POLITECHNIKA GDAŃSKA
WYDZIAŁ ELEKTROTECHNIKI I AUTOMATYKI

Imię i nazwisko studenta: Daniil Klimuk
Nr albumu: 187740
Poziom kształcenia: Studia pierwszego stopnia
Forma studiów: stacjonarne
Kierunek studiów: Automatyka, robotyka i systemy sterowania
Specjalność: Automatyka i Systemy Sterowania

PRACA DYPLOMOWA INŻYNIERSKA

Tytuł pracy w języku polskim: Procedury automatyzacji procesów w systemie mikroprocesorowym

Tytuł pracy w języku angielskim: Methods for implementing control algorithms using embedded systems

Opiekun pracy: dr inż. Robert Smyk

W momencie napisania pracy systemy cyfrowe i oprogramowanie zarządzające tymi systemami są bardzo rozbudowane, posiadające od kilku do kilkunastu poziomów abstrakcji i podsystemów. Z powodu znacznego obniżenia ceny technologii półprzewodnikowej i dużego skoku technologicznego, problemy małej pamięci i niskiej wydajności systemów cyfrowych są już przeszłością. Aktualnie dużo zasobów jest poświęcane na dodawanie kolejnego poziomu do już istniejących systemów, a rzadko jest zwracana uwaga na to, co leży pod spodem, chociaż tam znajduje się logika nie mniej interesująca i definiująca właściwości całego systemu.

Centrum uwagi tej pracy stanowi logika zarządzająca zasobami obliczeniowymi systemu cyfrowego, nazywana ogólnie systemem operacyjnym, a w szczególności oprogramowanie planujące wykonywanie zadań i zarządzające zasobami jednostki obliczeniowej systemu — procesora.

Praca rozpoczyna się od podstawowych pojęć i architektur systemowych, a następnie stopniowo pogłębia omawiane zagadnienia, aż do programów zarządzających zasobami jednostki obliczeniowej. Następnie omówione wcześniej programy zostaną przeanalizowane. W ostatniej części pracy kilka algorytmów zarządzania jednostką obliczeniową zostaną zaimplementowane w realnym systemie operacyjnym w celu zbadania ich cech. Na końcu autor przedstawi własne przemyślenia i wnioski, bazując na poprzedniej pracy teoretycznej i praktycznej.

Słowa kluczowe: system operacyjny, hiperwizor, system cyfrowy, zarządzanie zasobami procesora, planowanie zadań, FreeRTOS.

Dziedzina nauki i techniki, zgodnie z wymaganiami OECD: 2.2 Elektrotechnika, elektronika, inżynieria informatyczna; 2.2.b Robotyka i automatyka

At the time of writing, the digital systems and software running on these systems are very complex, having from several to dozen layers of abstractions and subsystems. Due to the significant reduction in the price of semiconductor technology and a large leap in progress, the problems of small memory and low efficiency of digital systems are a thing of the past. Currently, a lot of resources are spent on adding another abstraction level to existing systems, paying rather low attention to what is hidden underneath, although there is logic there that is no less interesting and defines the properties of the entire system.

The focus of this work is the logic managing the computational resources of the digital system, generally called the operating system, and, in particular, the software scheduling the execution of tasks and managing the resources of the system's computing unit - the processor.

The work will start with basic concepts and system architectures, and will delve deeper and deeper until it reaches the original goal. Then the programs mentioned before will be analyzed. In the last part of the work, several algorithms for managing computational unit resources will be implemented in a real operating system in order to investigate their characteristics. At the end, the author will present his own thoughts and conclusions based on the previous theoretical and practical work.

Keywords: operating system, hypervisor, digital system, scheduling, FreeRTOS.

Spis treści

Spis rysunków

1 Wstęp

1.1 Wprowadzenie

Począwszy od pojawienia się pierwszych urządzeń elektronicznych i maszyn obliczeniowych minęło sporo czasu, w trakcie którego postęp technologiczny napędzał cele stawiane przed systemami cyfrowymi, co powodowało ich szybką ewolucję. Od list instrukcji zapisanych na kartach perforowanych, przez skomplikowane programy zajmujące duże obszary pamięci, dodanie do sporej mocy obliczeniowej wielu urządzeń peryferyjnych i podejmowanie prób abstrahowania się od ich implementacji, wprowadzenie skomplikowanych abstrakcji systemowych w celu ułatwienia dalszego rozwoju i zwiększenia bezpieczeństwa, aż do łączenia licznych systemów w jeden, wielofunkcyjny system.

Aktualnie logika zarządzająca systemami cyfrowymi i sprawiająca, że system realizuje postawione przed nim cele oraz spełnia określone oczekiwania, jest bardzo skomplikowana. Charakteryzuje się ona wieloma poziomami abstrakcji, co znacząco utrudnia integrację i analizę.

Z kolei bardzo ważną dla automatyki częścią tego postępu jest ewolucja systemów czasu rzeczywistego. Wciąż można spotkać systemy cyfrowe pełniące rolę systemu czasu rzeczywistego, w których oprogramowanie działa bezpośrednio na sprzęcie (tj. bez żadnych warstw abstrakcji pomiędzy sprzętem a aplikacją, np. bez systemu operacyjnego). Jednak coraz częściej wykorzystywane są RTOS (ang. Real Time Operating System; System Operacyjny Czasu Rzeczywistego [18]) lub inne, bardziej złożone oprogramowanie, które wykorzystuje wzrost funkcjonalności i wydajności mikrokontrolerów, na przykład zaawansowaną wirtualizację.

Wraz z coraz większą skomplikowanością systemów wbudowanych i ich oprogramowania wzbudzają zainteresowanie też sposoby i metody na realizację wymagań systemu czasu rzeczywistego przez taki system, jak również i sprawdzenie tych wymagań. Akurat w zakresie spełnienia jakichkolwiek wymagań najlepszym obiektem badania będą tzw. RTOS'y, które nie tylko są zbudowane z myślą o uruchomieniu na platformach wbudowanych z mocno ograniczonymi zasobami, ale i, przede wszystkim, muszą dbać o wymogi czasowe postawione im przez aplikacje.

Ze wzrostem złożoności systemów wbudowanych i ich oprogramowania rośnie zainteresowanie sposobami oraz metodami realizacji wymagań systemu czasu rzeczywistego przez takie systemy, a także ich weryfikacją. W zakresie spełniania takich wymagań najlepszym obiektem badań są tzw. RTOS'y, które nie tylko są projektowane z myślą o uruchamianiu na platformach wbudowanych z mocno ograniczonymi zasobami, ale przede wszystkim muszą spełniać wymogi czasu rzeczywistego narzucone przez aplikacje.

1.2 Cel i zakres pracy

Celem pracy jest analiza oprogramowania zarządzającego zasobami jednostki (lub jednostek) obliczeniowej w cyfrowych systemach wbudowanych oraz planującego wykonywanie zadań, w

tym w systemach cyfrowych pełniących rolę systemów czasu rzeczywistego na sprzęcie opartym na architekturze ARM (ang. Advanced RISC Machine, lub, poprzednio, Acorn RISC Machine; brytyjskie przedsiębiorstwo zajmujące się projektowaniem architektur mikroprocesorów, z siedzibą w Cambridge, Anglia [2]) oraz weryfikacja spełnienia determinizmu czasowego przez te systemy.

Zakres pracy:

- Przedstawienie i omówienie głównych cech architektur oprogramowania uruchomianego na systemach wbudowanych skonstruowanych na podstawie systemów cyfrowych opartych na architekturze ARM'owej;
- Przedstawienie przykładów realizacji omówionych architektur oprogramowania systemowego;
- Dekompozycja wybranych architektur oprogramowania;
- Wyodrębnienie części oprogramowania odpowiadających za zarządzanie zasobami jednostki obliczeniowej i planujących wykonywanie zadań dla wybranych architektur;
- Analiza wyodrębnionych części oprogramowania;
- Przedstawienie przykładów realizacji wyodrębnionych części oprogramowania;
- Sprawdzenie sposobów realizacji i weryfikacji wymagań czasu rzeczywistego;
- Wybranie realizacji oprogramowania systemowego dla każdej z przedstawionych architektur i przeprowadzenie badań praktycznych w zakresie determinizmu czasowego, w tym, modyfikacja wybranych realizacji z celem sprawdzenia realizacji warunku determinizmu czasowego.

2 Architektury i przykłady realizacji

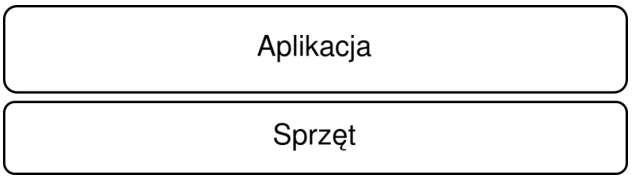
Ten rozdział przedstawia i opisuje architektury oprogramowania w systemach wbudowanych opartych na jednostkach obliczeniowych skonstruowanych na podstawie architektury ARM'owej. Używana w tej pracy definicja pojęcia "system wbudowany"znajduje się w załączniku nr 1. Omówione zostaną także cechy przedstawionych architektur oraz właściwości, które je powodują. Następnie zostaną przedstawione przykładowe realizacje tych architektur. Należy również wspomnieć, że ARM wprowadza także technologię TrustZone, która oferuje kilka dodatkowych architektur. Ponieważ celem tej technologii jest bezpieczeństwo — architektury, które ona wprowadza, nie zostaną tu omówione.

W dalszej części pracy omówiony zostanie termin "wirtualizacja"i terminy pochodne (parawirtualizacja, wirtualizacja natywna), których wyjaśnienie można znaleźć w załączniku nr 2. W załączniku tym omówione są także pojęcia separacji, translacji pamięci i przekierowywania przerwań.

Pojęcia przełączenia stanów i przełączenia kontekstu zostały wyjaśnione w załączniku nr 3.

2.1 Architektura nagi metal

Najprostsza pod względem liczby poziomów abstrakcji architektura, składająca się ze sprzętu i aplikacji wykorzystującej ten sprzęt (rys. 1). Aby funkcjonować jako system wbudowany, potrzebna jest jednostka obliczeniowa dla wykonywania poleceń, pamięć do przechowywania poleceń, moduły wejścia/wyjścia dla kontaktu ze środowiskiem zewnętrznym i kontroler przerwań dla poinformowania aplikacji o zdarzeniach z zewnątrz. Zasoby sprzętowe są przydzielane statycznie podczas kompilacji, nie ma separacji środowisk lub uprzywilejowanego dostępu.



Rysunek 1: Architektura nagi metal

Poniżej przedstawiono najważniejsze cechy:

- Prostota: architektura nie posiada poziomów abstrakcji, aplikacja kontaktuje się bezpośrednio ze sprzętem, nie ma skomplikowanych modułów do zarządzania bezpieczeństwem, np. MMU (ang. Memory Management Unit, układ odpowiedzialny za dostęp do pamięci przez oprogramowanie wykonywane na CPU, jest bardziej złożony niż MPU [15]) lub GIC (ang. General Interrupt Controller, kontroler przerwań skonstruowany dla wirtualizacji oraz obsługi kilku jednostek obliczeniowych na raz [13]);
- Wydajność: architektura nie posiada rozbudowanego kodu systemu operacyjnego ani hiperwizora, nie ma translacji pamięci i przekierowania przerwań, nie ma przełączania kontekstu;
- Determinizm: brak konkurencji za zasoby jednostki obliczeniowej przez aplikacje, z powodu możliwej obecności tylko jednej aplikacji, pełna gwarancja reakcji na zdarzenie;

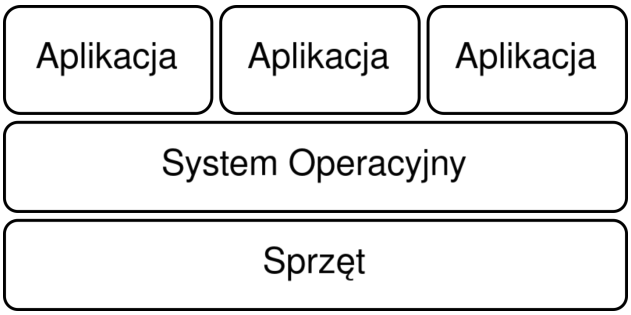
Uwzględniając powyższe cechy i właściwości, można wysnuć wniosek, że taka architektura jest przeznaczona dla małych jednostek w systemie, np. czujników, każda z których ma zdefiniowany jedyny cel. Przykładem realizacji danej architektury może być dowolna aplikacja niewykorzystująca systemu operacyjnego ani hiperwizora działająca na dowolnej architekturze ARM'owej. Chociaż najczęściej wybierana będzie jedna z architektur z rodziny Cortex M (rodzina architektur ARM'owych stworzonych z myślą o małych energooszczędne układach, lub tzw. mikrokontrolery [14]), ponieważ są one najlepiej do tego przystosowane. Dana architektura nie stanowi przedmiotu zainteresowania w ramach tej pracy, ponieważ nie posiada oprogramowania zarządzającego dostępem do zasobów jednostki obliczeniowej.

Architektura posiadająca jedną warstwę abstrakcji, system operacyjny (rys. 2) - jest to oprogramowanie systemowe, które w pełni lub częściowo (zależy od architektury systemu operacyjnego) zachowuje kontrolę nad sprzętem. Opiera się na dodatkowych modułach sprzętowych zarządzających pamięcią, którymi są najczęściej MPU (ang. Memory Protection Unit, układ odpowiedzialny za dostęp do pamięci przez oprogramowanie wykonywane na CPU, jest mniej złożony w porówna-

niu do MMU (typowo posiada tylko mechanizmy zarządzania uprawnieniami) [16]), lub też MMU, aby oddzielić kod systemu operacyjnego od kodu aplikacji, oraz na bardziej złożonym kontrolerze przerwań, obsługującym nie tylko zdarzenia zewnętrzne, ale także zdarzenia wewnętrzne, pozwalające na zmianę stanu systemu.

2.2 Architektura z systemem operacyjnym

Jednostka obliczeniowa ma wbudowany podział stanów systemu: tryb aplikacji, w którym wykonywany kod nie ma dostępu do zasobów systemowych, oraz tryb systemu operacyjnego, mający dostęp do wszystkich zasobów. Dla większości systemów operacyjnych obecność funkcjonalności podziału na tryby ze strony sprzętowej, tj. obecność odpowiednich rejestrów oraz instrukcji ISA (ang. Instruction Set Architecture, architektura jednostki obliczeniowej z punktu widzenia programisty [10]), jest obowiązkowa.



Rysunek 2: Architektura z systemem operacyjnym

Poniżej przedstawiono najważniejsze cechy:

- Wysoki poziom skomplikowania: W zależności od architektury systemu operacyjnego (monolityczne jądro, mikrojądro, nanojądro), system operacyjny może posiadać różną liczbę narzędzi i funkcjonalności. Ogólnie jednak jest to bardziej złożona architektura ze względu na konieczność wykorzystania dodatkowego sprzętu systemowego oraz wprowadzenie kilku poziomów abstrakcji (np. w sterownikach).
- Niższa wydajność: Obecność abstrakcji w systemie operacyjnym (sterowniki, interpretatorzy aplikacji), przełączanie stanów pomiędzy systemem operacyjnym a aplikacją, oraz przełączenie kontekstu pomiędzy aplikacjami skutkuje zmniejszeniem zasobów dostępnych dla aplikacji.
- Większe bezpieczeństwo: Możliwość ”narzucania”zasad (definiowanie dostępów i innych ograniczeń) dla aplikacji z poziomu systemu operacyjnego oraz separacja aplikacji zwiększa bezpieczeństwo.
- Skalowalność: Obecność oprogramowania zarządzającego zasobami jednostki obliczeniowej pozwala na uruchomienie wielu aplikacji oraz definiowanie priorytetów dla każdej z nich oddzielnie, co umożliwia realizację wielu celów przez jeden system wbudowany.
- Mniejszy determinizm: W przypadku obecności wielu aplikacji pojawia się prawdopodobień-

stwo, że zadania przypisane każdej z tych aplikacji zostaną wykonane w innym czasie, niż przewidziano. Wynika to z konkurencji o dostęp do jednostki obliczeniowej między aplikacjami.

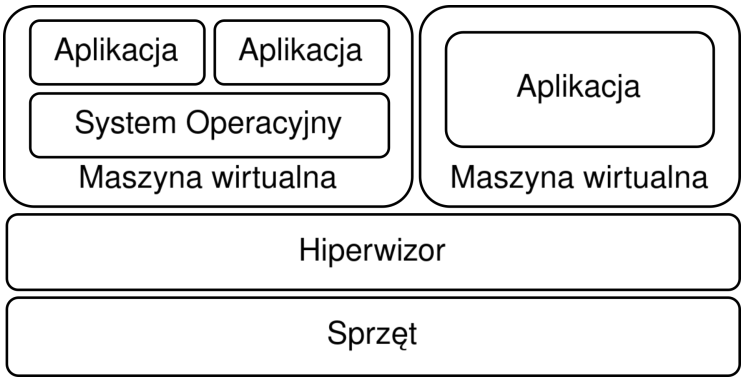
Taką architekturę oprogramowania można znaleźć na wszystkich architekturach ARM'owych, t.j.: Cortex M, Cortex A (rodzina architektur ARM'owych stworzonych z myślą o układach o dużej mocy obliczeniowej i skomplikowanej architekturze oprogramowania [14]) i Cortex R (rodzina architektur ARM'owych stworzonych z myślą o skomplikowanych systemach które są również systemami czasu rzeczywistego [14]). Przykłady jej zastosowania obejmują:

- FreeRTOS;
- Zephyr RTOS;
- Linux;
- FreeBSD, NetBSD;
- Windows;
- seL4.

Architektura z system operacyjnym będzie poruszona w tej pracy, ponieważ zawiera ona oprogramowanie zarządzające zasobami jednostki obliczeniowej, zwane także nadzorcą. Nadzorca jest częścią jądra systemu operacyjnego.

2.3 Architektura z wirtualizacją

Architektura z wirtualizacją jest realizowana poprzez dodania kolejnego poziomu abstrakcji do wcześniej omówionych architektur (rys. 3). Hiperwizor może uruchamiać VM (ang. Virtual Machine, dosłownie - maszyna wirtualna, lub środowisko wirtualne tworzone poprzez VMM [8]) z aplikacją w architekturze ”nagi metał”jak również z systemem operacyjnym, który będzie odpowiedzialny za uruchomianie aplikacji.



Rysunek 3: Architektura z wirtualizacją

Typowo hiperwisory wymagają odpowiedniego sprzętu dla działania (ang. virtualization support). Głównie chodzi o rozszerzenia do obsługi przerwań oraz separacji pamięci, oraz dodaniu oddzielnego stanu jednostki obliczeniowej), np. EL2 (ang. Exception Level, poziom uprawnień przypisywanych wykonującemu oprogramowaniu w architekturze ARM'owej, możliwe poziomy:

EL0, EL1, EL2 i EL3 [12]) w architekturach Cortex A. W takim przypadku wirtualizacja nazywana jest natywną (ang. native virtualization, także ang. full virtualization, lub ang. hardware-accelerated virtualization).

Istnieje także możliwość stworzenia hiperwizora niezależnego od sprzętu, czyli takiego który nie wymaga odpowiedniego sprzętu do działania, jest to tzw. parawirtualizacja (ang. paravirtualization).

Poniżej przedstawiono najważniejsze cechy:

- Bardzo wysoki poziom skomplikowania: Oprócz faktu, że hiperwizor jest przeznaczony do uruchamiania różnych architektur w środowisku wirtualnym, co w przypadku parawirtualizacji może być dużym wezwaniem, dochodzą również kwestie związane z komunikacją pomiędzy środowiskami oraz podziałem dostępu do sprzętu pomiędzy poszczególnymi środowiskami, w tym podziału przerwań. Dodatkowo może być wykorzystywana emulacja.
- Jeszcze mniejsza wydajność: Dalsze obniżenie wydajności wynika głównie z konieczności translacji pamięci, przechwytywania i wirtualizacji przerwań oraz dodatkowego przełączania stanów i kontekstu.
- Jeszcze większe bezpieczeństwo: Do separacji oprogramowania w pamięci dochodzi precyzyjne przypisywanie zasobów sprzętowych do poszczególnych środowisk wirtualnych.
- Konfigurowalny determinizm: Oznacza to, że na jednym urządzeniu można uruchomić dwie lub więcej zupełnie różnych architektur, np. ”nagi metal”i system operacyjny, które charakteryzują się różnymi właściwościami deterministycznymi i rozłożyć priorytety. Można również zmienić sposób zarządzania zasobami jednostki obliczeniowej, przechodząc od dynamicznego zarządzanie do statycznego zarządzania.

Omawianą architekturę oprogramowania systemowego można spotkać na architekturach sprzętowych z rodzin Cortex A i Cortex R, które są wyposażone w specjalny sprzęt wspierający wirtualizację. Możliwe jest również jej uruchomienie na architekturach niewyposażonych w takie wsparcie, jednak w takim przypadku konieczne będzie zastosowanie parawirtualizacji.

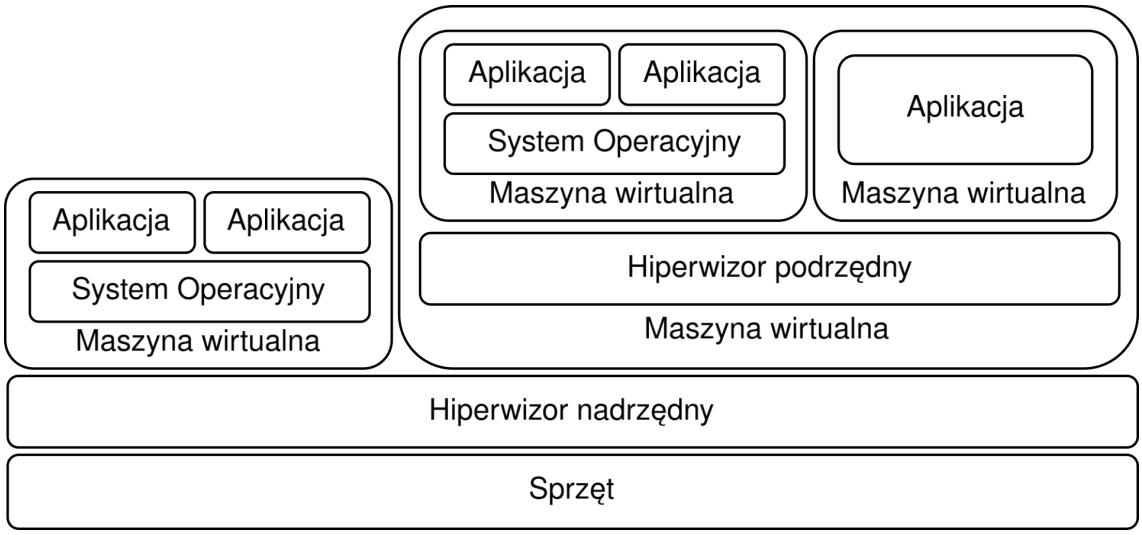
Przykłady realizacji:

- Jailhouse;
- Xen;
- Bao;
- seL4 CAmkES;
- KVM;
- Crosscon Hypervisor.

Główną uwagą w tym przypadku jest fakt, że hiperwizor jest dodatkową warstwę zarządzającą zasobami jednostki obliczeniowej, co oznacza, że jest on także punktem zainteresowania w ramach tej pracy. W przypadku, gdy w środowisku wirtualnym realizowana jest architektura z systemem operacyjnym, można mówić o zagnieżdżonym (dwupoziomowym) zarządzaniu dostępem do zasobów jednostki obliczeniowej.

2.4 Architektura z wirtualizacją zagnieżdżoną

Wirtualizacja zagnieżdżona (ang. nested virtualization) polega na stworzeniu VM wewnątrz VM (rys. 4), co wiąże się z dodaniem też dodanie dodatkowego hiperwizora, który może być nazywany ”hiperwizorem podrzędnym”.



Rysunek 4: Architektura z wirtualizacją zagnieżdżona

Aktualnie architektury ARM’owe wspierają tę architekturę częściowo, tzn. nie ma ani dodanego kolejnego EL, ani wsparcia dla trzypoziomowej translacji pamięci, ani dedykowanych mechanizmów przekierowywania przerwań, jest jedynie podstawowe wsparcie w postaci rozszerzeń do niektórych rejestrów w architekturach z rodziny Cortex A [5].

Ta architektura dodaje kolejną warstwę z oprogramowaniem, które zarządza dostępem do zasobów jednostki obliczeniowej. Oprogramowanie to znajduje się w hiperwizorze podrzędnym, co, w przypadku gdy w VM utworzonej przez hiperwizor podrzędny jest wykorzystywana architektura z systemem operacyjnym, skutkuje trzema poziomami zarządzania dostępem do jednostki obliczeniowej: oprogramowanie w hiperwizorze nadrzędnym, oprogramowanie w hiperwizorze podrzędnym i oprogramowanie w systemie operacyjnym. To stanowiłoby interesujący punkt dla badań, gdyby nie słabe wsparcie tej architektury w społeczeństwie systemów wbudowanych, co powoduje liczne problemy, które nie stanowią centrum uwagi tej pracy [5]. Dlatego ta architektura nie będzie omawiana w ramach tej pracy.

3 Dekompozycja i wyodrębnienie

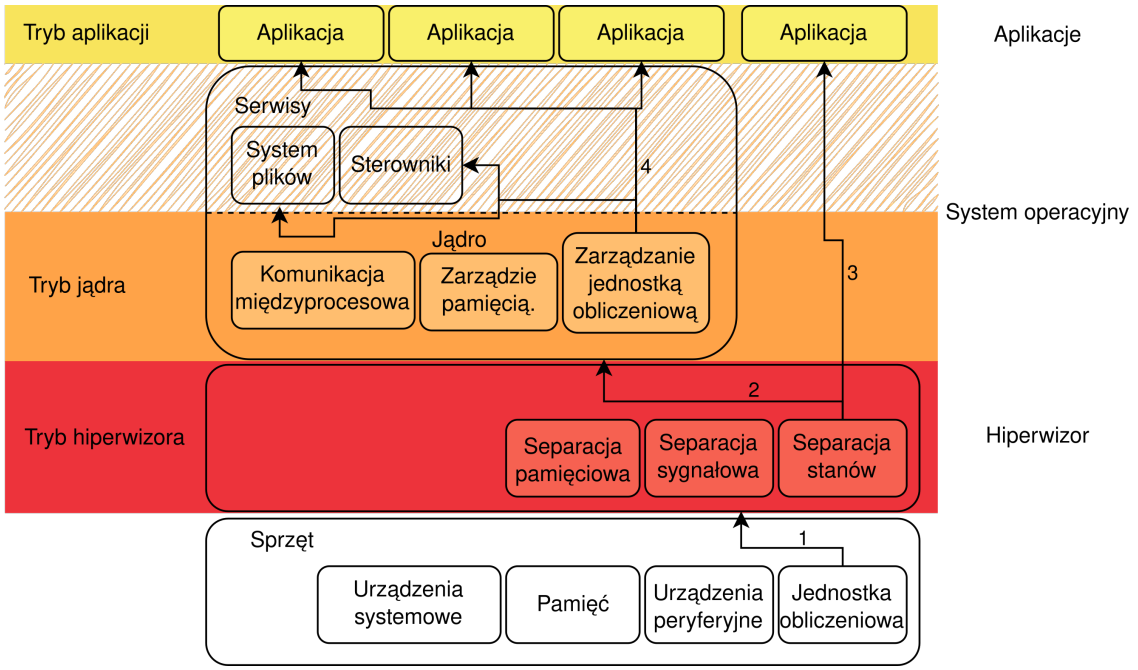
W tej części pracy zostanie przedstawiona dekompozycja oprogramowania systemowego omówionego w poprzedniej części w architekturach z systemem operacyjnym i wirtualizacją, a konkretnie: systemu operacyjnego i hiperwizora. Następnie oprogramowanie odpowiadające za zarządzanie dostępem do zasobów jednostki obliczeniowej zostanie wyodrębnione ze wspomnianego wcześniej oprogramowania systemowego dla przeprowadzenia analizy w następnej części

pracy.

Należy zauważyć, że celem tej pracy nie jest omówienie wszystkich elementów oprogramowania systemowego. Pozostałe elementy oprogramowania systemowego będą wspomniane, jeżeli zajdzie potrzeba, tylko w sposób ogólny, w celu prześledzenie udziału oprogramowania zarządzającego zasobami jednostką obliczeniową w ogólnej architekturze oprogramowania systemowego, oraz umożliwienia wyodrębnienia tej części do dalszej analizy.

3.1 Dekompozycja

Jak wspomniano wcześniej, z punktu widzenia zarządzania zasobami jednostki obliczeniowej szczególnie interesujące są dwa rodzaje oprogramowania systemowego: system operacyjny i hiperwizor. Na rys. 5 przedstawiono przydzielenie zasobów jednostki obliczeniowej za pomocą tego oprogramowania systemowego. Kolorami zaznaczone są tryby, w których znajduje się jednostka obliczeniowa podczas wykonywania instrukcji przez określone oprogramowanie, a zacieniony obszar wskazuje, że serwisy systemu operacyjnego mogą być wykonywane zarówno w trybie jądra, jak i w trybie aplikacji. Strzałkami zaznaczono przydzielenie zasobów jednostki obliczeniowej do określonego oprogramowania przez odpowiednią logikę w oprogramowaniu systemowym.



Rysunek 5: Przydzielenie dostępu do zasobów jednostki obliczeniowej przez oprogramowanie systemowe: system operacyjny i hiperwizor

Ten rysunek ilustruje architekturę z wirtualizacją, podczas gdy architektura bez wirtualizacji wykorzystująca jedynie system operacyjny, może być przedstawiona usuwając z rys. 5 cały hiperwizor i strzałkę o numerze 3. W takim przypadku strzałka o numerze 1 będzie wskazywać bezpośrednio na jądro systemu operacyjnego.

Jest to struktura hierarchiczna, tzn. (numery punktów są odpowiednie do numerów strzałek na rys. 5):

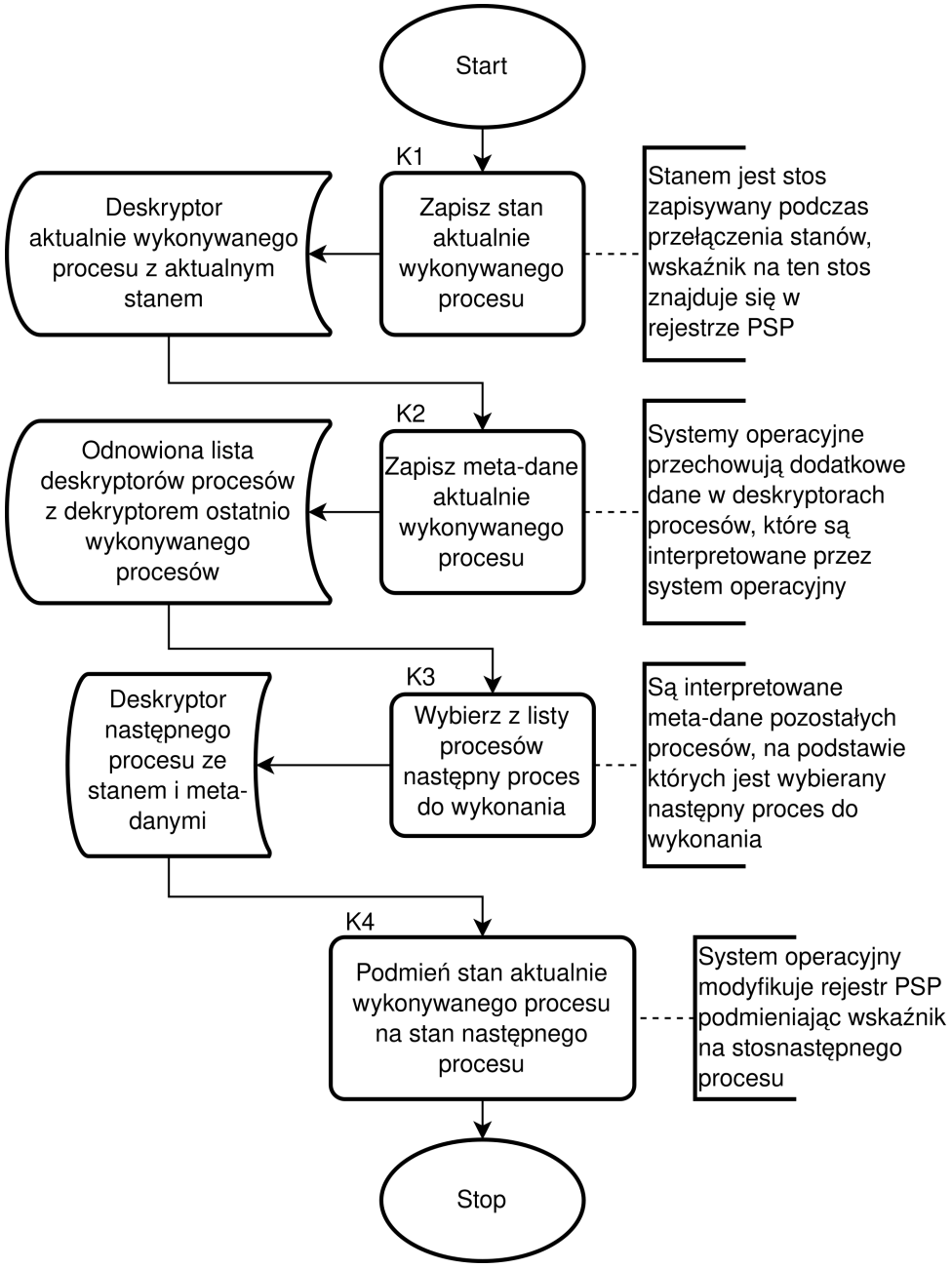
1. Hiperwizor otrzymuje kontrolę nad wszystkimi zasobami jednostki obliczeniowej i przydziela je do oprogramowania, które on uruchamia: system operacyjny (strzałka nr 2 na rys. 5) lub aplikacja (strzałka nr 3 na rys. 5).
2. System operacyjny otrzymuję od hiperwizora pewną część zasobów jednostki obliczeniowej (strzałka nr 2 na rys. 5) i przydziela je do serwisów lub aplikacji (strzałki o wspólnym numerze 4 na rys. 5).
3. Aplikacja uruchomiona bez systemu operacyjnego otrzymuje przydzielone zasoby od hiperwizora (strzałka nr 3 na rys. 5) i zaczyna wykonywać swoją pracę.
4. Aplikację i serwisy uruchomione przez system operacyjny, otrzymują przydzielone im zasoby przez system operacyjny (strzałki o wspólnym numerze 4 na rys. 5) i zaczynają wykonywać swoją pracę.

W systemie operacyjnym funkcjonalność przydzielenia zasobów jednostki obliczeniowej jest jedną z trzech kluczowych, pozostałe dwie to: komunikacja międzyprocesowa i zarządzanie pamięcią. Natomiast w hiperwizorze jest to jeden ze sposobów separacji, pozostałe dwa to: separacja sygnałowa i separacja pamięciowa. Wytłumaczenie separacji jest zamieszczone w załączniku nr 2).

Logikę lub kod, który odpowiada za zarządzanie zasobami jednostki obliczeniowej zarówno w systemie operacyjnym, jak i hiperwizorze (pomiędzy strzałkami 1 i 2, oraz pomiędzy strzałkami 2 i 4, rys. 6) można nazwać nadzorcą (ang. supervisor). Takie nazewnictwo będzie stosowane w dalszej części pracy. Nadzorcę można także podzielić na dwie części: dyspozytora (ang. dispatcher) i planistę (ang. scheduler). Centrum uwagi tej pracy jest planista.

3.2 Wyodrębnienie

Wspomniany w poprzedniej części nadzorca jest wywoływany tylko w pewnych, szczególnie zdefiniowanych przypadkach, aby wykonać przełączenie kontekstu. Wywoływany on może być periodycznie, za pomocą periodycznego przerwania, lub ”na żądanie”za pomocą przerwania systemowego. W przypadku ARM’u są to przerwania generowane instrukcjami SVC (ang. Supervisor Call, przerwanie systemowe, z którego może skorzystać aplikacja, aby skomunikować się z jądrem systemu operacyjnego [12]) i HVC (ang. Hypervisor Call, przerwanie systemowe, z którego może skorzystać system operacyjny, aby skomunikować się z hiperwizorem [12]).



Rysunek 6: Ogólna logika przełączenia kontekstu

Po wywołaniu nadzorcy następuje przełączenie kontekstu. Na rys. 6 ogólnie opisane są kroki

przełączania kontekstu. Przy czym kroki *K1*, *K2* i *K4* (rys. 6) są wykonywane przez dyspozytora, a tylko krok *K3* (rys. 6) jest wykonywany przez planistę.

Zadaniem dyspozytora jest otrzymywanie informacji i przekierowanie jej w odpowiednim formacie do planisty (kroki *K1* i *K2* na rys. 6), który podejmuje decyzję na podstawie zdefiniowanych reguł (krok *K3* na rys. 6), i informuje o tym dyspozytora. Dyspozytor podejmuje odpowiednie dla podjętej decyzji działania (krok *K4* na rys. 6).

Wejściem do planisty jest lista deskryptorów procesów oczekujących na przydzielenie zasobów obliczeniowych oraz ilość dostępnych zasobów obliczeniowych. Wyjściem jest deskryptor procesu z przydzielonymi zasobami jednostki obliczeniowej. Planista podejmuje decyzję na podstawie wstępnie zdefiniowanych reguł. Reguły te będą omówione w następnej części pracy.

4 Algorytmy

W tej części pracy zostanie szczególnie omówiona i przeanalizowana jedna z części nadzorcy — planista, zostaną przedstawione i omówione algorytmy zarządzania zasobami jednostki obliczeniowej realizowane przez planistę. Pojęcie zasobu jednostki obliczeniowej jest wyjaśnione w załączniku nr 4.

4.1 Rodzaje algorytmów

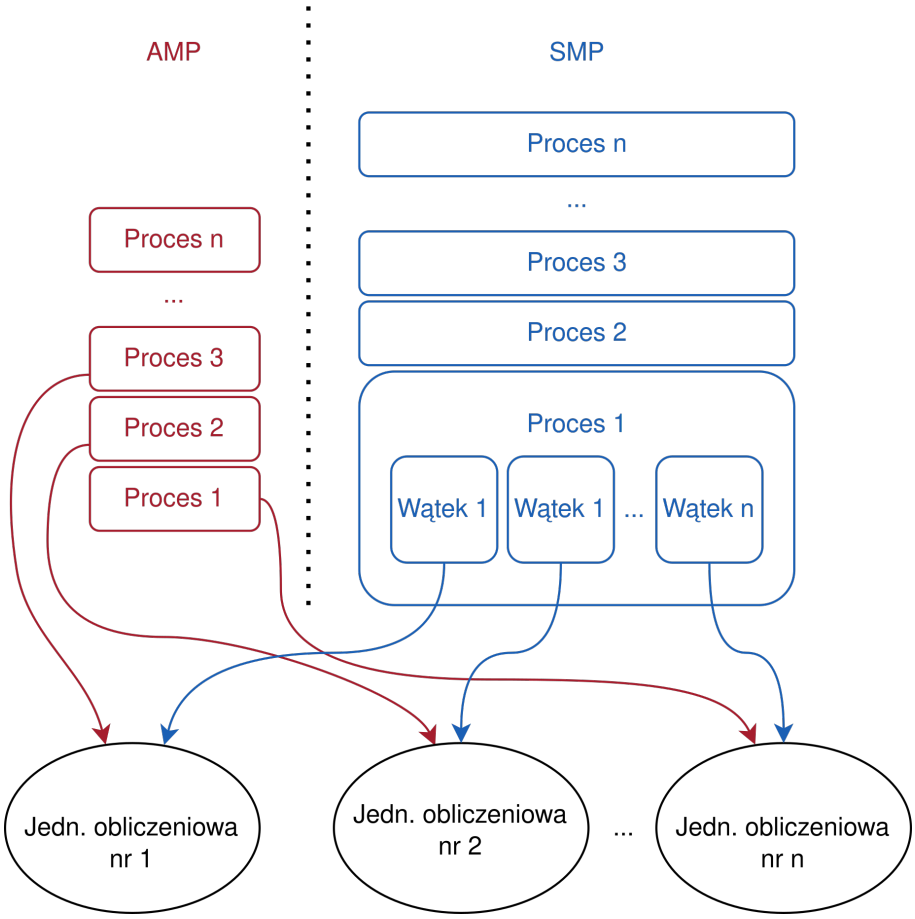
4.1.1 Techniki wielordzeniowe

Najpierw należy zauważyć, że jeden system może zawierać więcej niż jedną jednostkę obliczeniową. Jednostki te mogą być zorganizowane w sposób umożliwiający równoczesne wykonywanie instrukcji procesów dostępnych dla wszystkich jednostek (tzw. wielordzeniowość; ang. multicore). W takich przypadkach są stosowane techniki AMP (ang. Asynchronous Multiprocessing, techniki zarządzania kilkoma jednostkami obliczeniowymi w sposób, w którym każdej jednoscę jest przydzielane oddzielny proces do wykonania [19]) lub SMP (ang. Synchronous Multiprocessing, techniki zarządzania kilkoma jednostkami obliczeniowymi w sposób, w którym proces jest dzielony na wątki które są przydzielane każdej jednostkie obliczeniowej, to znaczy wszystkie jednostki obliczeniowe zostają na pewny okres czasu przepisane do jednego procesu [20]).

Na rys. 7 przedstawiono różnię pomiędzy AMP a SMP. W przypadku AMP każdej jednostce obliczeniowej przypisywany jest oddzielny proces, natomiast w przypadku SMP każdy proces jest dzielony na wątki, a każdemu wątkowi przypisywane są zasoby poszczególnych jednostek obliczeniowych.

W niniejszej pracy techniki SMP nie będą omawiane, ponieważ ich zastosowanie wymaga, aby zarówno oprogramowanie systemowe, jak i program uruchamiany na jego podstawie, były napisane z myślą o użycie wątków (np. POSIX PThreads, znany także jako IEEE 1003.1c-1995). Z jednej strony wprowadza ona znaczne skomplikowanie, a z drugiej strony nie wprowadza znacznych zmian w algorytmy zarządzania zasobami jednostki obliczeniowej. Zmiana polega jedynie na

tym, że zamiast przypisywania zasobów procesom, planista przydziela je wątkom, które z punktu widzenia planisty nie różnią się od procesów. Zarówno wątki, jak i procesy posiadają listę instrukcji do wykonania, stos oraz metadane interpretowane przez planistę.



Rysunek 7: Różnica pomiędzy technikami AMP i SMP

Więc, w ramach niniejszej pracy, w przypadku obecności wielu jednostek obliczeniowych,analizowane będą wyłącznie techniki AMP.

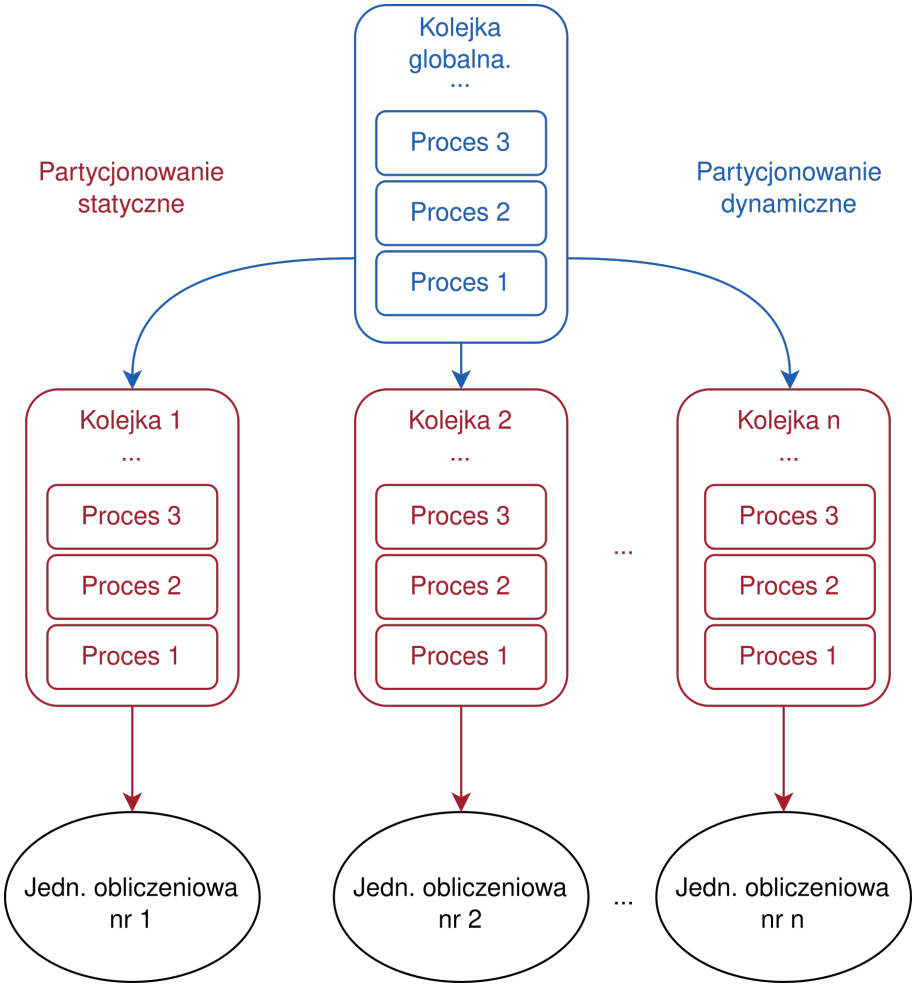
4.1.2 Techniki AMP

AMP oznacza, że każdej jednostce obliczeniowej przypisany jest jeden proces rys. 7. Powstaje wtedy pytanie: kiedy i w jaki sposób każdemu procesowi przydzielana jest jednostka obliczeniowa?

Istnieją dwie zupełnie różne techniki rozwiązujące ten problem: partycjonowanie statyczne (ang. static partitioning scheduling; rys. 8) oraz zarządzanie globalne (global scheduling rys. 9). Oprócz nich istnieje także trzecia technika, wymyślona w celu zaadresowania problemów poprzednich dwóch technik — partycjonowanie dynamiczne (ang. dynamic partitioning, znana również jako ang. semi-static partitioning; rys. 8). [17]

Partycjonowanie statyczne opiera się na podziale jednostek obliczeniowych pomiędzy wszystkie procesy jeszcze przed uruchomieniem nadzorcy. Typowo, tworzone są kolejki przypisane do każdej jednostki obliczeniowej (rys. 8). Decyzja o tym, jaki proces otrzyma zasoby jednostki ob-

liczeniowej z danej kolejki, należy do planisty, który zarządza wykonaniem procesów w każdej z kolejek. Głównym problemem tego podejścia jest sytuacja, w której wszystkie procesy w pewnych kolejkach znajdują się w stanie nieaktywnym — wówczas jednostki obliczeniowe przypisane do tych kolejek pozostają niewykorzystane, co prowadzi do marnowania zasobów.

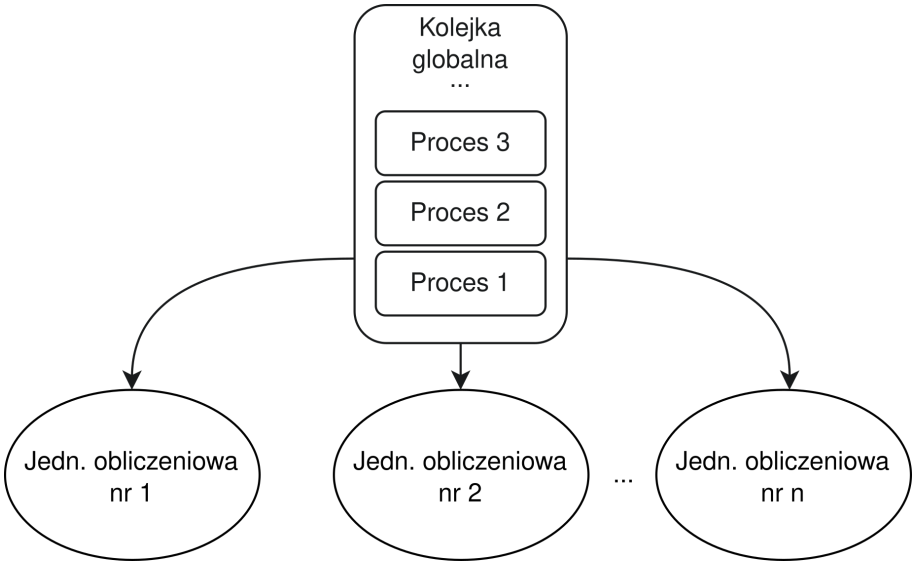


Rysunek 8: Techniki AMP: partycjonowanie statyczne i partycjonowanie dynamiczne

Zarządzanie globalne opiera się na jednym planiście, który przydziela zasoby wszystkich jednostek obliczeniowych procesom znajdującym się w jednej wspólnej kolejce (rys. 9). Głównym problemem tego podejścia jest migracja zadań pomiędzy jednostkami obliczeniowymi, która wymaga realizacji skomplikowanej logiki. Powoduję to nadużycie zasobów obliczeniowych przez planistę i dystrybutora.

Partycjonowanie dynamiczne jest połączeniem dwóch wcześniej opisanych technik. Używane są kolejki dla każdej z dostępnych jednostek obliczeniowych, zgodnie z tym, jak to jest robione w partycjonowaniu statycznym, oraz kolejka globalna, podobna do tej stosowanej w zarządzaniu dynamicznym. Pomysł polega na podziale procesów na dwie grupy: procesy, które są statycznie przywiązane do określonej jednostki obliczeniowej, oraz procesy, które ”wędrują”pomiędzy jednostkami obliczeniowymi. Celem tej techniki jest minimalizacja skutków problemów opisanych w poprzednich technikach:

- Jeśli procesy w statycznie zdefiniowanej kolejce będą nieaktywne, proces z kolejki globalnej zostanie przydzielony do swobodnej jednostki obliczeniowej.
- Ponieważ w kolejce globalnej znajduje się tylko część procesów, częstość migracji procesów będzie zmniejszona, co oznacza, że mniej czasu będzie marnowane na działanie planisty i dyspozytora.



Rysunek 9: Technika AMP: zarządzanie globalne

Te techniki nie definiują, dlaczego i w jaki sposób każdy proces otrzymuję zasoby jednostki obliczeniowej, lecz określają sposoby rozprowadzenia obciążenia w systemie z wieloma jednostkami obliczeniowymi. Algorytmy przypisywania zasobów poszczególnych jednostek obliczeniowych do przypisanych do tych jednostek procesów są ukryte w planistach przypisanych do każdej kolejki i nazywane są technikami jednordzeniowymi. Będzie to omówione w następnej części pracy.

4.1.3 Techniki jednordzeniowe

W przypadku, gdy w systemie jest obecna tylko jedna jednostka obliczeniowa lub gdy zachodzi potrzeba zaimplementowania planisty dla poszczególnych kolejek w technikach AMP, jest mowa o technikach jednordzeniowych. Techniki te można podzielić na grupy, które posiadają niektóre z właściwości pokazanych na rys. 10.

Wywłaszczenie to właściwość, która pozwala nadzorcy na interwencję w wykonywanie procesu i jego podmianę na inny proces, kiedy poprzedni jeszcze nie osiągnął swojego celu. Taka właściwość umożliwia planistowi podejmowanie decyzji o tym, który proces będzie wykonywany następnie, periodycznie, a nie tylko po zakończeniu wykonywania każdego z procesów.

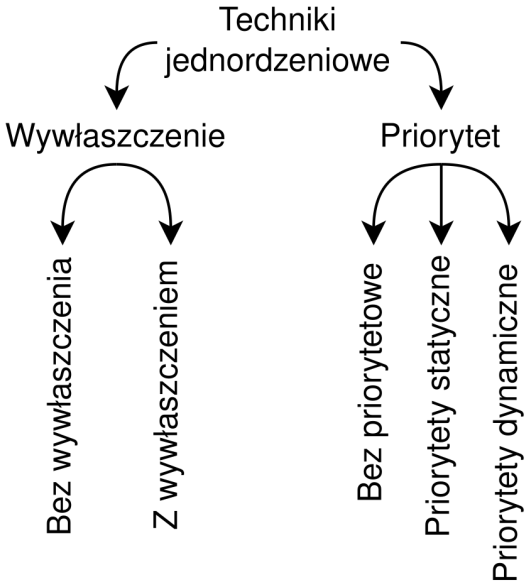
Priorytet pozwala na nierównomierne traktowanie procesów przy przypisywaniu im zasobów jednostki obliczeniowej, co wprowadza dodatkowe możliwości optymalizacji algorytmów w zależności od określonych celów. Priorytety mogą być statyczne, zdefiniowane przed uruchomieniem nadzorcy, lub dynamiczne, przeliczane w czasie wykonywania nadzorcy.

Lista przykładowych algorytmów:

- FCFS — ang. First Come First Served, to algorytm bezpriorytetowy, liniowy, niewykorzystujący wywłaszczenia. Zasada działania: proces, który jako pierwszy trafia do kolejki, otrzymuje tyle czasu, ile potrzebuje na osiągnięcia swojego celu, a kolejne procesy są obsługiwane w takiej samej kolejności. Ten algorytm nie interpretuje żadnych metadanych procesu, tylko przydziela czas jednostki obliczeniowej pierwszemu procesowi w kolejce.
- SJF — ang. Shortest Job First, jest to algorytm z priorytetami statycznymi, niewykorzystujący wywłaszczenia. Zasada działania: podczas przełączania kontekstu wybierany jest proces, który ma najmniejszy czas potrzebny na zakończenie zadania. Algorytm zakłada, że czas wykonania procesu jest znany przed rozpoczęciem jego realizacji.
- SRTN — ang. Shortest Remaning Time Next, to algorytm z dynamicznymi priorytetami i wywłaszczeniem. Zasada działania: podczas przełączania kontekstu wybierany jest proces, który ma najmniejszy czas pozostały do zakończenia zadania. Ponieważ czas pozostały do wykonania procesu zmienia się za każdym razem, gdy proces otrzymuje czas jednostki obliczeniowej, priorytet jest dynamiczny. Algorytm zakłada, że podczas przełączenia kontekstu dostępna będzie informacja o czasie pozostałym do zakończenia wykonywania procesu.
- RR — ang. Round Robin, algorytm bezpriorytetowy wykorzystujący wywłaszczanie. Zasada działania: każdemu procesowi przypisywana jest określona część zasobów jednostki obliczeniowej (tzw. kwant czasu). Gdy proces zużyje przypisany kwant, jednostka obliczeniowa jest przekazywana do kolejnego procesu w kolejce, i tak w kółko, dopóki każdy proces nie zakończy swojego wykonywania.
- RM — ang. Rate-Monotonic, algorytm ze statycznymi priorytetami wykorzystujący wywłaszczanie. Zasada działania: priorytet jest obliczany przed uruchomieniem nadzorcy, najwyższy priorytet ma proces o najmniejszej częstości pojawienia w systemie (ang. rate); proces o wyższym priorytecie wywłaszcza proces o mniejszym priorytecie i używa zasobów jednostki obliczeniowej, dopóki nie osiągnie swojego celu; następnie do wykonania zostaje przewrócony proces o niższym priorytecie. Ten algorytm zakłada, że częstość pojawienia się procesu w systemie jest znana.
- DM — ang. Deadline-Monotonic, algorytm ze statycznymi priorytetami wykorzystujący wywłaszczanie. Zasada działania: priorytet jest obliczany przed uruchomieniem nadzorcy, najwyższy priorytet ma proces z najkrótszym najgorszym czasem osiągnięcia celu (ang. deadline); proces o wyższym priorytecie wywłaszcza proces o niższym i używa zasobów jednostki obliczeniowej, dopóki nie osiągnie swojego celu; następnie do wykonania zostaje przewrócony proces o niższym priorytecie. Ten algorytm zakłada, że najgorszy cas osiągnięcia celu przez proces w systemie jest znany.
- EDF — ang. Earliest Deadline First, algorytm wykorzystujący dynamiczne priorytety i wykorzystujący lub niewykorzystujący wywłaszczanie, co zależy od konfiguracji. Zasada działania: najwyższy priorytet otrzymuję proces o najbliższym najgorszym terminie osiągnięcia

celu (ang. deadline).

- LST lub LLF — ang. Least Slack Time lub Least Laxity First, algorytm wykorzystujący dynamiczne priorytety i wykorzystujący lub niewykorzystujący wywłaszczenie, co zależy od konfiguracji. Zasada działania: najwyższy priorytet otrzymuje proces, który ma najmniejszy czas pomiędzy osiągnięciem celu a najgorszym terminem osiągnięcia celu.
- DARTS — ang. Dynamic Real Time Task Scheduling, algorytm wykorzystujący dynamiczne priorytety i wywłaszczenie. Jest to połączenie algorytmów EDF i LLF, w którym uwzględniany jest ostateczny termin osiągnięcia celu przez proces. Dokładniejsza zasada działania tego algorytmu będzie wyjaśniona w następnej części pracy. [7]



Rysunek 10: Techniki jednordzeniowe

Istnieje kilka sposobów przeprowadzenia analizy algorytmów zarządzania zasobami jednostki obliczeniowej w systemach operacyjnych, m.in. analiza matematyczna i symulacja. Natomiast w ramach tej pracy zdecydowano przeprowadzić analizę na podstawie realnego systemu, odpowiednio go modyfikując, co zostanie opisane w następnej części pracy.

5 System operacyjny

W tej części pracy zostanie omówiona analiza algorytmów zarządzania zasobami jednostki obliczeniowej w systemie z architekturą oprogramowania wykorzystującą systemem operacyjny (rys. 2).

Do tych celów wybrano FreeRTOS, jako jedno z najbardziej stabilnych i najdłużej istniejących mikrojąder na rynku z otwartym kodem źródłowym. Dodatkową zaletą jest fakt, że FreeRTOS korzysta z licencji MIT, która pozwala na dowolne modyfikowanie kodu źródłowego.

FreeRTOS charakteryzuje się następującymi cechami:

- Jest mikrojądrem.

- Posiada funkcjonalności:
  - Komunikacja międzyprocesowa: powiadomienia bezpośrednie między procesami, kolejki, strumienie, bufory.
  - Zarządzanie jednostką obliczeniową: jeden algorytm priorytetowy z wywłaszczaniem i kwantyzacją czasu.
  - Zarządzanie pamięcią: dostępnych jest kilka realizacji dynamicznej alokacji pamięci oraz jedna realizacja statycznej alokacji pamięci.
  - Synchronizacja: mutexy i semafory.
  - Liczniki realizowane przez oprogramowanie (ang. software timers).
- Wsparcie dla większości architektur sprzętowych.

Jest to zatem dobry wybór, który pozwala szybko przejść przez wprowadzenie do oprogramowania i skupić się na docelowym zadaniu.

5.1 Używana platforma

Ograniczenia dotyczące platformy, na której ma być uruchomiony system operacyjny:

- Możliwość uruchomienia systemu operacyjnego bez konieczności dodawania wsparcia dla platformy;
- Możliwość debugowania i śledzenia wykonywania instrukcji na platformie bez interwencji w ciąg wykonywanych instrukcji (ang. real time tracing).

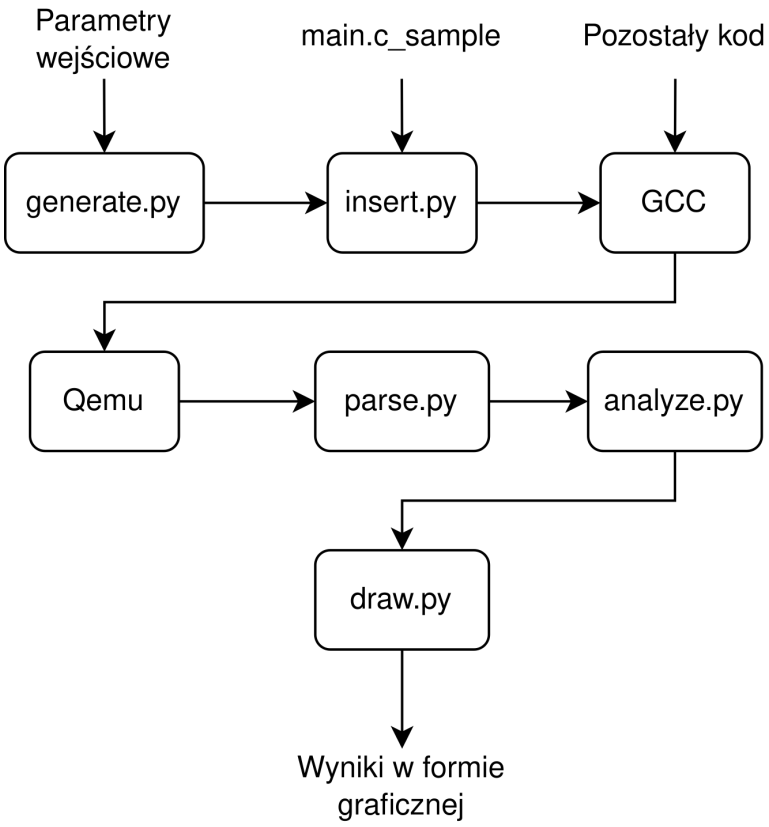
Początkowo planowano użyć w tym celu platformy NUCLEO-64 F401RE, ponieważ wykorzystuje ona mikrokontroler oparty na jednostce obliczeniowej ARM Cortex M4, bazującej na architekturze ARMv7M. Jest to dobrze znana i dokładnie zbadana architektura, która jest wspierana przez FreeRTOS.

Problem pojawił się przy debugowaniu i śledzeniu wykonywanych instrukcji bez interwencji. Mikrokontroler ten posiada wbudowane narzędzie ETM (ang. Embedded Trace Macrocell, narzędzie w architekturach ARM'owych pozwalające na bezingerencyjne śledzenie pracy jednostek obliczeniowych i elementow do nich podłączonych [6]) które umożliwia realizację takiego zadania, jednak, do jego wykorzystania wymagany jest debugger (np. Segger J-Link), którego koszt był poza budżetem autora. W związku z tym zdecydowano zrezygnować z tej platformy.

Jako alternatywę dla platformy sprzętowej zdecydowano użyć emulatora Qemu, który może emulować docelową architekturę. Jako emulowaną płytkę wybrano Stellaris LM3S6965EVB. Jest to platforma producenta Texas Instruments oparta na mikrokontrolerze LM3S6965 z jednym rdzeniem Cortex M3, które również jest oparte na architekturze ARMv7M. Platforma ta jest wspierana przez FreeRTOS i posiada oddzielny projekt przykładowy, co ułatwiło przeprowadzenie eksperymentów. W dalszej części pracy ta emulowana platforma będzie nazywana po prostu ”platformą”lub ”LM3S6965EVB”, w zależności od potrzeby rozróżnienia.

5.2 Sposób pomiaru i analizy

Skoro platforma docelowa jest emulowana przez Qemu, istnieje możliwość skorzystania ze wszystkich funkcji Qemu do śledzenia zachowań oprogramowania. ”Śledzenie”oznacza ciągle odczytywanie wykonywanych instrukcji lub funkcji uruchomianych na platformie i zapisywanie tych danych, w przypadku Qemu, do pliku tekstowego. Ponieważ Qemu wykonuje instrukcje z prędkością zbliżoną do rzeczywistej platformy, plik tekstowy gromadzi w ciągu kilku sekund działania platformy miliony linii danych, co uniemożliwia ich manualną analizę.



Rysunek 11: Zestaw narzędzi i połączenie z Qemu

Aby to przeanalizować, stworzono zestaw narzędzi w języku Python, który automatyzuje ten proces. Finalna architektura jest pokazana na rys. 11, gdzie każdy wątek odpowiada za:

- *generate.py*: generuje plik JSON zawierający metadane procesów.
- *insert.py*: na podstawie metadanych procesów wygenerowanych w poprzednim tworzy funkcje w języku C (ponieważ każdy proces jest reprezentowany jako funkcja w FreeRTOS) z wykorzystaniem parametrów z wygenerowanego przez *generate.py* pliku JSON. Następnie wygenerowane funkcje są wklejane do odpowiednio przygotowanego pliku *main.c_sample*, a otrzymany plik o nazwie *main.c* jest zapisywany w systemie plików, gdzie on może zostać znaleziony przez kompilator.
- GCC (ang. GNU Compiler Collection, zestaw narzędzi do kompilacji o otwartym kodzie źródłowym rozwijany w ramach projektu GNU [9]): otrzymuje *main.c* oraz pozostały kod systemu

operacyjnego, kompiluje, FreeRTOS przechodząc przez cały proces kompilacji programu w języku C, tzn. od etapu preprocesora do linkowania, tworząc plik binarny dla architektury ARMv7M. Plik ten jest zapisywany w systemie plików, gdzie on może być znaleziony przez Qemu.

- Qemu: odczytuje plik binarny i uruchamia go. Po uruchomieniu system operacyjny przechodzi przez proces inicjalizacji i zaczyna wykonywać wcześniej zdefiniowane procesy. W trakcie wykonywania Qemu zapisuje informacje o wykonywanych funkcjach systemowych do pliku tekstowego, który jest zapisywany w systemie plików, gdzie on może być znaleziony i przeanalizowany.
- *parse.py*: otwiera i analizuje plik tekstowy zawierający informacje o wykonywanych funkcjach, zapisane wcześniej przez Qemu. Po analizie dane dotyczące każdego z procesów są zapisywane w pliku JSON w systemie plików. Zapisywane dane to: rozpoczęcie wykonywania każdego z procesów, zakończenie wykonywania każdego z procesów oraz liczba wywłaszczeń dla każdego z procesów.
- *analyze.py*: odczytuje plik JSON otrzymany z poprzedniego kroku i oblicza następujące wskaźniki: liczbę sukcesów w osiągnięciu celu przed najgorszym czasem osiągnięcia celu dla każdego z procesów, liczbę porażek w osiągnięciu celu przed najgorszym czasem osiągnięcia celu dla każdego z procesów.
- *draw.py*: tworzy wykresy na podstawie pliku JSON wygenerowanego w poprzednim kroku.

Każdy z elementów przedstawionych na rys. 11 jest szczegółowo omówiony w dalszej części pracy.

5.2.1 Problemy i założenia

Podczas tworzenia wyżej przedstawionej listy narzędzi ujawniło się kilka problemów, które wpłynęły nie tylko na proces pomiaru i analizy, ale także na proces generowania metadanych procesów, jak również i na implementację algorytmów.

Wykorzystanie Qemu zamiast rzeczywistej płytki wprowadziło komplikacje, ponieważ Qemu jest emulatorem uruchamianym wewnątrz systemu operacyjnego użytkownika (w ramach tej pracy używano dystrybucji Linux'a o nazwie Ubuntu w wersji 22.04.1 na architekturze sprzętowej x86_64) jako aplikacja. Oznacza to, że Qemu dzieli zasoby jednostki obliczeniowej użytkownika z innymi procesami, co sprawia, że w pewnym momencie Qemu może nie otrzymać czasu na wykonanie swoich instrukcji, co z kolei oznacza, że instrukcje Qemu nie są wykonywane w czasie rzeczywistym.

Z kolei, Qemu uruchamia wewnątrz siebie emulowaną platformę, na której działa badany system operacyjny. Skoro instrukcje Qemu nie są wykonywane w czasie rzeczywistym, emulowana platforma także nie działa w czasie rzeczywistym. Oznacza to, że jeżeli zegar systemowy emulowanej platformy został skonfigurowany na 50 MHz, to nie oznacza, że po uruchomieniu Qemu, rejestr przechowujący wartość zegara systemowego będzie modyfikowany z częstotliwością 50

MHz. Może to być dowolna inna częstotliwość, zależna od zasobów jednostki obliczeniowej użytkownika oraz od okresu, w jakim Qemu je otrzymuje. Może to nawet nie być periodyczne. Z tego wynika, że nie można mierzyć czas potrzebny na wykonanie czegokolwiek uruchamianego wewnątrz Qemu w sekundach ani częstości w hercach. W takim przypadku, w jaki sposób mierzyć i określać czas wykonywania procesów oraz momenty rozpoczęcia i zakończenia ich wykonywania?

Pierwszym pomysłem było użycie wartości zegara systemowego emulowanej przez Qemu architektury, czyli mierzenie wszystkich wielkości systemowych w różnicach pomiędzy wartościami zegara, które są dostępne w pewnym miejscu w pamięci. Jest to możliwe do zrobienia wewnątrz systemu (tzn. FreeRTOS może odczytywać wartości zegara), jak i z zewnątrz. Analiza ciągłego odczytu pamięci emulowanej architektury umożliwia też analizę wykonywanych działań w systemie, ponieważ oprócz rejestru ze stanem zegara w pamięci można znaleźć inne rejestry, np. wskaźnik na aktualnie używany przez emulowaną jednostkę obliczeniową stos, co pozwala na śledzenie wykonywanych procesów.

Byłoby to dobre rozwiązanie, gdyby nie problem, że Qemu nie emuluje mikroarchitekturę (tzn. właściwości sprzętowych), lecz tylko architekturę widzianą przez programistę systemowego i aplikacji. Oznacza to, że Qemu emuluje wykonywanie instrukcji dla określonej architektury (ARMv7M), ale nie emuluje, jak te instrukcje są wykonywane w odpowiedniej mikroarchitekturze.

Głównym problemem jest to, że nie wiadomo, ile okresów emulowanego zegara systemowego zajmie wykonywanie jednej instrukcji emulowanej architektury. Na przykład, dla Cortex M3 czas potrzebny na wykonanie instrukcji *cmp* wynosi jeden okres zegara systemowego. Kiedy Cortex M3 jest emulowany przez Qemu, nie ma pewności, czy instrukcja *cmp* zostanie wykonana w ciągu jednego okresu emulowanego zegara systemowego.

Uwzględniając powyższe, można wyciągnąć wniosek, że pomiar emulowanego zegara systemowego i porównywanie zdarzeń na podstawie tych pomiarów nie zawsze będzie sensowne. Ponieważ pomiędzy na przykład, dwoma periodycznymi wydarzeniami w systemie, rejestr zawierający wartość zegara systemowego może być zmodyfikowany o różne wartości, czasem wartość wzrośnie o jedną jednostkę, a innym razem to będzie inna jednostka.

Aby móc analizować efektywność używanych algorytmów, symulować czas potrzebny na osiągniecie celu przez proces oraz przeprowadzać pomiary, jako miarę czasu (czyli minimalny odcinek czasu) wybrano wartość zegara systemu operacyjnego, zwanego w FreeRTOS jako *SysTick* (ang. System Tick). Nie jest istotne, ile realnego czasu minie pomiędzy chwilami, w których zegar systemu operacyjnego zmienia swój stan. Ważne jest, że ta zmiana się odbyła, a pomiędzy chwilami zmian stanu zegara wykonywane były inne operacje, które trwały dokładnie tyle czasu, ile minęło pomiędzy tymi dwoma zmianami stanu zegara systemu operacyjnego.

Czyli teraz czas wykonywania procesu będzie mierzony nie w sekundach lub ich ułamkach, lecz w okresach zegara systemu operacyjnego. Na przykład, można powiedzieć, że proces potrzebuje 100 okresów zegara systemu operacyjnego, aby osiągnąć swój cel albo, że proces jest

periodyczny i pomiędzy dwoma momentami czasu, kiedy proces przechodzi do gotowości do wykonania, mija 500 okresów zegara systemu operacyjnego, i tak dalej. Jednostka, reprezentująca okres zegara systemu operacyjnego, w dalszej części pracy będzie nazywana *SysTick*.

Owszem, takie założenie jest użyteczne, gdy nie jest ważne, ile dokładnie czasu minęło pomiędzy zmianami stanu zegara systemu operacyjnego.

Podczas pomiaru i analizy nie będzie uwzględniony czas wykonywania jakichkolwiek innych funkcji systemu operacyjnego, ponieważ: większość funkcji systemu operacyjnego została usunięta podczas kompilacji jako zbędne w ramach tej pracy, oraz dlatego, że celem tej pracy jest determinizm, a nie wydajność systemu. Uwzględniając powyższe założenia, można wyciągnąć wniosek, że wydajność końcowego systemu w ramach tej pracy nie wpłynie na analizę determinizmu, ponieważ jako miara czasu przyjęto fakt zmiany stanu zegara systemu operacyjnego, nie uwzględniając, ile realnego czasu minęło pomiędzy dwoma zmianami, ani ile czasu zajęła ta zmiana.

5.2.2 Generacja parametrów, tworzenie procesów

Generacja parametrów dla tworzenia procesów, jak zostało to opisane powyżej (rys. 11), odbywa się za pomocą skryptu *generate.py*. Skrypt ten otrzymuje następujące parametry w formacie JSON (zamieszczone poniżej dane są przykładowe):

Przykład 1: Parametry wejściowe do generacji

```
{
    "TaskAmount": 10,
    "PriorityOverlapping": 0,
    "StdMeanPeriod": [1600,100],
    "LowUpPeriod": [2,30],
    "StdMeanDeadline": [1400,100],
    "LowUpDeadline": [2,1],
    "StdMeanExec": [20,10],
    "LowUpExec": [10,300],
    "StdMeanPriority": [126,250],
    "LowUpPriority": [1,251]
}
```

Gdzie:

- *TaskAmount*: liczba procesów.
- *PriorityOverlapping*: flaga sygnalizująca, czy priorytety zdefiniowane dla procesów mogą się powtarzać.
- *StdMeanPeriod*: parametry rozkładu normalnego (rozkładu Gaussa) używanego do generowania okresu gotowości do wykonywania procesów periodycznych. Pierwszy element tabeli to odchylenie standardowe, a drugi to wartość oczekiwana.
- *LowUpPeriod*: ograniczenia dla wartości okresu, odpowiednio dolne i górne;

- *StdMeanDeadline*: parametry rozkładu normalnego (rozkładu Gaussa) używanego do generowania najgorszego czasu osiągnięcia celu przez proces. Pierwszy element tabeli to odchylenie standardowe, a drugi to wartość oczekiwana.
- *LowUpDeadline*: ograniczenia dla wartości najgorszego czasu osiągnięcia celu przez proces, odpowiednio dolne i górne.
- *StdMeanExec*: parametry rozkładu normalnego (rozkładu Gaussa) używanego do generowania czasu potrzebnego procesowi na osiągnięcie swojego celu. Pierwszy element tabeli to odchylenie standardowe, a drugi to wartość oczekiwana.
- *LowUpExec*: ograniczenia dla wartości czasu potrzebnego procesowi na osiągnięcie swojego celu, odpowiednio dolne i górne.
- *StdMeanPriority*: parametry rozkładu normalnego (rozkładu Gaussa) używanego do generowania priorytetu przypisywanego każdemu procesowi. Pierwszy element tabeli to odchylenie standardowe, a drugi to wartość oczekiwana.
- *LowUpPriority*: ograniczenia dla wartości priorytetu, odpowiednio dolne i górne.

Rozkłady prawdopodobieństwa zostały wybrane do generowania parametrów procesów w celu przybliżenia do rozkładu parametrów w rzeczywistych sytuacjach. Oznacza to, że w rzeczywistości procesy nie mogą mieć identycznych lub bardzo podobnych parametrów, tylko ich wartości są rozproszone na określonym przedziale. Zadaniem rozkładów prawdopodobieństwa jest symulacja tej niejednorodności.

Wszystkie wymienione powyżej parametry są dobierane ręcznie w celu zasymulowania przybliżonego do rzeczywistości przypadku. Na przykład, można wygenerować zestaw parametrów procesów, w którym w systemie będzie występowała duża liczba procesów o niskich priorytetach oraz długie procesy o wyższych priorytetach, a następnie przekazać te dane do Qemu, aby sprawdzić pewny algorytm w takich warunkach.

Wygenerowane dane są zapisywane w formacie JSON w następującej postaci (przykład dla jednego procesu):

Przykład 2: Wygenerowane parametry procesu

```
{
    "v1_Task": {
        "TaskName": "v1_Task",
        "TaskPriority": 57,
        "TaskExecutionTime": 13,
        "TaskDeadline": 30,
        "TaskPeriod": 106
    }
}
```

Gdzie:

- *TaskName*: nazwa procesu.
- *TaskPriority*: priorytet procesu.

- *TaskExecutionTime*: czas potrzebny procesowi na osiągnięcie celu, jednostka - *SysTick*.
- *TaskDeadline*: najgorszy czas osiągnięcia celu przez proces, jednostka - *SysTick*.
- *TaskPeriod*: okres gotowości do wykonywania procesu, jednostka - *SysTick*.

Te dane są interpretowane przez skrypt *insert.py*, który generuje kod w języku C i wkleja go do wspomnianego wyżej (rys. 11) pliku *main.c_sample*. Na przykładzie 3 pokazano wygenerowany kod w języku C dla danych z przykładu 2:

Przykład 3: Wygenerowany kod w języku C

```
void v1_Task(void *pvParameters);

int main(void){
  xTaskCreate(v1_Task, "v1_Task", TASK_STACK_LENGHT_WORDS, NULL, 57, 13, 30, NULL);

  vTaskStartScheduler();
  while(1){};
}

void v1_Task (void *pvParameters){
  TickType_t xLastWakeTime = 0;

  while (1){
    pxCurrentTCB->xTaskCurrentExecutionTime = pxCurrentTCB->xTaskExecutionTime;
    do{}while( pxCurrentTCB->xTaskCurrentExecutionTime != 0 );

    vTaskDelayUntil(&xLastWakeTime, 106);
    }
    vTaskDelete(NULL);
}
```

A więc wygenerowany został prototyp funkcji reprezentującej proces, polecenie tworzenia procesu w systemie operacyjnym za pomocą funkcji *xTaskCreate* i procesu w postaci definicji funkcji *v1_Task*. Należy również zauważyć, że jest to część wygenerowanego pliku, obcięta w celu prezentacji przykładu. Nie zawiera ona wszystkich definicji i poleceń.

5.2.3 Kompilacja i wybór algorytmów

Kompilacja systemu operacyjnego była przeprowadzona za pomocą narzędzi Make i GCC. Narzędzie Make organizuje cały kod źródłowy w jeden projekt i przekazuje go do narzędzi GCC.

Wśród zbieranego przez Make kodu źródłowego znajduje się plik *FreeRTOS.h*, który zawiera definicje preprocesora języka C konfigurujące FreeRTOS. W pliku tym dodano także definicje umożliwiające wybór używanego algorytmu.

5.2.4 Sposób zbierania danych o procesach

Zgodnie z założeniem opisanym powyżej, wszystkie operacje w systemie są mierzone w okresach zegara systemu operacyjnego (czyli FreeRTOS). Pozostaje jedynie znaleźć sposób, w jaki można mierzyć te okresy oraz wydarzenia spowodowane zmianami wartości zegara systemowego.

Jak już wspomniano, do tego celu można wykorzystać możliwość odczytywania pomięci emulowanego systemu i późniejszej jej analizy. Ten sposób został uznany przez autora za zbyt skomplikowany. Jako alternatywę zdecydowano skorzystać z funkcji Qemu, która pokazuje, które funkcje są wykonywane w danym momencie w emulowanym systemie. Przykład:

Przykład 4: Przykładowe logi z Qemu

```
Stopped execution of TB chain before 0x7df2b4031880 v1_Task
Trace 0: 0x7df2b401e280 SysTick_Handler
Trace 0: 0x7df2b401e480 SysTick_Handler
Trace 0: 0x7df2b401e700 xTaskIncrementTick
Trace 0: 0x7df2b401f200 xTaskIncrementTick
Trace 0: 0x7df2b401f980 SysTick_Handler
Trace 0: 0x7df2b40200c0 SysTick_Handler
Trace 0: 0x7df2b4031880 v1_Task
Stopped execution of TB chain before 0x7df2b4031880 v1_Task
Trace 0: 0x7df2b401e280 SysTick_Handler
Trace 0: 0x7df2b401e480 SysTick_Handler
Trace 0: 0x7df2b401e700 xTaskIncrementTick
Trace 0: 0x7df2b401f200 xTaskIncrementTick
Trace 0: 0x7df2b401f980 SysTick_Handler
Trace 0: 0x7df2b40200c0 SysTick_Handler
Trace 0: 0x7df2b4031880 v1_Task
```

Jest to fragment logów wykonywania funkcji FreeRTOS z Qemu. Widać tu funkcje: *v1_Task*, *SysTick_Handler* i *xTaskIncrementTick*. Pierwsza funkcja jest jednym z procesów zdefiniowanych przed uruchomieniem systemu, zaś ostatnie dwie są funkcjami jądra FreeRTOS, odpowiadającymi za modyfikację wartości zegara systemu operacyjnego.

Mając wiedzę o tych funkcjach i o kolejności ich wykonywania, można wywnioskować, co jest robione w danym momencie w systemie operacyjnym. Na przykład w powyższym przykładzie przedstawiono dwa momenty zmiany stanu zegara systemu operacyjnego, wraz z procesem, który jest wykonywany pomiędzy tymi zmianami.

Za pomocą tej wiedzy można stworzyć listę reguł opisujących pewne działania podejmowane przez system operacyjny. Ponieważ logi te są dość obszerne, zdecydowano napisać skrypt w języku Python, który automatycznie analizuje logi, wykorzystując te reguły do wykrywania potrzebnych dla analizy procesów zachodzących wewnątrz systemu operacyjnego. Jak wspomniano wcześniej (rys. 11), skrypt ten nazywa się *parse.py*.

W tym skrypcie były zdefiniowane reguły, na podstawie których można wykryć następujące

wydarzenia:

- Moment, w którym proces po raz pierwszy pojawia się w systemie.
- Moment, w którym proces ponownie zaczyna swoje wykonywanie.
- Moment, w którym proces zostaje wywłaszczony przez inny proces.
- Moment, w którym proces kontynuuje wykonywanie po wywłaszczeniu.
- Moment, w którym proces kończy wykonywanie.
- Moment, w którym modyfikowana jest wartość zegara systemu operacyjnego.

Skrypt zapisuje, przy jakiej wartości zegara systemu operacyjnego te wydarzenia występują. Wartość zegara jest obliczana również na podstawie rejestrowanej działalności systemu operacyjnego, zgodnie z przykładem powyżej. Przykład zapisanych danych w formacie JSON w przypadku wykonywania jednego procesu:

Przykład 5: Przykładowe pomiary dla jednego procesu

```
"v1_Task": {
    "TaskName": "v1_Task",
    "TaskStart": [8,44],
    "TaskEnd": [11,47],
    "TaskPreemptedTimes": 0
},
```

Gdzie:

- *TaskName*: zawiera nazwę procesu.
- *TaskStart*: tabela zawierająca stany zegara systemu operacyjnego, kiedy proces rozpoczął swoje wykonywanie, jednostka - *SysTick*.
- *TaskEnd*: tabela zawierająca stany zegara, kiedy proces zakończył swoje wykonanie, jednostka - *SysTick*.
- *TaskPreemptedTimes*: liczba razy, kiedy proces został wywłaszczony w analizowanym odcinku czasu.

5.2.5 Analiza i wyniki w postaci graficznej

Dane uzyskane ze skryptu *parse.py* są przekazywane do skryptu *analyze.py*, który je analizuje i generuje niezbędne dane do opisu zachowania procesów w analizowanym odcinku. Poniżej znajdują się przykładowe dane dla przypadku z czterema procesami:

Przykład 6: Przykładowe wyniki analizy dla czterech procesów

```
{
    "MetDeadline": {
        "v3_Task": 5,
        "v2_Task": 3,
        "v4_Task": 4,
        "v1_Task": 2
    },
```

```
    "MissDeadline": {
        "v3_Task": 0,
        "v2_Task": 0,
        "v4_Task": 0,
        "v1_Task": 0
    },
    "LoadFactor": 0.5951709401709402,
    "MeanTaskExecutionTime": 2.75
}
```

Gdzie:

- *MetDeadline*: lista procesów oraz wartości, które pokazują, ile razy w badanym przypadku każdy z procesów osiągnął swój cel, spełniając ograniczenia czasowe; ten parametr nie ma jednostek.
- *MissDeadline*: lista procesów oraz wartości, które pokazują, ile razy w badanym przypadku każdy z procesów osiągnął swoją cel, nie spełniając ograniczeń czasowych; ten parametr nie ma jednostek.
- *LoadFactor*: obciążenie systemu dla analizowanego przypadku; ten parametr nie ma jednostek.
- *MeanTaskExecutionTime*: średni czas potrzebny procesowi na osiągnięcie swojego celu, mierzony w jednostkach zegara systemu operacyjnego.

Oprócz danych otrzymanych z poprzedniego kroku *analyze.py* potrzebuje także danych wygenerowanych przez skrypt *generate.py*. Alg. 1 przedstawia algorytm obliczenia spełnienia ograniczeń czasowych (czyli *MetDeadline* i *MissDeadline*) na podstawie danych uzyskanych ze skryptów *parse.py* i *analyze.py*.

Algorytm 1 Algorytm liczący spełnienie ograniczeń czasowych dla każdego z procesów

```
1:  for proces in lista procesów do
2:      MetDeadline[proces]=0
3:      MissDeadline[proces]=0
4:      for all indeks in oblicz rozmiar TaskEnd[proces] do
5:          czas gotowości do wykonywania = indeks * TaskPeriod[proces]
6:          najgorszy czas osiągnięcia celu = czas gotowości do wykonywania + TaskDeadline[proces]
7:          if TaskEnd[proces] <= najgorszy czas osiągnięcia celu then
8:              MetDeadline[proces] = MetDeadline[proces] + 1
9:          else if TaskEnd[proces] > najgorszy czas osiągnięcia celu then
10:             MissDeadline[proces] = MissDeadline[proces] + 1
11:         end if
12:     end for
13: end for
```

Natomiast obciążenie systemu (ang. utilization factor) oblicza sie według następującej formuły:

$$U = \sum_{i=1}^{n} \frac{T_{ex\_i}}{T_{ready\_i}} \tag{1}$$

Gdzie: U - współczynnik obciążenia systemu; n - liczba procesów, T_{ex_i} - czas potrzebny procesowi na osiągnięcie celu, jednostka: *SysTick*; T_{ready_i} - czas, kiedy każdy proces jest przypisywany do listy *pxReadyTasksLists*, jednostka: *SysTick*.

Średni czas potrzebny procesowi na osiągnięcie celu oblicza się zgodnie z następującą formułą:

$$T_{exav} = \frac{1}{n} \sum_{i=1}^{n} T_{ex\_i} \tag{2}$$

Gdzie: T_{exav} - średni czas potrzebny procesowi na osiągnięcie celu, jednostka: *SysTick*; n - liczba procesów; T_{ex_i} - czas potrzebny każdemu procesowi na osiągnięcie celu, jednostka: *SysTick*.

Następnie skrypt *draw.py* zbiera dane od skryptu *analyze.py* i generuje wykresy na ich podstawie danych.

5.3 Implementacja wybranych algorytmów w FreeRTOS

W tej części pracy omówione zostaną wybrane algorytmy zarządzania zasobami jednostki obliczeniowej oraz ich implementacja w FreeRTOS.

5.3.1 Standardowe algorytmy w FreeRTOS

FreeRTOS ma jeden standardowy algorytm, który posiada trzy odmiany. Algorytm ten nazywa się HPF (ang. Highest Priority First). Zasada jego działania jest następująca: zasoby jednostki obliczeniowej jako pierwszy otrzymuje proces o największym statycznie zdefiniowanym priorytecie. Priorytet jest definiowany przez użytkownika przy programowaniu systemu i nie zmienia się w trakcie jego działania.

Początkowo ten algorytm nie wykorzystuje wywłaszczania, ale wywłaszczanie można włączyć, dodając do pliku *FreeRTOSConfig.h* definicję preprocesora *configUSE_PREEMPTION* z wartością *1*. To jest druga odmiana algorytmu. Trzecią odmianą jest dodanie współdzielenia czasu przez procesy o tym samym priorytecie. Jest to możliwe po dodaniu do pliku *FreeRTOSConfig.h* definicji preprocesora *configUSE_TIME_SLICING* o wartości *1*.

5.3.2 Kluczowe pojęcia w FreeRTOS

Metadane każdego procesu w FreeRTOS są przechowywane w strukturach zwanych TCB (ang. Task Control Block, struktura w FreeRTOS przechowująca metadane procesu [4]). Na przykładzie 7 przedstawiony jest TCB używany podczas pracy przed wprowadzeniem modyfikacji.

Przykład 7: Przykładowy TCB przed zmianami

```
typedef struct tskTaskControlBlock
{
```

```
    volatile StackType_t * pxTopOfStack;
    ListItem_t xStateListItem;
    ListItem_t xEventListItem;
    UBaseType_t uxPriority;
    StackType_t * pxStack;
    char pcTaskName[ ( 10 ) ];
} tskTCB;
```

Gdzie:

- *pxTopOfStack*: wskaźnik na ostatni element w stosie procesu, potrzebny dla dyspozytora.
- *xStateListItem*: lista, do której proces jest aktualnie przypisany. Może to być lista procesów uśpionych (*pxDelayedTaskList*) lub lista procesów gotowych do wykonania (*pxReadyTasksLists*). W zależności od listy, do której jest przypisany proces, można wnioskować o stanie tego procesu, tj. czy jest uśpiony, czy gotowy do wykonywania.
- *xEventListItem*: podstawowa funkcjonalność FreeRTOS, która nie jest używana w tej prace, a więc nie będzie tu opisana.
- *uxPriority*: zmienna przechowująca priorytet procesu.
- *pxStack*: wskaźnik na stos procesu.
- *pcTaskName*: tabela z nazwą procesu.

Struktura pokazana na przykładzie 7 przeszła już przez preprocesor języka C, więc jest pozbawiona wszystkich zbędnych elementów, które są niepotrzebnych w ramach tej pracy.

Proces jest tworzony jako funkcja w języku C (przykład 3), i rejestrowany w systemie za pomocą funkcji *xTaskCreate*. Po zarejestrowaniu wszystkich procesów w systemie uruchamiany jest nadzorca systemu operacyjnego za pomocą funkcji *vTaskStartScheduler*. Podczas uruchomienia nadzorcy, tworzony jest tzw. proces zerowy, który jest wykonywany, gdy żaden inny proces nie jest gotowy do wykonania. W trakcie pracy systemu procesom są przydzielane zasoby jednostki obliczeniowej zgodnie z zasadami zdefiniowanymi przez planistę.

Podczas wykonywania procesy zmieniają swoje stany. W trakcie tej pracy będą używane tylko dwa stany procesu: gotowy do wykonywania i uśpiony. Stany te są definiowane poprzez przepisanie procesu do odpowiedniej listy, tj. *pxReadyTasksLists* i *pxDelayedTaskList*. Te listy są strukturami w języku C. Ich dokładny opis nie jest potrzebny, ponieważ nie będą one modyfikowane w ramach tej pracy.

Podczas pracy FreeRTOS przechowuje wskaźnik na TCB aktualnie wykonywanego procesu w globalnej zmiennej *pxCurrentTCB*.

5.3.3 Problemy i założenia

Napotkane problemy:

- Problem z priorytetami w FreeRTOS.
- Problem z symulacją czasu wykonywania procesu.

- Problem z metadanymi procesów.

W tej prace wybrane do integracji w FreeRTOS algorytmy zarządzania zasobami jednostki obliczeniowej posiadają różne właściwości. Problemem jest to, że FreeRTOS został zbudowany w sposób, który ściśle wiąże cały kod systemu operacyjnego z algorytmem, a dokładniej ze sposobem definiowania, zarządzania i używania priorytetów procesów. Priorytety te są używane wszędzie w kodzie FreeRTOS, a co najważniejsze, listy, do których są przepisywane wskaźniki na TCB (czyli *pxReadyTasksLists* i *pxDelayedTaskList*), także wykorzystują priorytety.

Struktura kodu systemu operacyjnego, która używa pewnych danych systemowych do definiowania większości obiektów systemu, ma sens, gdy te dane są definiowane przed definiowaniem tych pozostałych obiektów systemu.

Jest to dobre rozwiązanie dla standardowego algorytmu używanego w FreeRTOS, ponieważ w takim przypadku priorytety, które są używane do definiowania większości obiektów w FreeRTOS, są definiowane przez użytkownika na etapie programowania. Natomiast w ramach tej pracy będą implementowane algorytmy, które nie używają priorytetów (np. FCFS), lub algorytmy używające priorytetów dynamicznych.

Aby zaimplementować te algorytmy, wprowadzono modyfikacje dostosowujące FreeRTOS do przypadków, gdy priorytety są dynamiczne lub nie są używane w ogóle. Aktywacja tych modyfikacji i przywrócenie standardowego kodu jest możliwa za pomocą manipulacji definicją preprocesora języka C o nazwie *USE_FREERTOS_CLASSIC_SCHEDULER* w pliku *FreeRTOSConfig.h*.

Podczas badań potrzebny był również sposób na symulację czasu, który proces potrzebuje na osiągnięcie swojego celu. Ponieważ w ramach tego zadania analizowana będzie wydajność algorytmów, cel procesu nie jest istotny, jak również nie ma znaczenia, w jaki sposób każdy z procesów będzie ten cel osiągał. Ważne jest natomiast, aby można było łatwo i dokładnie określić, ile czasu potrzebuje proces na osiągnięcie swojego celu.

Uwzględniając powyższe, zdecydowano symulować wykonywanie procesu za pomocą pętli. Na przykładzie 8 pokazano wybraną strukturę procesu.

Przykład 8: Używana struktura procesu

```
void v1_Task (void *pvParameters){
  TickType_t xLastWakeTime = 0;

  while (1){
    pxCurrentTCB->xTaskCurrentExecutionTime = pxCurrentTCB->xTaskExecutionTime;
    do{}while( pxCurrentTCB->xTaskCurrentExecutionTime != 0 );

    vTaskDelayUntil(&xLastWakeTime, 106);
  }

  vTaskDelete(NULL);
}
```

Gdzie:

- *v1_Task*: przykładowa nazwa procesu.
- *pvParameters*: parametry, które mogą być przekazane procesowi. Nie są one wykorzystywane w tej prace i nie będą tu omówione.
- *xLastWakeTime*: zmienna potrzebna dla periodycznego uśpienia procesu po osiągnięciu celu.
- *xTaskCurrentExecutionTime*: zmienna przechowująca czas, którego proces potrzebuje do osiągnięcia celu, zmniejszana o jeden przy każdej zmianie stanu zegara systemu operacyjnego.
- *xTaskExecutionTime*: czas, który proces potrzebuje na osiągnięcie celu od początku, jest statycznie definiowany i niezmieniany podczas wykonywania.
- *vTaskDelayUntil*: funkcja powodująca uśpienie procesu na zdefiniowany okres, w tym przypadku 106 SysTick. Ten okres jest okresem gotowości procesu do wykonywania, znanym także jako *TaskPeriod* (było to przedstawione na przykładzie 2).
- *vTaskDelete*: funkcja, która teoretycznie nigdy nie musi wykonać się. Jest częścią struktury procesu zgodnie z wymogami FreeRTOS i nie będzie tu opisana.

Jak widać z przykładu 8, proces wykonuje się, dopóki czas potrzebny na jego wykonanie (zdefiniowany statycznie przed uruchomieniem systemu poprzez zmienną *xTaskExecutionTime*) nie upłynie (co jest sygnalizowane przez porównanie zmiennej *xTaskCurrentExecutionTime* do zera). Podczas wykonywania proces znajduje się w pustej pętli i niczego nie robi. Wartość zmiennej *xTaskExecutionTime* jest generowana przez skrypt *generate.py* (czyli jest to *TaskExecutionTime* z przykładu 2), zaś wartość zmiennej *xTaskCurrentExecutionTime* jest dynamicznie definiowana podczas rejestrowania procesu, oraz, jak widać na przykładzie 8, kiedy proces wychodzi z uśpienia (czyli funkcja *vTaskDelayUntil* kończy swoje wykonywanie).

Należy również zauważyć, że zmienna *pxCurrentTCB* nie powinna być udostępniana procesom i powinna być dostępna tylko dla jądra systemu operacyjnego. Natomiast w ramach tej pracy pominięto kwestie bezpieczeństwa i standardów napisania kodu, aby w krótszym casie zaimplementować potrzebne algorytmy i przeprowadzić potrzebne badania. Zresztą, te kwestie nie są w zakresie tej pracy, więc można je pominąć.

Jak widać na przykładzie 8, do TCB zostały dodane nowe elementy pozwalające na symulację czasu potrzebnego procesowi na osiągnięcie swojego celu, są to zmienne *xTaskCurrentExecutionTime* i *xTaskExecutionTime*.

Oprócz tych dwóch nowych elementów, dodano także kilka innych elementów, potrzebnych do implementacji niektórych algorytmów. Nowy TCB, który był wykorzystywany w ramach tej pracy, jest pokazany na przykładzie 9

Przykład 9: Zmodyfikowany TCB

```
typedef struct tskTaskControlBlock
{
    volatile StackType_t * pxTopOfStack;
```

```
    ListItem_t xStateListItem;
    ListItem_t xEventListItem;
    StackType_t * pxStack;
    TickType_t xTaskStarted;
    TickType_t xTaskTimeExecuted;
    TickType_t xTaskExecutionTime;
    TickType_t xTaskCurrentExecutionTime;
    TickType_t xTaskExecutionPeriod;
    TickType_t xTaskExecutionDeadline;
    char pcTaskName[ ( 10 ) ];
} tskTCB;
```

Gdzie nowe elementy (w porównaniu do przykładu 8):

- *xTaskStarted*: przechowuje wartość zegara systemu operacyjnego, kiedy proces został przypisany do listy procesów gotowych do wykonywania.
- *xTaskTimeExecuted*: przechowuje liczbę zmian stanu zegara systemu operacyjnego od momentu, kiedy proces zaczął wykonywanie. W przypadku, gdy proces zaczął wykonywanie, a następnie został wywłaszczony przez inny proces, logowanie czasu do tej zmiennej jest wstrzymywane do momentu, kiedy proces powróci do wykonania po wywłaszczeniu.
- *xTaskExecutionTime*: wyjaśnione w przykładzie 8.
- *xTaskCurrentExecutionTime*: wyjaśnione w przykładzie 8.
- *xTaskExecutionPeriod*: okres gotowości procesu do wykonywania, generowany przez skrypt *generate.py* (zgodnie z 2 to jest *TaskPeriod*).
- *xTaskExecutionDeadline*: najgorszy czas osiągnięcia celu przez proces, generowany przez skrypt *generate.py* (zgodnie z 2 to jest *TaskDeadline*).

Niektóre z tych nowych elementów są inicjalizowane statycznie podczas rejestrowania procesu, a inne są modyfikowane podczas pracy systemu.

5.3.4 Sposób integracji nowych algorytmów

Poprzednio pokazano (rys. 5), że za zarządzanie zasobami jednostki obliczeniowej w systemie operacyjnym jest odpowiedzialny jeden moduł, zwany także nadzorcą. Następnie nadzorca był podzielony (rys. 6) na dyspozytora i planistę.

Jednak w rzeczywistości funkcje planisty i dyspozytora pełni nie jedna funkcja, a moduł ten nie jest umieszczony w jednym konkretnym miejscu w kodzie systemu operacyjnego, tylko jest to kod rozrzucony po całym systemie, który jednak można podzielić na części i wyodrębnić. Podział taki będzie specyficzny dla każdego systemu operacyjnego, ponieważ zależy od jego implementacji.

Natomiast w FreeRTOS planista może zostać podzielony na następujące części:

1. Kod, który jest wykonywany przy rejestracji procesu w systemie. W FreeRTOS ten kod znajduje się wewnątrz funkcji *prvAddNewTaskToReadyList*.
2. Kod, który jest wykonywany w pewnych przypadkach powodujących przełączenie kontekstu.

W FreeRTOS ten kod znajduje się wewnątrz funkcji *vTaskSwitchContext*.

3. Kod, który jest wykonywany periodycznie. W FreeRTOS ten kod znajduje się wewnątrz *xTaskIncrementTick*.

To, który z tych punktów trzeba zaimplementować, zależy od algorytmu. Punkt drugi jest jednak zawsze obowiązkowy. Na przykład punkt trzeci pewnie będzie potrzebny algorytmom, które odwołują się do planisty przy każdej zmianie wartości zegara systemowego, niezależnie od tego, czy wystąpiło zdarzenie powodujące zmianę kontekstu (np. algorytmy DARTS i RR tego potrzebują).

Wyjaśnienia implementacji poszczególnych algorytmów w dalszej części pracy będą odwoływać się do tych trzech punktów.

5.3.5 FCFS

Alg. 2 demonstruje algorytm FCFS w postaci pseudokodu.

Algorytm 2 Pseudokod algorytmu FCFS

```
1: pxCurrentTCB = Wybierz pierwszy proces z listy(pxReadyTasksLists)
```

W tym przykładzie widać nazwy napisane bez spacji oraz nazwy napisane kursywą. Tu i w dalszej części pracy nazwy napisane bez spacji będą oznaczać zmienne, natomiast nazwy napisane bez spacji z użyciem kursywy będą oznaczać zmienne istniejące w kodzie systemu operacyjnego.

Potrzebne modyfikacje jądra FreeRTOS:

1. Wybranie pierwszego zarejestrowanego procesu jako pierwszego procesu gotowego do wykonywania jeszcze przed uruchomieniem nadzorcy, czyli wewnątrz funkcji *prvAddNewTaskToReadyList*.
2. Wybranie pierwszego procesu z kolejki *pxReadyTasksLists* do wykonania w momencie, gdy aktualny proces osiąga swój cel i jednostka obliczeniowa zostaje zwolniona, czyli potrzebna była modyfikacja funkcji *vTaskSwitchContext*.
3. Każdy proces zmieniający swój stan na stan gotowy do wykonywania musi być dodany do końca listy *pxReadyTasksLists*.

W standardowej implementacji FreeRTOS wszystkie procesysą dodawane na koniec listy *pxReadyTasksLists*, więc implementacja trzeciego punktu nie była potrzebna.

5.3.6 SJF

Alg. 3 demonstruje pseudokod algorytmu SJF.

Algorytm 3 Pseudokod algorytmu SJF

```
1: for all procesTCB in pxReadyTasksLists do
2:     if procesTCB.xTaskExecutionTime < pxCurrentTCB.xTaskExecutionTime then
3:         pxCurrentTCB=procesTCB
4:     end if
5: end for
```

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany został do wykonania proces zgodnie z algorytmem SJF.
2. Modyfikacja funkcji *vTaskSwitchContext* aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej były przedzielane procesowi zgodnie z algorytmem SJF.

5.3.7 SRTN

Alg. 4 demonstruje pseudokod algorytmu SRTN.

Algorytm 4 Pseudokod algorytmu SRTN

```
1: for all procesTCB in pxReadyTasksLists do
2:     procesTCB.RemainingExecutionTime = Oblicz RemaningExecutionTime dla procesu(procesTCB)
3:     pxCurrentTCB.RemainingExecutionTime = Oblicz RemaningExecutionTime dla procesu(pxCurrentTCB)
4:     if procesTCB.RemainingExecutionTime < pxCurrentTCB.RemainingExecutionTime then
5:         pxCurrentTCB=procesTCB
6:     end if
7: end for
```

RemaningExecutionTime z tego przykładu jest obliczany zgodnie z formułą:

$$R(t) = T_{ex} - T_{excu}(t) \tag{3}$$

Gdzie: $R(t)$ - czas do osiągnięcia celu przez proces w danej chwili; T_{ex} - całkowity czas potrzebny procesu na osiągnięcie celu; T_{excu} czas, który proces już zużył na osiągnięcie celu w danej chwili.

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany do wykonania został proces zgodnie z algorytmem SRTN.
2. Modyfikacja funkcji *vTaskSwitchContext* aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej były przedzielane procesowi zgodnie z algorytmem SRTN.
3. Modyfikacja funkcji *xTaskIncrementTick*, aby zaimplementować funkcjonalność wywłaszczania.

5.3.8 RR

Alg. 5 demonstruje pseudokod algorytmu RR.

Algorytm 5 Pseudokod algorytmu RR

```
1: if pxCurrentTCB.xTaskTimeExecuted >= kwant czasu then
2:     pxCurrentTCB = Wybierz pierwszy proces z listy(pxReadyTasksLists)
3: end if
```

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany do wykonania został proces zgodnie z algorytmem RR.
2. Modyfikacja funkcji *vTaskSwitchContext*, aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej zostały przedzielone procesowi zgodnie z algorytmem RR.
3. Modyfikacja funkcji *xTaskIncrementTick*, aby zaimplementować funkcjonalność wywłaszczania w przypadku, gdy aktualnie wykonywany proces wyczerpie wydany jemu kwant czasu.

5.3.9 RM

Alg. 6 demonstruje pseudokod algorytmu RM.

Algorytm 6 Pseudokod algorytmu RM

```
1:  if Lista jest pusta(pxReadyTasksLists) then
2:      TCBNowegoProcesu.uxPriority = najniższy priorytet
3:  else
4:      for proces in pxReadyTasksKists do
5:          if proces.xTaskPeriod > TCBNowegoProcesu.xTaskPeriod then
6:              TCBNowegoProcesu.uxPriority = proces.uxPriority + 1
7:          else if proces.xTaskPeriod < TCBNowegoProcesu.xTaskPeriod then
8:              TCBNowegoProcesu.uxPriority = proces.uxPriority
9:              proces.uxPriority = proces.uxPriority + 1
10:         else
11:             TCBNowegoProcesu.uxPriority = proces.uxPriority
12:         end if
13:     end for
14: end if
15: Zarejestruj nowy proces(TCBNowegoProcesu)
```

Ten algorytm używa tej samej logiki priorytetowej co standardowy algorytm FreeRTOS, więc większość kodu nie wymaga zmian. Konieczna jest tylko jedna modyfikacja: zmiana funkcji *prvAddNewTaskToReadyList*, aby przy rejestracji priorytety procesów zostały przeliczone zgodnie z algorytmem RM.

Niestety, nie udało się zaimplementować tego algorytmu, więc nie będą przedstawione wyniki jego testów.

5.3.10 DM

Alg. 7 demonstruje pseudokod algorytmu DM.

Algorytm 7 Pseudokod algorytmu DM

```
1: if Lista jest pusta(pxReadyTasksLists) then
2:     TCBNowegoProcesu.uxPriority = najniższy priorytet
3: else
4:     for proces in pxReadyTasksLists do
5:         if proces.xTaskExecutionDeadline > TCBNowegoProcesu.xTaskExecutionDeadline
   then
6:             TCBNowegoProcesu.uxPriority = proces.uxPriority + 1
7:         else if proces.xTaskExecutionDeadline < TCBNowegoPro-
   cesu.xTaskExecutionDeadline then
8:             TCBNowegoProcesu.uxPriority = proces.uxPriority
9:             proces.uxPriority = proces.uxPriority + 1
10:        else
11:            TCBNowegoProcesu.uxPriority = proces.uxPriority
12:        end if
13:    end for
14: end if
15: Zarejestruj nowy proces(TCBNowegoProcesu)
```

Ten algorytm używa tej samej logiki priorytetowej co standardowy algorytm FreeRTOS, więc większość kodu nie wymaga zmian. Konieczna jest tylko jedna modyfikacja: zmiana funkcji *prvAddNewTaskToReadyList*, aby przy rejestracji priorytety procesów zostały przeliczone zgodnie z algorytmem DM.

Niestety, nie udało się zaimplementować tego algorytmu, więc nie będą przedstawione wyniki jego testów.

5.3.11 EDF

Alg. 8 demonstruje pseudokod algorytmu EDF.

Algorytm 8 Pseudokod algorytmu EDF

```
1: for all procesTCB in pxReadyTasksLists do
2:     procesTCB.Deadline = oblicz Deadline dla procesu(procesTCB)
3:     pxCurrentTCB.Deadline = oblicz Deadline dla procesu pxCurrentTCB
4:     if procesTCB.Deadline < pxCurrentTCB.Deadline then
5:         pxCurrentTCB=procesTCB
6:     end if
7: end for
```

Deadline z przykładu jest obliczany zgodnie z następującą formułą:

$$D(t) = T_{ready}(t) + T_{dead} \tag{4}$$

Gdzie: $D(t)$ - najgorszy czas osiągnięcia celu przez proces dla danej chwili; T_{ready} - czas, kiedy proces zostaje przypisany do listy *pxReadyTasksLists* w danej chwili; T_{dead} - względny (względem T_{ready}) najgorszy czas osiągnięcia celu przez proces.

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany został do wykonania proces zgodnie z algorytmem EDF.
2. Modyfikacja funkcji *vTaskSwitchContext*, aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej były przydzielane procesowi zgodnie z algorytmem EDF. Lub, jeżeli jest to skonfigurowane, wywoływanie planisty musi być wymuszone przy pojawieniu nowego procesu w liście *pxReadyTasksLists*.

5.3.12 LLF

Alg. 9 demonstruje pseudokod algorytmu LLF.

Algorytm 9 Pseudokod algorytmu LLF

```
1: for all procesTCB in pxReadyTasksLists do
2:     procesTCB.Laxity = Oblicz laxity dla procesu(procesTCB)
3:     pxCurrentTCB.Laxity = Oblicz laxity dla procesu(pxCurrentTCB.Laxity)
4:     if procesTCB.Laxity < pxCurrentTCB.Laxity then
5:         pxCurrentTCB=procesTCB
6:     end if
7: end for
```

Laxity jest obliczane zgodnie z następującą formułą:

$$L(t) = D(t) - (t + R(t)) \tag{5}$$

Gdzie: $L(t)$ - *laxity* dla danej chwili; $D(t)$ - najgorszy czas osiągnięcia celu przez proces, obliczony zgodnie z formułą (4); t - aktualny okres zegara systemu operacyjnego; $R(t)$ - czas do osiągnięcia celu, obliczony zgodnie z formułą (3).

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany został do wykonania proces zgodnie z algorytmem LLF.
2. Modyfikacja funkcji *vTaskSwitchContext*, aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej były przydzielane procesowi zgodnie z algorytmem LLF. Lub, jeżeli jest to skonfigurowane, wywoływanie planisty musi być wymuszone przy pojawieniu nowego procesu w liście *pxReadyTasksLists*.

5.3.13 DARTS

Alg. 10 demonstruje pseudokod algorytmu DARTS. Formuła obliczenia *score* z tego przykładu:

$$S(t) = \frac{R(t)}{D(t)L(t)} \tag{6}$$

Gdzie: $S(t)$ - wspomniany wyżej *score* dla danej chwili; $R(t)$ - czas do osiągnięcia celu przez

proces dla danej chwili, obliczony zgodnie z formułą (3); $D(t)$ - najgorszy czas osiągnięcia celu przez proces, obliczony zgodnie z formułą (4); $L(t)$ - *laxity* dla danej chwili, obliczona zgodnie z formułą (5).

Algorytm 10 Pseudokod algorytmu DARTS

```
1: for all procesTCB in pxReadyTasksLists do
2:     procesTCB.Score = Oblicz score(procesTCB)
3:     pxCurrentTCB.Score = Oblicz score(pxCurrentTCB)
4:     if procesTCB.Score > pxCurrentTCB.Score then
5:         pxCurrentTCB=procesTCB
6:     end if
7: end for
```

Potrzebne modyfikacje jądra FreeRTOS:

1. Modyfikacja funkcji *prvAddNewTaskToReadyList*, aby przed uruchomieniem nadzorcy wybrany został do wykonania proces zgodnie z algorytmem DARTS.
2. Modyfikacja funkcji *vTaskSwitchContext*, aby przy każdym wywołaniu planisty zasoby jednostki obliczeniowej były przydzielane procesowi zgodnie z algorytmem DARTS.
3. Modyfikacje dla dodania wywłaszczania: przy każdym pojawieniu nowego procesu w liście *pxReadyTasksLists* i przy każdej zmianie wartości zegara systemu operacyjnego.

5.4 Porównywanie działania algorytmów

Dla porównania zaimplementowanych algorytmów zostało wygenerowano 165 zestawów danych procesów z wartościami obciążenia od $0,246$ do $1,589$. Każdy zbiór zawierał 25 procesów, których metadane zostały wygenerowane losowo, używając rozkładów prawdopodobieństwa (zgodnie z opisem skryptu *generate.py*). Dane dla każdego testu były zbierane do momentu, gdy wartość zegara systemu operacyjnego osiągała 2500 SysTick. Po osiągnięciu tej wartości zegara, narzędzia pomiarowe przechodziły do analizy zebranych danych.

Każdy z zaimplementowanych algorytmów został przetestowany przy użyciu tego zbioru danych. Przetestowane także zostały dwie odmiany standardowego algorytmu FreeRTOS: bez wywłaszczania i współdzielenia, oraz z wywłaszczaniem i bez współdzielenia. Algorytm FreeRTOS'a wymaga wygenerowania priorytetów, więc do wspomnianych wyżej wygenerowanych danych dodano priorytety w zakresie od 1 do 255. Powtarzanie priorytetów było wyłączone.

Wyniki przeprowadzonych testów w wersji graficznej są zamieszczone na rys. 13. Po analizie stwierdzono, że w celu porównania rezultatów algorytmów należy wygładzić wyniki, aby pozbyć się niepotrzebnych w tym przypadku wahań. Wahania te będą omówione oddzielnie. Zgładzona wersja wyników w postaci graficznej jest pokazana na rys. 13. Dla wygładzania użyto algorytmu Simple Moving Average, który obliczał średnią z 5 punktów.

Analizując dane, można zanotować następujące fakty:

1. Wszystkie algorytmy z sukcesem radziły z zarządzaniem systemem do wartości obciążenia 0.496.

2. Pierwszym algorytmem, którego procesy zaczęły spóźniać się z osiągnięciem celu jest FCFS.
3. Najgorzej z wszystkich w zarządzaniu procesami były trzy algorytmy: FCFS, RR i DARTS. Pozostałe radziły z zadaniem w przybliżeniu z tym samym sukcesem.
4. Najlepiej z zadaniem, szczególnie przy obciążeniu większym niż 0.7, poradziły algorytmy SJF i SRTN.
5. Rożnica pomiędzy dwoma przetestowanymi odmianami standardowego algorytmu FreeRTOS jest znikoma, jak również i pomiędzy odmianami LLF i EDF.

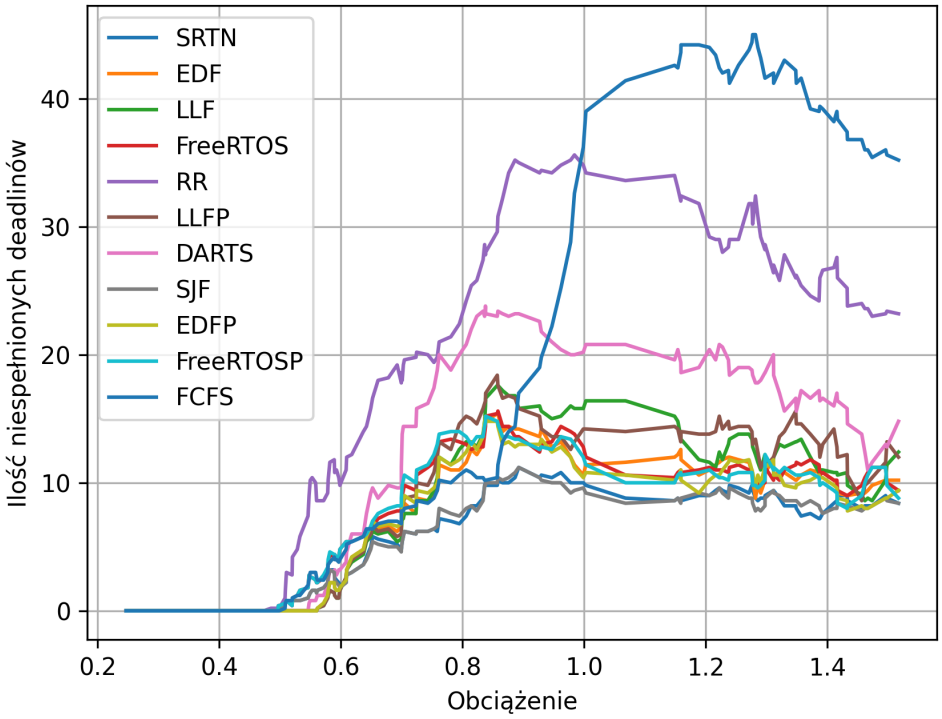
Fakt, że DARTS znalazł się wśród najgorszych wyników, był nieoczekiwany, biorąc pod uwagę skomplikowaność algorytmu.

Podczas analizy zauważono jedną niezgodność w wynikach. Patrząc na rys. 12a, można dostrzec, że w pewnym momencie ilość opóźnień dla każdego algorytmu zaczyna spadać, mimo ciągłego zwiększania obciążenia. Jest to trochę mylące, ponieważ, gdy algorytm przy pewnej wartości obciążenia systemu zaczyna wykazywać wzrost liczby niespełnionych celów w określonych terminach, to przy dalszym zwiększaniu obciążenia, oczekiwany jest dalszy wzrost liczby opóźnień (zakładając, że ilość procesów w systemie się nie zmienia, co jest prawdą w badanym przypadku). Dlaczego więc w tym przypadku wartość opóźnień się zmniejsza?

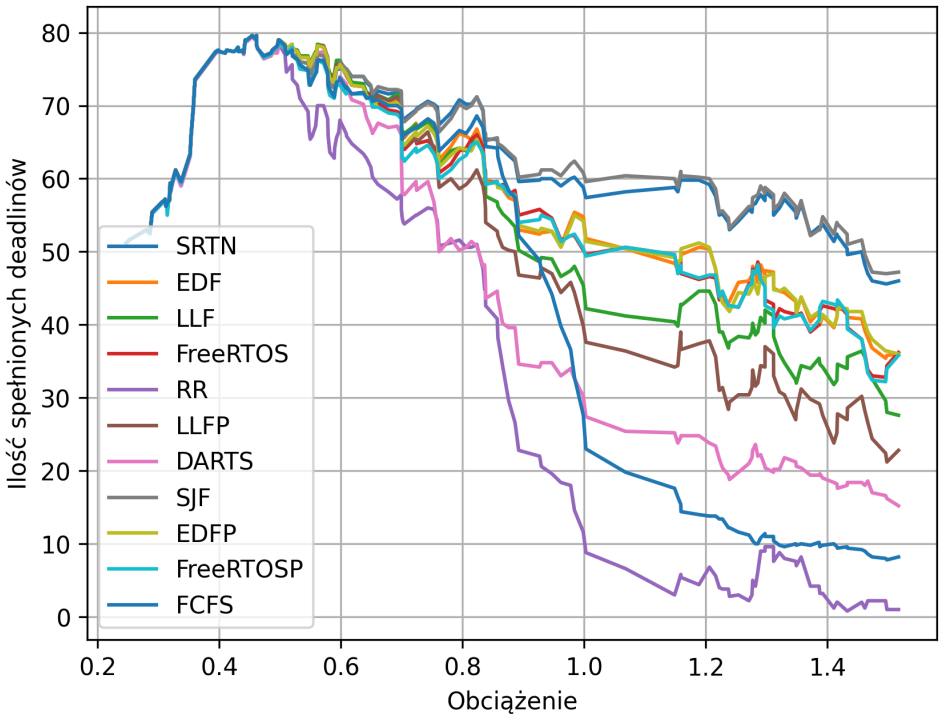
Aby odpowiedzieć na to pytanie, należy przyjrzeć się wykresowi rys. 12b, który pokazuje liczbę osiągniętych celów bez opóźnienia. Uwzględniając powyższe, jeśli liczba niespełnionych celów spada, to liczba spełnionych celów, przedstawiona na rys. 12b, powinna rosnąć. Jednak na rys. 12b, liczba spełnionych celów ciągle spada. Gdzie są wtedy te spełnione lub niespełnione cele?

Problem był ukryty w sposobie śledzenia wykonywania procesów i dokonywania pomiarów wybranym w ramach tej pracy. Skrypt *parse.py* wykrywa aktywność procesów na podstawie wykonywanych funkcji, które są odpowiednikami procesów w FreeRTOS i są wykonywane tylko wtedy, gdy nadzorca przydzieli im zasoby jednostki obliczeniowej. W przypadku zwiększenia obciążenia systemu, niektóre procesy mogą w ogóle nie dostawać zasobów jednostki obliczeniowej, co uniemożliwia nawet rozpoczęcie wykonywania.

Fakt, że niektóre procesy przy większym obciążeniu nie mogą rozpocząć swojego wykonywania, oraz sam sposób wykrywania aktywności procesów, prowadzą do wniosku, że sukcesy lub porażki w osiągnięciu celów nie zostały wykryte, ponieważ procesy te nawet nie zaczęły swoje wykonywanie. To z kolei spowodowało ich niewykrycie podczas pomiarów i analizy. Skoro te procesy nawet nie rozpoczęły swojego wykonywania, można je przepisać do grupy procesów, którym nie udało się osiągnąć swojego celu w wyznaczonych ograniczeniach czasowych. Taki stan można nazwać nasyceniem.

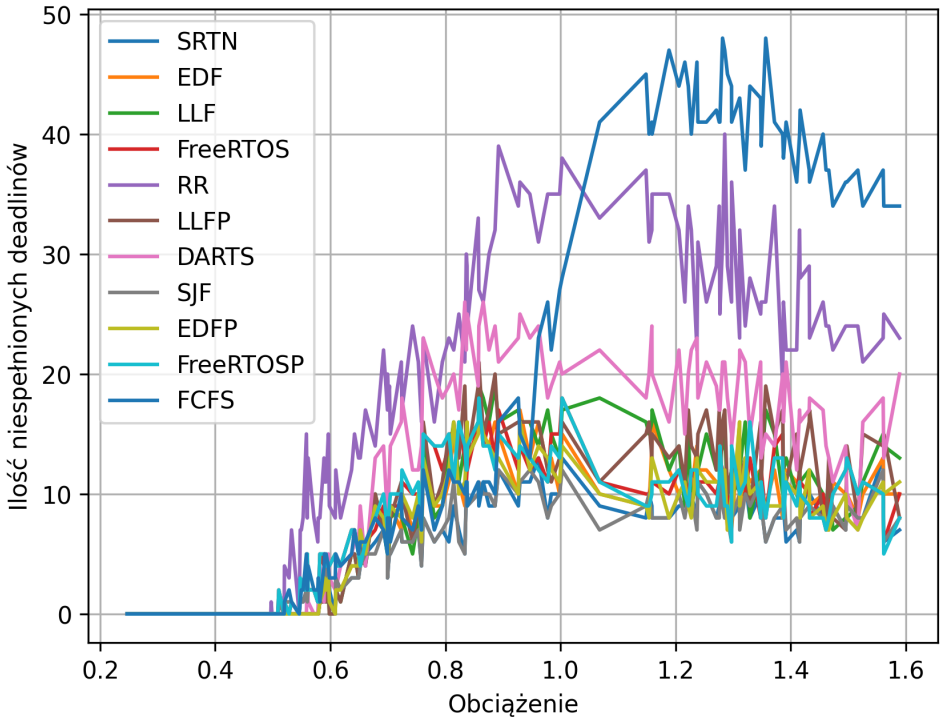


(a) Zależność liczby niespełnionych ograniczeń czasowych od obciążenia systemu

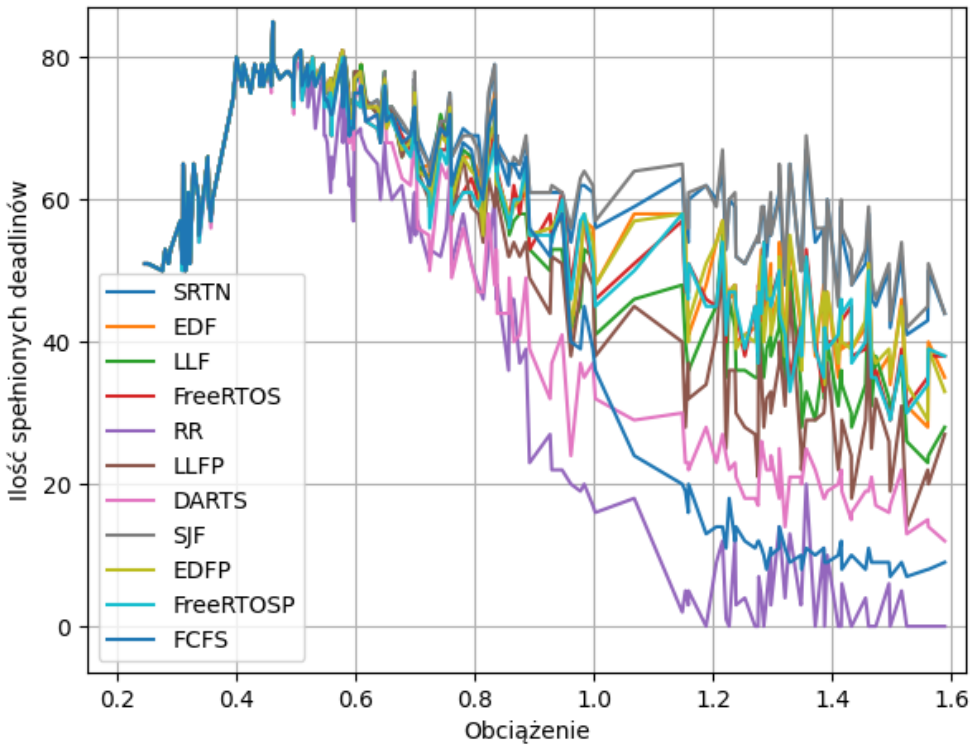


(b) Zależność liczby spełnionych ograniczeń czasowych od obciążenia systemu

Rysunek 12: Wyniki testów zgładzone, dla obciążenia i najgorszego czasu osiągnięcia celu (deadline na wykresach), dla każdego algorytmu (dodatek *P* oznacza odmianę algorytmu z dodaniem wywłaszczania)



(a) Zależność liczby niespełnionych ograniczeń czasowych od obciążenia systemu



(b) Zależność liczby spełnionych ograniczeń czasowych od obciążenia systemu

Rysunek 13: Wyniki testów nie zgładzone, dla obciążenia i najgorszego czasu osiągnięcia celu (deadline na wykresach), dla każdego algorytmu (dodatek *P* oznacza odmianę algorytmu z dodaniem wywłaszczania)

Patrząc na wyniki, można też zauważyć, że algorytmy SRTN i SJF zachowują się najlepiej w tym stanie.

Jeżeli chodzi o niewygładzone wyniki, głównym pytaniem jest, dlaczego te wyniki charakteryzują się tak dużymi wahaniami wartości spełnionych i niespełnionych celów w ramach ograniczeń czasowych? Odpowiedź jest ukryta w generacji danych dla testów.

Jak było wspomniane przy opisie skryptu *generate.py* (przykład 2), proces posiada listę metadanych, która nie ogranicza się tylko i wyłącznie obciążeniem systemu, które jest relacją pomiędzy czasem potrzebnym na osiągnięcie celu, okresem przejścia procesu do stanu gotowości i liczbą procesów zdefiniowaną statycznie (zgodnie z formułą eq. (1)). To właśnie te metadane, a dokładnie ich wzajemna relacja, stanowią drugi najważniejszy czynnik (obok używanego algorytmu do zarządzania zasobami jednostki obliczeniowej) wpływający na ostateczny wynik.

Fakt, że wszystkie metadane procesów były generowane losowo, uwzględniając tylko jedną zależność pomiędzy trzema zmiennymi, spowodował takie wahania w finalnych rezultatach.

Podsumowanie

W tej części pracy przedstawione zostaną wnioski i przemyślenia autora, które pojawiły się podczas realizacji poszczególnych etapów pracy.

Planowanie i zakres

Początkowo planowano przeanalizować dwie architektury oprogramowania systemowego: architekturę z wirtualizacją i bez wirtualizacji, uwzględniając różne realizacje tych architektur. Plan ten opierał się na wiedzy koncepcyjnej autora na temat tych architektur, dlatego nie było możliwe dokładne przewidzenie problemów oraz ich skutków. W wyniku tego zakres prac został zmniejszony do dwóch architektur w części teoretycznej oraz jednej architektury i jednej realizacji w części praktycznej.

W takim przypadku należało skonsultować się z osobami posiadającymi większą wiedzę na ten temat, aby lepiej przeanalizować i rozplanować tok pracy.

Algorytmy

W ramach tej pracy analizowane były głównie algorytmy jednordzeniowe, chociaż początkowo planowano także przeanalizować algorytmy wielordzeniowe. Większość z algorytmów jednordzeniowych jest już znana od końca dwudziestego wieku, i w tym zakresie już mało zostało do badań i usprawnień. Przykładem może być algorytm DARTS, który jest najnowszym z badanych algorytmów, jednak pokazuje rezultaty gorsze od algorytmów starszych.

Natomiast większym wyzwaniem jest dobór odpowiedniego algorytmu do konkretnego przypadku praktycznego oraz implementacja tego algorytmu w rzeczywistym systemie. Najwięcej trud-

ności pojawia się w systemach z kilkoma jednostkami obliczeniowymi, co nie zostało zbadane w ramach tej pracy.

Po analizie i porównaniu algorytmów autor nie widzi przyczyny dalszego rozwoju i optymalizacji algorytmów jednordzeniowych, chociaż autor nie jest także w stanie tego stwierdzić na pewno. Aby to potwierdzić, konieczne byłoby przeprowadzenie analizy matematycznej.

Potencjał dalszego rozwoju

W ramach tej pracy zaimplementowane i przeanalizowane zostały najpopularniejsze algorytmy jednordzeniowe. Głównym celem tego było zdobycie wiedzy praktycznej i teoretycznej na temat tych algorytmów. Jednak przeprowadzona analiza była niekompletna i może być kontynuowana. Lista przykładowych przypadków testowych dla dalszej analizy:

- Analiza zależności pomiędzy rozkładem średniego czasu potrzebnego na osiągnięcie celu a liczbą sukcesów i porażek w osiągnięciu celów przez procesy.
- Analiza zależności pomiędzy ilością procesów a ilością sukcesów i porażek w osiągnięciu celów przez te procesy.
- Analiza zależności pomiędzy rozkładem priorytetów procesów i ilością sukcesów i porażek w osiągnięciu celów przez procesy.
- Analiza możliwości konfiguracji algorytmów (np. algorytm RR posiada możliwość ustawienia kwantu czasu) i wpływu tej konfiguracji na wyniki wyjściowe.

Warto również zwrócić uwagę na optymalizacje doboru metadanych procesów, aby uniknąć wahań i niepewności w rezultatach wyjściowych, zaprezentowanych na rys. 13.

Natomiast analiza algorytmów w przypadku obecności kilku jednostek obliczeniowych lub hiperwizora jest tematem bardziej skomplikowanym, ponieważ oprócz algorytmów zarządzania zasobami jednostki obliczeniowej pojawiają się również problemy determinizmu związane z pamięcią podręczną jednostki obliczeniowej (ang. cache). Jednak największym wezwaniem w takich przypadkach są implementacja, pomiar i analiza, co wynika z obecności dodatkowej jednostki obliczeniowej lub warstwy abstrakcji w postaci hiperwizora.

Wykaz literatury

[1] Samuel O. Aletan. “An Overview of RISC Architecture”. W: *SAC ’92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing: technological challenges of the 1990’s* (kw. 1992).

[2] *Arm Holdings*. wikipedia.org. URL: `https://en.wikipedia.org/wiki/Arm_Holdings`.

[3] *Armv8-A Address Translation*. Wer. 1.1. ARM. Lip. 2024. URL: `https://developer.arm.com/documentation/100940/0101/`.

[4] Greg Wilson Christopher Svec Amy Brown. “The Architecture of Open Source Applications, Volume 2”. W: Lulu.com, 2012. Rozd. FreeRTOS. URL: https://aosabook.org/en/v2/freertos.html.

[5] Christoffer Dall. *Nested Virtualization on ARM*. Linaro. URL: https://www.youtube.com/watch?v=cuXyGkZRUKU.

[6] *Embedded Trace Macrocell Architecture Specification*. Wer. 3.5. ARM. Wrz. 2011. URL: https://developer.arm.com/documentation/ihi0014/q/.

[7] Robert P. Goldberg Gerald J. Popek. “DARTS: DynAmic Real-time Task Scheduling”. W: *7th Conference on Information and Knowledge Technology* (maj 2015). URL: https://ieeexplore.ieee.org/document/7288767.

[8] Robert P. Goldberg Gerald J. Popek. “Formal Requirements for Virtualizable Third Generation Architectures”. W: *Communications of ACM* (lip. 1974). URL: https://dl.acm.org/doi/10.1145/361011.361073.

[9] *GNU Compiler Collection*. wikipedia.org. URL: https://pl.wikipedia.org/wiki/GNU_Compiler_Collection.

[10] *Instruction set architecture*. wikipedia.org. URL: https://en.wikipedia.org/wiki/Instruction_set_architecture.

[11] *Introduction to the Armv8-M Architecture and its Programmers Model User Guide*. Wer. 1.1. ARM. Lip. 2023. URL: https://developer.arm.com/documentation/107656/0101/.

[12] *Learn the architecture - AArch64 Exception Model*. Wer. 1.3. Arm. Grud. 2022. URL: https://developer.arm.com/documentation/102412/0103.

[13] *Learn the architecture - Generic Interrupt Controller v3 and v4*. Wer. 3.2-01. ARM. Grud. 2021. URL: https://developer.arm.com/documentation/198123/0302.

[14] *Learn the architecture - Introducing the Arm architecture*. Wer. 2.1. Arm. Wrz. 2024. URL: https://developer.arm.com/documentation/102404/0201.

[15] *Memory management unit*. wikipedia.org. URL: https://en.wikipedia.org/wiki/Memory_management_unit.

[16] *Memory protection unit*. wikipedia.org. URL: https://en.wikipedia.org/wiki/Memory_protection_unit.

[17] R. C. Hansdah N. Saranya. “Dynamic Partitioning Based Scheduling of Real-Time Tasks in Multicore Processors”. W: *International Symposium on Object-Oriented Real-Time Distributed Computing* (kw. 2015). DOI: 10.1109/ISORC.2015.23. URL: https://ieeexplore.ieee.org/document/7153806.

[18] *Procesor*. wikipedia.org. URL: https://pl.wikipedia.org/wiki/Procesor.

[19] *Wieloprocesorowość asynchroniczna*. wikipedia.org. URL: https://pl.wikipedia.org/wiki/Wieloprocesorowo%C5%9B%C4%87_asynchroniczna.

[20] *Wieloprocesorowość symetryczna*. wikipedia.org. URL: https://pl.wikipedia.org/wiki/Wieloprocesorowo%C5%9B%C4%87_symetryczna.

Załącznik nr 1: System wbudowany

Skoro nie ma jednoznacznej definicji pojęcia ”system wbudowany”, należy określić definicję, która jest wykorzystywana w tej pracę.

System wbudowany jest systemem, to znaczy ma cechy systemu: jest złożony z więcej niż jednego elementu, jest funkcjonalnie niezależny od środowiska i posiada możliwość reagowania (tzn. odbierania, przetwarzania i udzielenia odpowiedzi) na bodźce zewnętrzne. ”Wbudowany”oznacza, że system jest częścią innego systemu fizycznie i/lub funkcjonalnie.

Natomiast przełożyć tę definicję na dziedzinę elektroniki i informatyki można w następujący sposób: system wbudowany — jest to system komputerowy składający się z jednostki obliczeniowej i modułów wejścia/wyjścia (tzn. może reagować na bodźce zewnętrzne), mający wszystkie narzędzia, w tym oprogramowanie, do pełnienia pewnej, zdefiniowanej funkcji (więc może być wyodrębniony) i będący częścią większego systemu.

Załącznik nr 2: Wirtualizacja

Wirtualizacja jest technologią wprowadzającą separację środowisk z oprogramowaniem poprzez tworzenie tzw. VM zarządzanych przez VMM (rys. 3). Rodzaje separacji używane w wirtualizacji to: separacja pamięciowa (poprzez translację pamięci), separacja sygnałowa (przekierowywanie przerwań) i separacja stanów (przełączanie kontekstu).

Celem separacji pamięciowej jest przypisanie do każdej VM przedziału adresów pamięci (kodu programu i urządzeń peryferyjnych, ponieważ rejestry tych urządzeń są zmapowane do pamięci; ang. memory-mapped peripherals) oraz wychwytywanie wszystkich prób manipulacji pamięcią, która nie jest przypisana do tej VM. W architekturach ARM tym zajmuje się MMU, który, począwszy od ARMv8, posiada dwa poziomy translacji: z poziomu aplikacji, tzw. VA (ang. Virtual Address, dosłownie adres wirtualny, pojęcie określające adresy pamięci przekazane aplikacji przez system operacyjny, nie są tą realne adresy w pamięci, tylko adresy w pewien sposób zmodyfikowane i przetwarzane na realne adresy podczas procesu translacji pamięci [3]), do poziomu systemu operacyjnego, tzw. IPA (ang. Intermediate Peripheral Address, pośredni realny adres, pojęcie ARM’owe określające adresy pamięci widziane przez system operacyjny znajdujący się wewnątrz VM, te adresy nazewane są ”pośrednio realnymi”, dlatego, że rzeczywiste adresy są ukrywane przez hiperwizor, z kolei system operacyjny uważa te adresy za rzeczywiste, skoro nie wie, że jest wewnątrz VM [3]), i do poziomu hiperwizora, tzw. PA (ang. Peripheral Address, dosłownie sprzętowy adres, tzn. realne adresy pamięci widziane przez hiperwizor [3]).

Celem separacji sygnałowej jest podział i przekierowywanie przerwań sprzętowych pomiędzy

VM. Każda VM ma przepisany podzbiór urządzeń peryferyjnych za pomocą separacji pamięci. Te urządzenia mogą generować przerwania, za przekierowywanie których jest odpowiedzialny hiperwizor i kontroler przerwań, w przypadku ARM'u to jest GIC.

Celem separacji stanów jest sprzętowy podział wykonywanych programów i przypisanie do nich odpowiednich poziomów uprawnień. Dokładniej to jest opisane w załączniku nr 3.

Przy czym, typowo hiperwizor jest zaprojektowany w taki sposób, że przy użyciu wszystkich tych rodzajów separacji aplikacja (lub system operacyjny), uruchomiana na tym hiperwizorze wcale nie musi wiedzieć, że jest uruchomiana nie wprost na rzeczywistym sprzęcie. Taki rodzaj wirtualizacji nazywany jest wirtualizacją natywną (ang. native virtualization, także ang. full virtualization, lub ang. hardware-accelerated virtualization).

W przypadku, gdy sprzęt nie posiada dedykowanych narzędzi do zapewnienia jednego z rodzajów separacji, można skorzystać z tzw. parawirtualizacji (ang. paravirtualization). W takim przypadku aplikacja, która jest uruchamiana na hiperwizorze, jest modyfikowana w sposób taki, aby wiedziała, że nie działa na realnym sprzęcie, a na wirtualnym, i była świadoma obecności hiperwizora. Przykładem może być separacja pamięciowa. Typowo próba otrzymania dostępu do zabronionej sekcji pamięci RAM jest wykrywana przez MMU. W przypadku nieobecności tego modułu aplikacja jest modyfikowana, co oznacza, że wszystkie instrukcje odczytujące dane z pamięci są podmieniane na zestaw instrukcji, który powoduje komunikat do hiperwizora w celu sprawdzenia dostępów.

Załącznik nr 3: Przełączanie stanu i kontekstu, definicja procesu

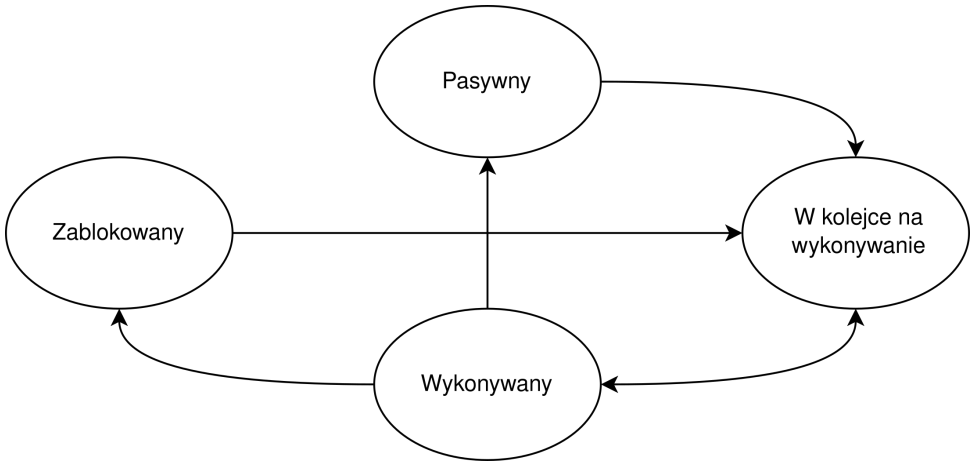
Na początku należy oddzielić pojęcie programu od pojęcia procesu. Program, w przypadku oprogramowania, to jest lista instrukcji do wykonania przechowywana w dowolnym miejscu. Program nie posiada, żadnego stanu, po prostu istnieje. Natomiast procesem jest program z przydzielonym stanem. Każdy stan procesu może zawierać dodatkowe dane, np. priorytet, zasoby, połączenia z innymi procesami i tak dalej. Na rys. 14 jest pokazany najbardziej popularne stany procesu oraz przejścia pomiędzy nimi.

Stan wykonywanego oprogramowania określany jest przez następujące elementy:

- Zawartości rejestrów ogólnego przeznaczenia: zawierają wyniki wykonywania ostatnich instrukcji;
- Rejestry wskazujące na adres następnej komendy (PC w architekturach ARM'owych), adres powrotu (LR w architekturach ARM'owych) i adres aktualnie używanego stosu (MSP lub PSP w architekturach ARM): określają "kierunek"(tzn. co będzie wykonane następne, gdzie wrócić po wykonaniu i tak dalej) wykonywanych instrukcji programu;
- Zawartość aktualnie używanego stosu: podobnie ja i w poprzednim punkcie, określa "kierunek"wykonywania instrukcji programu, przechowuje historie wykonywanych instrukcji pro-

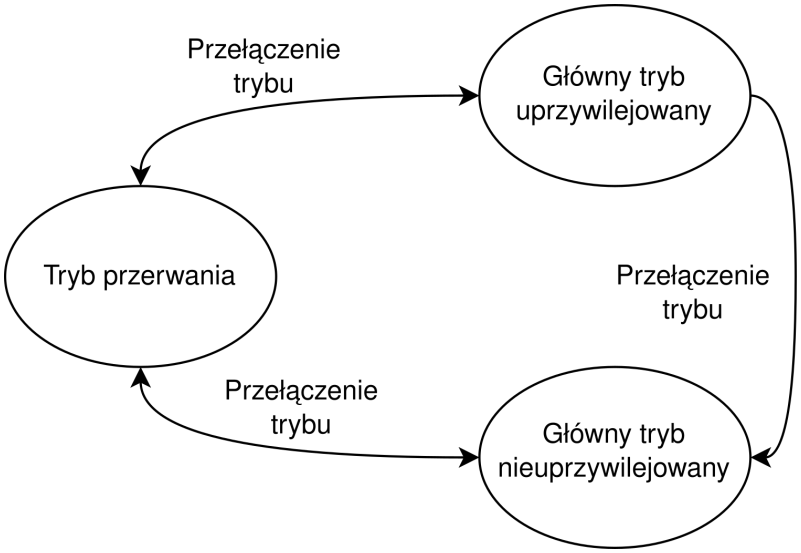
gramu (np. przy przerwaniu, do niego są zapisywane pewne rejestry, które po zakończeniu przerwania są odczytywane z powrotem, pozwalając na wznowienie wykonywanego programu);

- Rejestry systemowe (np. PSR i rejestry masek przerwań w architekturach ARM'owych): określają między innymi tryb jednostki obliczeniowej, w którym program jest wykonywany, co z olei określa poziom dostępu do zasobów sprzętowych.



Rysunek 14: Ogólny graf stanów procesu

Z powyższego wynika, że stan wykonywanego oprogramowania to zbiór pewnych wartości przechowywanych w rejestrach jednostki obliczeniowej, co oznacza, że przełączenie stanów odbywa się sprzętowo, przez jednostkę obliczeniową, ponieważ oprogramowanie nie ma dostępu do zapisu ani odczytu wszystkich rejestrów jednostki obliczeniowej.



Rysunek 15: Tryby procesora zbudowanego na podstawie architektury ARMv8M [11]

Z kolei przełączenie stanów to proces, w którym stan aktualnie wykonywanego oprogramowania jest zapisywany w stos przynależącego do tego oprogramowania i podmieniany na stan przynależący do innego oprogramowania, który jest odczytywany ze stosu uruchamianego opro-

gramowania. Występuje to podczas zmiany trybu jednostki obliczeniowej, ponieważ każdy tryb jednostki obliczeniowej ma własny stos.

Na rys. 15 pokazane są tryby jednostki obliczeniowej zbudowanej na podstawie architektury ARMv8M. Taka jednostka posiada trzy tryby z dwoma poziomami dostępu: tryb główny uprzywilejowany, główny nieuprzywilejowany i tryb przerwania. Każdy z tych trybów ma swój stos, w którym jest przechowywany stan aktualnie wykonywanego oprogramowania. Adres każdego ze stosów jest przechowywany w rejestrach MSP i PSP [11].

Natomiast kontekst programu to pojęcie zdefiniowane przez poszczególną implementację oprogramowania systemowego (system operacyjny bądź hiperwizor) i zawiera w sobie opisany wyżej stan programu oraz dodatkowe dane, interpretowane przez oprogramowanie systemowe (np. czas jednostki obliczeniowej użyty przez ten program, lub priorytet), zwane także metadanymi.

Stąd przełączenie kontekstu oznacza, że to nie jednostka obliczeniowa jest odpowiedzialna za wybór i uruchomienie programu, lecz oprogramowanie systemowe.

Na rys. 6 pokazano przykładowy przebieg takiego przełączenia dla architektury ARMv8M. Przełączenie kontekstu następuje podczas wykonywania kodu systemu operacyjnego, tzn. gdy jednostka obliczeniowa znajduje się w trybie głównym uprzywilejowanym (rys. 15), co oznacza, że przed przełączeniem kontekstu następuje przełączenie stanów, podczas którego jednostka obliczeniowa przełącza się z wykonywania kodu aktualnego programu na wykonywanie kodu systemu operacyjnego (czyli przechodzi z trybu głównego nieuprzywilejowanego do trybu głównego uprzywilejowanego, rys. 15). Natomiast po przełączeniu kontekstu następuje przełączenie stanów, podczas którego jednostka obliczeniowa przełącza się z wykonywania kodu systemu operacyjnego na wykonywanie kodu programu zaproponowanego przez system operacyjny podczas przełączenia kontekstu programu (tzn. przechodzi z trybu głównego uprzywilejowanego do trybu głównego nieuprzywilejowanego).

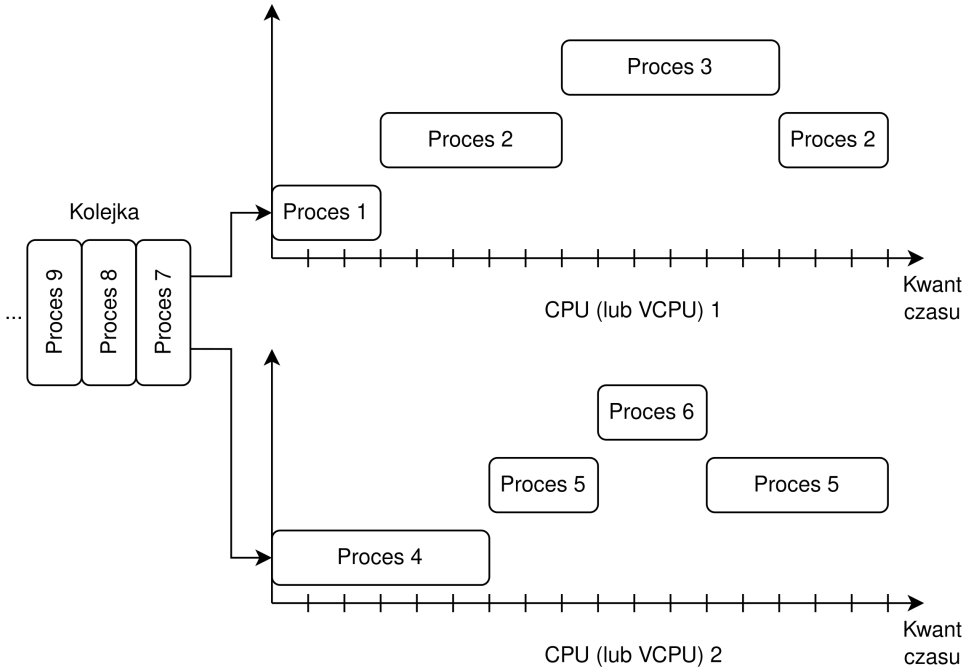
Załącznik nr 4: Zasób jednostki obliczeniowej

Jednostka obliczeniowa jest systemem, który przyjmuje instrukcje, wykonuje je i podaje wynik. Podstawową wielkością w tym przypadku będzie czas wykonania jednej instrukcji (w przypadku architektury RISC (ang. Reduced Instruction Set Computing; typ architektury mikroprocesorowego układu oblczeniowego o ograniczonej funkcjanolności do potrzebnego minimum [1]) większość instrukcji są wykonywane w przedziale jednego okresu zegara zasilającego jednostkę obliczeniową).

Każdy program wykonywany przez jednostkę obliczeniową jest listą instrukcji, i każdemu programu zależy, aby ta lista instrukcji była wykonana jak najszybciej, czyli idealnie, jakby jedna jednostka obliczeniowa wykonywała tylko jeden program. Z tego wynika, że programy walczą o czas, na który im pozwolono używać jednostki obliczeniową do wykonania swoich instrukcji. Fi-

nalnie zasobem jednostki obliczeniowej jest czas.

Czas ten można podzielić pomiędzy programy w różny sposób, za to odpowiada nadzorca w oprogramowaniu systemowym.



Rysunek 16: Współdzielenie zasobów jednostek obliczeniowych przez wiele procesów w technologii AMP

W przypadku, gdy jest kilka jednostek obliczeniowych dostępnych dla wszystkich programów, zarządzanie ich zasobami można zorganizować na dwa sposoby: przydzielać każdemu programowi czas oddzielnej jednostki obliczeniowej (techniki AMP) lub połączyć zasoby wszystkich jednostek obliczeniowych i przydzielać jednemu programowi pewną część tych zasobów(techniki SMP).

W przypadku wirtualizacji programom udostępniane są VCPU (ang. Virtual CPU, wirtualna jednostka obliczeniowa). Są to w pewnym sensie przerobione przez hiperwizor zasoby realnej jednostki obliczeniowej.

Na rys. 16 przedstawiono przykład współdzielenia zasobów jednostek obliczeniowych przez wiele procesów w technologii AMP. Każdemu procesowi przypisana jest pewna część zasobów obliczeniowych, reprezentowana przez liczbę kwantów czasu (ang. time slice; najmniejsza możliwa ilość czasu jednostki obliczeniowej, którą można przypisać procesu w niektórych realizacja oprogramowania systemowego).

Stąd można zrobić trzy wnioski:

1. Zasobem jednostki obliczeniowej jest jej czas;
2. Czas może być przypisywany bezpośrednio lub podzielony przez oprogramowanie systemowe na pewne minimalne odcinki: kwanty czasu;
3. W przypadku obecności wielu jednostek obliczeniowych, zarządzanie ich zasobami odbywa

się zgodnie z technikami AMP lub SMP.