Springer Science+Business Media, LLC 2011.

[BW2009] A. Burns, A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX* (4th Edition). Pearson Education Canada, 2009.

[CDKM2002] F. Cottet, J. Delacroix, C. Kaiser, Z.Z. Mammeri. *Scheduling in Real-Time Systems*. Wiley, 2002.

[Coo2019] J. Cooling. The Complete Edition – Software Engineering for Real-Time Systems: A software engineering perspective toward designing real-time systems. Packt Publishing, 2019.

[Dou1999] B.P. Douglass. *Doing Hard Time: Developing Real-Time Systems with UML*, Objects, Frameworks, and Patterns. Addison-Wesley Professional; 1st edition, 1999.

[DSE2001a] Dedicated Systems Experts (http://www.dedicated-systems.com/), *RTOS Evaluation Project, EVALUATION REPORT DEFINITION*, 2001.

[DSE2001b] Dedicated Systems Experts (http://www.dedicated-systems.com/), *RTOS Evaluation Project, WHAT MAKES A GOOD RTOS*, 2001.

[Erc2019] K. Erciyes. Distributed Real-Time Systems, Theory and Practice. Springer Cham, 2019.

[Fus2007] J. Fusco. Linux Programmer's Toolbox. Financial Times Prentice Hall, 2007.

[Gal1995] B.O. Gallmeister. POSIX. 4: Programming for the Real World. O'Reilly & Associates, Inc., 1995.

[GCL2016] A. Gupta, A.K. Chandra, P. Luksch. *Real-Time and Distributed Real-Time Systems: Theory and Applications*. CRC Press, 2016.

[Gom2000] H. Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley Professional, 2000.

[HS1991] W.A. Halang, A.D. Stoyenko. *Constructing Predictable Real Time Systems. Springer*, 1991.

[Ker2024] M. Kerrisk. *https://Man7.org*. Dostęp: marzec 2024.

[KH2003] R. Krten, C. Herborth. The QNX Cookbook: Recipes for Programmers, Parse, 2003.

[Kle1993] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, M.G. Harbour. *A Practitioner's Handbook for Real-Time Analysis, Guide to Rate Monotonic Analysis for Real-Time Systems*, Kluwer Academic Publishers, 1993.

[KLR2012] K. Kuźniar, K. Lal, T. Rak. *Programowanie w Linuksie, Ćwiczenia*. Helion, 2012.

[KS2022] H. Kopetz, W. Steiner. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Springer Cham, 2022.

[LRO2003] K. Lal, T. Rak, K. Orkisz. *RTLinux system czasu rzeczywistego*. Helion 2003.

[Mat2000] N. Matthew i inni. *Professional Linux Programming*. Apress, 2000.

[MS2007] N. Matthew, R. Stones. *Beginning Linux Programming, 4th Edition*. Wiley, 2007.

[RCh2009] J. Ras, A.M.K. Cheng. *An Evaluation of the Dynamic and Static Multiprocessor Priority Ceiling Protocol and the Multiprocessor Stack Resource Policy in an SMP System*. 15th IEEE Real-Time and Embedded Technology and Applications Symposium, San Francisco, CA, USA, 2009.

[SSRB1998] J.A. Stankovic, M. Spuri, K. Ramamritham, G.C. Buttazzo. *Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms*. Springer Science+Business Media New York, 1998.

[Sta2018] W. Stallings. *Systemy operacyjne. Architektura, funkcjonowanie i projektowanie*. Helion, 2018.

[Ste2000] W.R. Stevens. *UNIX programowanie usług sieciowych, tom 1 API: gniazda i XTI.* WNT, 2000.

[Szm2000] T. Szmuc. *Specyfikacja i projektowanie oprogramowania systemów czasu rzeczywistego*. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków 2000.

[Szm2001] T. Szmuc. Modelowanie i metody inżynierii oprogramowania systemów czasu rzeczywistego. Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków 2001.

[Uła2007] J. Ułasiewicz. *Systemy Czasu Rzeczywistego QNX6 Neutrino*. Wydawnictwo TBC, 2007.

[VxW2011] VxWorks. *APPLICATION PROGRAMMER'S GUIDE 6.9*, Wind River, 2011.

[WS1998] G.R. Wright, W.R. Stevens. *Biblia TCP/IP, tom II. Implementacje.* Wydawnictwo RM, 1998.

Dodatki

Kody źródłowe aplikacji omawianych w skrypcie są dostępne na stronie: http://ssamolej.kia.prz.edu.pl/Skrypt_Linux_RTS/.