

POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)

SPECJALNOŚĆ: Systemy informatyczne w automatyce (ASI)

**PRACA DYPLOMOWA
INŻYNIERSKA**

Planista wykonywania zadań dla systemów
mikroprocesorowych czasu rzeczywistego

The scheduler of code execution for real-time
microprocessor systems

AUTOR:
Aleksandra Mika

PROWADZĄCY PRACĘ:
dr inż. Jarosław Pempera

OCENA PRACY:

Spis treści

1. Cel i zakres pracy	3
1.1. Wstęp	3
1.2. Cel pracy	3
1.3. Zakres pracy	4
2. Systemy czasu rzeczywistego	5
2.1. Klasyfikacja	5
2.1.1. Jądro	6
2.1.2. Pseudojądro	6
2.2. Przykładowe systemy operacyjne	8
2.2.1. FreeRTOS	8
2.2.2. Micrium MicroC/OS-II	9
2.3. Czas systemowy	9
2.4. Szeregowanie zadań	10
2.4.1. Możliwość utworzenia harmonogramu	10
2.4.2. Zadania o statycznym priorytecie	11
2.4.3. Zadania o dynamicznym priorytecie	12
2.4.4. Szeregowanie zadań zależnych	12
3. Projekt systemu	15
3.1. Program główny	15
3.2. Interfejs użytkownika	15
3.3. Jądro	17
3.4. Pamięć	17
3.5. Przerwania zegarowe, czas systemowy	17
3.6. Planista	18
3.7. Alokacja pamięci	19
3.8. Synchronizacja międzyprocesowa	19
3.9. Port na system Linux x86	20
3.9.1. Inicjalizacja sprzętowa	20
3.9.2. Kontekst jądra	20
3.9.3. Włączenie i wyłączenie przerw	21
3.9.4. Inicjalizacja i uruchomienie timera	21
4. Planista	23
4.1. Interfejs	23
4.2. Algorytmy szeregujące	23
4.2.1. Szeregowanie częstotliwościowe (RM)	24
4.2.2. Szeregowanie terminowe (EDF)	24

5. Rezultaty	27
5.1. Wizualizacja	27
5.2. Testy	27
5.2.1. Test 1. Niewielkie obciążenie procesora	27
5.2.2. Test 2. Znaczne obciążenie procesora	27
5.2.3. Test 3. Zadania harmoniczne, pełne wykorzystanie procesora	29
5.2.4. Test 4. Przeciążenie procesora	30
6. Implementacja	35
6.1. Algorytm wizualizacji	35
6.2. Wizualizacja przebiegu zadań	37
6.3. Implementacja kluczowych elementu systemu	40
6.3.1. Program główny	40
6.3.2. Pamięć systemowa	40
6.3.3. Jądro systemu	41
6.3.4. Czas systemowy	42
6.3.5. Planista	42
6.3.6. Alokacja pamięci	45
6.4. Implementacja portu symulacyjnego na system Linux	45
6.4.1. Inicjalizacja sprzętowa	45
6.4.2. Kontekst jądra	46
6.4.3. Przerwania zegarowe	47
7. Podsumowanie	49
7.1. Plany rozwoju	49
7.2. Wnioski	49
Bibilografia	51
Spis rysunków	53
Spis algorytmów	54
Spis tabel	54
Spis kodów	54

Rozdział 1.

Cel i zakres pracy

1.1. Wstęp

Systemy komputerowe pełnią ważną rolę w naszym społeczeństwie. Wiele z nich nadzoruje procesy, gdzie wymagana jest duża precyzja, szybkość reakcji i niezawodność, na przykład system nadzorujący elektrownię albo komputer pokładowy w samolocie. Czas odpowiedzi systemu nie może przekroczyć określonego dla niego terminu wykonania, wynikającego z fizycznej natury kontrolowanego zjawiska.

Dla przykładu, gdyby wskazania z akcelerometrów w samolocie nie zostały w odpowiednim czasie przetworzone, groziłoby to nieprawidłowym obliczeniem położenia i stanu maszyny, a w konsekwencji nawet narażeniem życia ludzi.

Systemy, w których kluczowe jest nie tylko uzyskanie wyniku, ale też uzyskanie go w określonym czasie, określane są systemami czasu rzeczywistego (ang. *real-time operating system*, RTOS). Mogą one wykonywać rozmaite zadania, zróżnicowane czasem trwania, istotnością, terminem czy znajomością czasu pojawienia się zadania. Bardzo istotną kwestią jest ustalenie takiej kolejności wykonania zadań, aby każde z nich dotrzymało swojego terminu zakończenia (ang. *deadline*). Funkcjonalność ta realizowana jest przez część systemu zwaną planistą (ang. *scheduler*) [LO12]. Może on ustalać harmonogram zadań przed uruchomieniem systemu albo już podczas działania, zgodnie z zadaną polityką szeregowania.

Każdy system pracuje w określonym środowisku, korzysta z różnych cech zjawisk fizycznych i może służyć różnemu celowi. Dlatego należy dobrać jego charakterystykę, tak aby odpowiadała konkretnym potrzebom. Optymalizacja działania systemu szczególnie jest istotna w systemach mikroprocesorowych, gdzie minimalizuje się (m.in. ze względu na koszt w masowej produkcji) zarówno moc obliczeniową jak i pamięć operacyjną. Taki system musi być zaprojektowany, by jednocześnie spełniać wymagane od niego zadania, a zarazem aby koszt wytworzenia był jak najmniejszy.

1.2. Cel pracy

Głównym celem pracy było zaprojektowanie oraz zaimplementowanie modułu planisty dla systemów czasu rzeczywistego opartego na mikrokontrolerach. Cel pracy został podzielony na następujące etapy:

1. Zapoznanie się z istniejącymi systemami czasu rzeczywistego.

2. Opracowanie algorytmów obsługi podstawowych funkcji systemu czasu rzeczywistego.
3. Opracowanie algorytmów planowania wykonania zadań.
4. Implementacja wyżej wymienionych algorytmów.
5. Przeprowadzenie badań eksperymentalnych algorytmów planowania.

Do celu badań stworzono mikrosystem czasu rzeczywistego oraz jego port symulacyjny na system Linux. System można przenosić na inne platformy, wymaga to implementacji poleceń specyficznych dla danej architektury procesora (np. mechanizmu zablokowania przerwania).

Zaimplementowano i zbadano działanie różnych algorytmów szeregujących zadania w systemie czasu rzeczywistego, sprawdzono zachowanie się takiego systemu w przypadku różnego obciążenia zadaniami, od niewielkiego wykorzystania procesora do przeciążenia zadaniami.

Oprócz tego, stworzono program wizualizujący przebieg zadań w systemie, ułatwiający analizę danej polityki szeregowania.

1.3. Zakres pracy

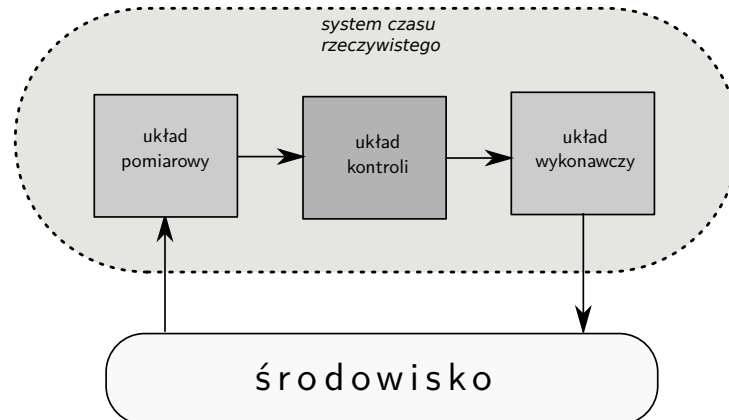
W rozdziale 2. przedstawiono ogólne cechy systemów czasu rzeczywistego, ich podział, istniejące systemy, niektóre negatywne zjawiska zachodzące w systemie, jak również teoretyczne podstawy szeregowania zadań oraz algorytmy. Rozdział 3. zawiera omówienie projektu i strukturę systemu stworzonego do badań polityk szeregowania. W rozdziale 4. bardziej szczegółowo omówiono planistę w stworzonym systemie oraz zaimplementowane polityki szeregowania. Rozdział 5. zawiera wyniki badań, przebieg pracy systemu z różnymi politykami szeregowania zadań, jak również działanie przy różnym stopniu obciążania zadaniami. W rozdziale 6. przedstawiono implementację programów, w tym algorytm działania programu wizualizującego oraz implementację wybranych poleceń systemu. Rozdział 7. zawiera podsumowanie wyników badań oraz wnioski.

Rozdział 2.

Systemy czasu rzeczywistego

System czasu rzeczywistego jest to system, którego poprawność logiczna bazuje zarówno na poprawności otrzymywanych rezultatów, jak i na spełnieniu terminów ich wykonania [LO12]. W przypadku niedotrzymania terminu, system traci w mniejszym lub większym stopniu swoją funkcjonalność.

Systemy czasu rzeczywistego powinny dostarczyć trzy podstawowe funkcjonalności: planowanie, zarządzanie zadaniami oraz mechanizmy komunikacji i synchronizacji między procesami. Systemy komputerowe, nie tylko czasu rzeczywistego, aby być użyteczne, muszą wymieniać informację ze światem zewnętrznym [Kop11]. Na rysunku 2.1. przedstawiono schemat interakcji w systemach czasu rzeczywistego. Przy projekcie systemu powinien być zapewniony interfejs do komunikacji z urządzeniami wyjścia/wejścia, jak również koordynacja wykonywania zadań w systemie.



Rysunek 2.1. Schemat interakcji systemu czasu rzeczywistego

W ogólności, system czasu rzeczywistego ma dostarczać przewidywalną i stabilną platformę, o małym narzucie czasowym, powinien też umożliwiać wielozadaniowość oraz współdzielenie zasobów [SR04].

2.1. Klasyfikacja

Ze względu na konsekwencje, które wynikają z niedotrzymania terminów zadań, można wyróżnić systemy czasu rzeczywistego [LO12]:

- o łagodne (ang. *soft real-time*) – wydajność systemu jest obniżona, ale nie prowadzi do katastrofy poprzez niedotrzymanie terminu zadań,

- rygorystyczne (ang. *hard real-time*) – nieterminowość nawet jednego zadania, może prowadzić do całkowitej niewydolności systemu.

Ze względu na mocniejsze ograniczenia w systemach rygorystycznych, algorytmy szeregowania w niniejszej pracy będą przedstawiane dla tego typu systemu.

Jeszcze innym kryterium podziału systemów może być źródło wywoływania akcji w systemie. Można wyróżnić architekturę [Kop11]:

- wyzwalaną czasowo (ang. *time triggered*) – każda akcja jest inicjowana poprzez cykliczne przerwanie zegarowe,
- wyzwalaną zdarzeniowo (ang. *event triggered*) – komunikacja oraz przetwarzanie następuje w momencie przyścia zdarzenia (innego niż przerwanie zegarowe), taki system wymaga dynamicznej polityki szeregowania, w momencie przyścia nowego zdarzenia.

2.1.1. Jądro

Jądro (ang. *kernel*) systemu jest kluczową częścią systemu, odpowiedzialną za całe działanie. Dostarcza ono narzędzi do zarządzania czasem procesora i zadaniami w systemie. Kluczową rolą dostarczaną przez jądro jest przełączanie kontekstu jądra [Lab02].

Kontekst jądra jest to stan systemu, wraz ze stosem wywołań funkcji, stertą zmiennych. Przełączanie kontekstu pozwala na uruchomienie wielu zadań w systemie, przerwanie wykonywania obecnego zadania, obsłużenie przerwania i przywrócenie wykonywania zadania.

Czas na przełączenie kontekstu jest największym czynnikiem wpływającym na czas odpowiedzi systemu. W dobrze zaprojektowanych systemach narzut jądra wynosi $2\% \div 5\%$ [Lab02]. Dlatego też należy minimalizować liczbę przełączeń kontekstu oraz zapisywać jak najmniejszą ilość informacji by przywrócić zadanie po wywłaszczeniu. Zwykle w kontekście należy umieścić informację o rejestrach na których pracuje system, wskaźnik stosu oraz obrazy możliwych odwzorowanych w pamięci peryferiów [LO12]. Zmiana kontekstu musi odbywać się z wyłączonymi przerwaniem, aby zachować spójność danych.

W ramach jądra często wydziela się podmoduły: planisty, dyspozytora oraz mechanizmy komunikacji i synchronizacji między zadaniami [Lab02]. Planista ma za zadanie wyznaczenie zadania, które ma być w danym momencie wykonywane na procesorze. Dyspozytor obsługuje zmianę zadań i przełączanie kontekstu, a mechanizmy komunikacji i synchronizacji pozwalają na utrzymanie spójności danych oraz współpracę zadań między sobą.

Większość jąder czasu rzeczywistego bazuje na priorytetach [Lab02], każdemu zadaniu przydziela się priorytet w zależności od istotności zadania. Jądra mogą być wywłaszczające (ang. *preemptive*) lub niewywłaszczające (ang. *non-preemptive*). Jądro wywłaszczające pozwala na przerwanie wykonywania obecnego zadania i zastąpienie go innym, w przypadku jądra niewywłaszczającego – zadanie wykonywane jest, aż do momentu dobrowolnego oddania procesora. Jądro niewywłaszczające zwane jest też kooperatywnym.

2.1.2. Pseudojądro

Duże systemy operacyjne czasu rzeczywistego zapewniają zarówno podstawowe funkcje jądra, ale też mogą dostarczać usługi wejścia/wyjścia, interfejs użytkownika, zapewnić ochronę pamięci oraz wspomagać zarządzanie systemem plików. Często jednak tak rozbudowany system może się okazać zbyt zasobochłonny, a część funkcjonalności zbędne.

Najczęściej pełne systemy operacyjne są wykorzystywane, gdy ograniczenie czasowe nie jest rygorystyczne [LO12].

W niskokosztowych systemach wbudowanych, nie ma potrzeby korzystania ze wszystkich funkcjonalności, można zaimplementować pseudojądro, bez właściwego systemu operacyjnego oraz bez jądra. Wiąże się to jednak z trudnością w utrzymywaniu i rozwijaniu takiego oprogramowania. [LO12].

Wśród pseudojąder można wyróżnić następujące typy [LO12]:

- **Pętla odpytywań** (ang. *polled loop*).

Polega on na ciągłym sprawdzaniu czy dane zewnętrzne zdarzenia nastąpiły, czy zostały odebrane dane przez układ komunikacji, czy nastąpiła zmiana stanu bitu, czy zewnętrzny sterownik ustawił flagę gotowości. Jeżeli tak, wykonywany jest odpowiedni ciąg instrukcji, w przeciwnym przypadku dalej sprawdzana jest dany warunek. Wariantem tego systemu jest odpytywanie z opóźnieniem po wykonaniu instrukcji (np. opóźnienie wynikające z czasu potrzebnego na usunięcie efektu drgania zestyków). Często takie pętle są wykorzystywane w systemach sterowanych przerwami jako zadanie tła, lub jako pojedyncze zadanie w kodzie cyklicznym. Wadą jest to, że nie mogą one obsługiwać złożonych systemów, a odpytywanie marnuje czas procesora.

- **Kod cykliczny** (ang. *cyclic code*).

Kod cykliczny polega na wykonywaniu w nieskończonej pętli kolejnych instrukcji (zadań). Stosuje się go jedynie w najprostszych systemach czasu rzeczywistego, ponieważ stwarza trudności w rozdziale zadań oraz wydłuża czas odpowiedzi systemu [LO12].

- **Skończona maszyna stanów** (ang. *finite state machine*).

System oparty na tym algorytmie, przechodzi przez szereg stanów podczas swojego działania. W zależności od stanu systemu wykonywane są odpowiednie działania. Pozwala to też na podział pojedynczego zadania na wiele mniejszych segmentów i umożliwia tymczasowe zawieszenie zadania przed jego ukończeniem bez utraty danych krytycznych. Taki schemat dobrze sprawdza się wraz z kodem cyklicznym, kiedy zadania są zbyt długie do wykonania.

- **Koprocesy** (ang. *corutines*).

Schemat koprocesów wprowadza dodatkowe usprawnienie do schematu maszyny stanów. Zakłada się, że zadanie wykonuje się do ukończenia swojej kolejnej fazy, pozwala to na zrównoleglenie wykonywania zadań. Zadanie po ukończeniu kolejnej fazy programu woła planistę, który przechowuje stany zadań i wybiera rotacyjnie zadanie do wykonania. Komunikacja między zadaniami odbywa się poprzez zmienne globalne. Dane dla zadania, które mają być zachowane pomiędzy wywołaniami, również muszą być zachowywane w zmiennych globalnych.

- **System wyłącznie przerwaniowy** (ang. *interrupt-only system*).

W takim schemacie program główny jest pustą, nieskończoną pętlą. Zadania są uruchamiane poprzez obsługę przerw (ang. *interrupt handler*).

System w zależności od architektury, może wspierać jedno- lub wielopoziomowe przerwania. W przypadku, gdy system wspiera wielopoziomowe przerwania, to sprzęt

wykona rozdział kolejności zadań. Gdy dostępne jest tylko jeden poziom przerwań, to obsługa przerwania musi przeczytać rejestr przerwań i wybrać, które nastąpiły. Wewnątrz zmiany kontekstu jądra, przerwania są wyłączone. Przebieg przerwania polega na zapisaniu kontekstu, wykonaniu obsługi przerwania i przywróceniu kontekstu.

- o **System dwuplanowy** (ang. *foreground/background system*).

Są usprawnieniem działania systemów wyłącznie przerwaniowych. W nieskończonej pętli wykonywane jest zadanie tła – niskopriorytetowe, w pełni wywłaszczalne zadanie, bez określonego terminu zakończenia. W przypadku pojawienia się przerwania, zadanie tła jest wywłaszczane i uruchamiane jest odpowiednie zadanie pierwszoplanowe.

- o **Wywłaszczający system priorytetowy** (ang. *preemptive priority system*).

Na procesorze zawsze wykonywane jest zadanie o najwyższym priorytecie. Zadanie o wyższym priorytecie, gdy nadejdzie, przerywa wykonywanie zadania niżej priorytetowego i rozpoczyna swoje wykonywanie. Wszystkie zadania powinny być w pełni wywłaszczalne – w dowolnym momencie może nastąpić przerwanie i wywłaszczenie. Pozwala to na uzyskanie (pod pewnymi warunkami) optymalnego harmonogramu [Buz97]. Jest on też dlatego najczęściej stosowanym mechanizmem zarządzania zadaniami w systemach wbudowanych, w oparciu o niego działa jądro wywłaszczające [LO12]. Dla systemu wywłaszczającego priorytetowego można wyznaczyć, różne algorytmy szeregowania i przyznawania priorytetów (patrz: 2.4.).

2.2. Przykładowe systemy operacyjne

Wśród istniejących systemów czasu rzeczywistego, popularnie używane są FreeRTOS oraz Micrium MicroC/OS-II [Bro04]. Obydwa są systemami wywłaszczającymi priorytetowymi, obydwa zostały napisane w języku C z dodatkiem kodu assemblera (do poleceń specyficznych dla danej architektury procesora). Te systemy pisane tak, aby płynnie działać na mikrokontrolerach. Mogą być one przenoszone na różne platformy pod warunkiem, że będzie dostarczony wskaźnik stosu oraz jest możliwy dostęp do rejestrów procesora.

Każdy z nich zapewnia mechanizmy komunikacji i synchronizacji między zadaniami, takie jak: semaforey, flagi zdarzeniowe, kolejki wiadomości, itp. Zadania w tych systemach pisze się albo w nieskończonej pętli, albo muszą usunąć się po wykonaniu. Każde zadanie ma przydzielony swój stos

2.2.1. FreeRTOS

System FreeRTOS jest dostępny na zmodyfikowanej licencji GPL. Planista w tym systemie jest deterministyczny, przewidywalność uzyskano poprzez statyczny przydział priorytetów zadaniami. Oprócz tego, planista może być skonfigurowany na dwa tryby pracy: zarówno na pracę wywłaszczającą jak i kooperatywną (bez wywłaszczeń).

Kiedy procesor nie ma przydzielonego zadania, uruchamiane jest zadanie bezczynności (ang. *idle task*). Jest to zadanie o najniższym priorytecie, równym zero. Dowolna ilość zadań może mieć ten sam priorytet, w zależności od konfiguracji mogą być wykonywane rotacyjnie co określony kwant czasu lub dopiero po zakończeniu zadania.

W tym systemie, przerwanie zegarowe zwiększa zmienną liczącą tiki – możliwy jest też tryb beztikowy, pozwalający na oszczędzanie energii. Kiedy obecny tik odpowiada tikowi, w którym należy obudzić zadanie oraz priorytet budzonego zadania jest wyższy niż obecnie wykonywanego, to następuje zmiana kontekstu jądra.

Przywracanie kontekstu odbywa się następująco: zadanie które ma być wznowione, wcześniej miało rejestry procesora na odpowiednim stosie zadań. W momencie przywracania jądro uzyskuje wskaźnik na stos i zapisuje zawartość tych rejestrów w rejestrach procesora. [RTE14]

2.2.2. Micrium MicroC/OS-II

System Micrium MicroC/OS-II jest systemem komercyjnym, dostępnym bez opłat dla uczelni i uniwersytetów.

Priorytety są przydzielane zadaniom przez użytkownika, z zastrzeżeniem, że nie wolno przydzielić dwóm zadaniom tego samego priorytetu. Istnieje również możliwość zmiany priorytetu zadania podczas działania systemu.

Podczas obsługi tików zegarowych zwiększana jest zmienna licząca liczbę tików od uruchomienia systemu. Po przekroczeniu wartości maksymalnej, wartość ta jest zerowana. Potem, dla każdego zadania zmniejszana jest liczba tików, które musi jeszcze czekać, jeżeli wyniesie zero, to zadanie jest gotowe do uruchomienia. Daje to czas obsługi przerwania proporcjonalny do liczby zadań. [Lab02]

2.3. Czas systemowy

Czas systemowy jest najczęściej mierzony w tikach, czyli w liczbie przerwania zegarowego pojawiającego się co określoną jednostkę czasu. Zazwyczaj przyjmuje się wartości tików co $10 \div 200$ ms [Lab02]. Układ służący do odmierzenia czasu nazywany jest timerem lub zegarem przerwań. Nie są one jednak idealne, możliwe są błędy pracy takie jak nierównomierność (ang. *jitter*) i dryf (ang. *drift*). Nierównomierność ma charakter losowy, natomiast dryf wynika ze skumulowania opóźnień uruchomienia zadań (np. poprzez niezerowy czas przełączania kontekstu). Dryf może spowodować, że po dłuższym czasie pracy systemu, odmierzany czas będzie niepoprawny im dłużej system działa.

W literaturze [Kop11],[Lab02] wskazuje się na możliwość korekcji tików. Znając dryf timera, może być ona wykonana na dwa sposoby: albo poprzez korekcję stanu albo korekcję częstotliwości. Korekcja stanu jest prostsza do implementacji, ale ma tę wadę, że generuje nieciągłość w podstawie czasu. Korekcja częstotliwości z uwzględnieniem dryfu pozwala ominąć tę wadę. Implementuje się ją albo programowo, dzieląc tiki na mikrotiki i makrotiki i zmieniając liczbę mikrotików w makrotiku, albo też sprzętowo, zmieniając napięcie kryształu oscylatora

Dodatkowo, ze względu na ograniczony rozmiar zmiennej liczącej tiki, można zaimplementować mechanizm „zawijania” tików [Lab02] – po przekroczeniu maksymalnej wartości licznika tików, tiki są liczone znów od zera. Takie podejście wymaga przemyślanego algorytmu planującego zadania, aby zmiana czasu systemowego nie wpłynęła na utworzenie harmonogramu.

2.4. Szeregowanie zadań

Problem szeregowania zadań w systemie czasu rzeczywistego, polega na wyznaczeniu kolejności wykonywania zadań w taki sposób, aby wszystkie dotrzymały swoich terminów wykonania.

Algorytmy szeregowania zadań w rygorystycznych systemach czasu rzeczywistego można podzielić na [LO12]:

- o statycznych priorytetach
 - z wywłaszczaniem
 - bez wywłaszczania
- o dynamicznych priorytetach
 - z wywłaszczaniem
 - bez wywłaszczania

Oprócz podziału na statyczny i dynamiczny priorytet zadań, można wyszczególnić szeregowanie statyczne i dynamiczne. W przypadku statycznego planowania, harmonogram jest wyznaczany w momencie kompilacji, natomiast w dynamicznym planowaniu zadań – w trakcie działania systemu.

2.4.1. Możliwość utworzenia harmonogramu

Test, który pozwala na sprawdzenie, czy można utworzyć harmonogram dla danego zbioru zadań, nazywany jest testem planowalności (ang. *schedulability test*).

Sprawdzenie czy harmonogram jest możliwy do utworzenia, w przypadku istnienia zależności między zadaniami jest najczęściej zadaniem silnie NP-trudnym. Dla większej liczby zadań sprawdzenie każdej kombinacji ułożenia zadań jest praktycznie niemożliwe [Kop11].

Z tego powodu ustala się dodatkowe kryteria – konieczne i wystarczające, pozwalające ocenić, czy można utworzyć harmonogram. Niespełnienie kryterium koniecznego, wskazuje że harmonogramu na pewno nie da się utworzyć (natomiast spełnienie tego kryterium nie mówi o tym, czy harmonogram może być utworzony czy nie). Spełnienie kryterium wystarczającego mówi o tym, że harmonogram na pewno da się utworzyć (natomiast nie spełnienie warunku nie rozstrzyga tej kwestii).

W celu zbadania istnienia harmonogramu, wyznaczany jest tzw. współczynnik użycia procesora (ang. *utilisation factor*) U , [Kop11]. Jeżeli zadań jest N w systemie, czas wykonania pojedynczego zadania wynosi c_i , a jego okres p_i , wtedy współczynnik wykorzystania procesora wynosi [LL73]:

$$U = \sum_{i=1}^N \frac{c_i}{p_i}.$$

Dla zadań okresowych, wykonywanych na jednym procesorze, warunkiem koniecznym możliwości utworzenia harmonogramu jest spełnienie warunku [Kop11]:

$$U \leq 1.$$

Dla różnych algorytmów szeregowania istnieją różne kryteria wystarczające, określające możliwość istnienia harmonogramu. W dalszej części pracy do analizy planowalności zadań, zostały przyjęte następujące założenia [LL73]:

- zadania są okresowe, termin zakończenia jest równy okresowi,
- zadania są niezależne, zakończenie jednego zadania nie jest zależne od statusu ukończenia innego zadania,
- czas wykonania jest stały dla danego zadania, jest znany z góry i nie zmienia się w trakcie działania systemu,
- wszystkie zadania są w pełni wywłaszczalne,
- koszt zmiany kontekstu jest pomijalnie mały.

W przypadku spełnienia tych założeń – gdy zadania są niezależne, najczęściej stosowanym kryterium wystarczającym jest wartość współczynnika wykorzystania procesora U poniżej pewnego progu (o różnej wartości w zależności od algorytmu szeregującego). Kiedy zadania są zależne, do analizy należy uwzględnić dodatkowe parametry – takie jak maksymalny czas zablokowania zadania podczas synchronizacji.

2.4.2. Zadania o statycznym priorytecie

Zadania w systemie mają ustalony priorytet, który pomiędzy kolejnymi wywoływaniami zadania się nie zmienia.

Statyczny harmonogram

W harmonogramie statycznym (ang. *Fixed-Priority*) zadania mają priorytety przydzielone z góry, ustalone przez projektanta systemu. Wtedy harmonogram obliczany jest w momencie kompilacji, w trakcie działania systemu jest on już tylko realizowany. W tym celu potrzebna jest pełna wiedza *a priori* o zadaniach, np. maksymalny czas wykonania, czas zażądania zasobów oraz terminy zakończenia dla zadania. Wtedy wykonanie harmonogramu będzie wyzwalane przerwaniem zegarowym.

Taka strategia pozwala na zminimalizowanie czasu obliczeń planisty. Uzyskuje się też determinizm systemu, dokładnie wiadomo, które zadanie będzie wykonane w danym momencie czasowym. Wadą tego rozwiązania jest jednak mała skalowalność systemu. Każda zmiana w systemie, np. dodanie nowego zadania, wymaga powtórnej analizy i projektowania harmonogramu systemu.

Szeregowanie częstotliwościowe (RM)

Szeregowanie częstotliwościowe (ang. *Rate Monotonic*) zostało zaproponowane w 1973 roku przez [LL73]. Polega ono na ustaleniu priorytetu proporcjonalnie do częstotliwości występowania danego zadania. Zadanie o najkrótszym okresie otrzyma najwyższy priorytet, a zadanie o najdłuższym okresie – najniższy. Priorytety są ustalone przy starcie systemu i w trakcie jego działania już się nie zmieniają.

Przy założeniach wcześniej podanych dotyczących zadań, harmonogram można ustalić, jeżeli

$$U \leq N \left(2^{\frac{1}{N}} - 1 \right),$$

czyli przy nieskończonej ilości zadań:

$$U \leq \lim_{N \rightarrow \infty} N \left(2^{\frac{1}{N}} - 1 \right) = \ln 2 \approx 0.69.$$

Zatem harmonogram na pewno może być wyznaczony, jeżeli współczynnik wykorzystania procesora będzie $U \leq 0.69$. W rzeczywistości współczynnik wykorzystania często

może być wyższy. W szczególności może być równy jeden, jeżeli zadania mają harmoniczne okresy, tj. każdy z nich ma okres będący całkowitą wielokrotnością najkrótszego okresu.

Zaletą takiego uszeregowania jest lepsza skalowalność, niewiele obliczeń planisty, determinizm działania, nawet przy przeciążeniu systemu. Wadą jest, że harmonogram nie zawsze może być utworzony przy spełnieniu warunku $U \leq 1$.

2.4.3. Zadania o dynamicznym priorytecie

Zadania nie mają z góry ustalonego priorytetu, może on się zmieniać w poszczególnych wywołaniach zadania.

Szeregowanie terminowe (EDF)

W szeregowaniu terminowym (ang. *Earliest Deadline First*) zadanie mające najbliższy termin zakończenia będzie miało przydzielony najwyższy priorytet. Zadanie którego termin już minął, ma wyższy priorytet niż zadanie, którego termin dopiero ma nadejść.

Wykazano [LL73], że przy spełnionych założeniach co do modelu zadań w tym algorytmie nie ma czasu bezczynności procesora wymaganej do utworzenia harmonogramu – w przeciwieństwie do harmonogramu wyznaczonego poprzez algorytm częstotliwościowy. Oznacza to, że harmonogram może być utworzony zawsze, jeżeli współczynnik wykorzystania procesora spełnia:

$$U \leq 1.$$

Takie szeregowanie pozwala na elastyczność systemu i dostosowanie do zmieniających się warunków. Pozwala na utworzenie harmonogramu w sytuacji, gdy stosując szeregowanie częstotliwościowe nie jest to możliwe. Z drugiej strony, czas zajęty do wyznaczenia harmonogramu może być znaczny, zachowanie takiego systemu nie jest też deterministyczne podczas przeciążenia zadaniami i niespełnianie terminów będzie narastać lawinowo [LO12].

2.4.4. Szeregowanie zadań zależnych

W przypadku zadań, które potrzebują mieć dostęp do zasobów, są zależne od wykonania innych zadań należy odpowiednio zarządzać dostępem do zasobów, aby zachować spójność danych. Jednocześnie trzeba unikać niebezpiecznych zjawisk, jak zakleszczenia i inwersja priorytetów, które mogą spowodować wydłużenie czasu wykonania zadania, a ostatecznie niedotrzymanie terminu zadania.

Jednym z rozwiązań zapobiegania inwersjom priorytetów, jest niedopuszczenie do wyłączenia zadań posiadających zasoby [Mok93]. Jest to skuteczne rozwiązanie, gdy sekcje krytyczne są stosunkowo krótkie. W przeciwnym przypadku należy zastosować protokół dziedziczenia priorytetów (ang. *priority inheritance protocol*) lub protokół pułapu priorytetów (ang. *priority ceiling protocol*).

Inwersja priorytetów

W przypadku prawidłowo funkcjonującego systemu z wywłaszczaniem priorytetowym, zadanie o niższym priorytecie jest wywłaszczane przez zadanie o wysokim priorytecie. Dzieje się to natychmiast, gdy zadanie wysokopriorytetowe jest gotowe do uruchomienia. Aby utrzymać spójność danych, kiedy dwa zadania chcą mieć dostęp do jednego zasobu, dostęp do niego musi być kontrolowany.

Problem inwersji priorytetów następuje, gdy zadanie o wysokim priorytecie i o niskim chcą uzyskać dostęp do tego samego zasobu. Gdy zadanie o niskim priorytecie uzyska dostęp do zasobu jako pierwsze, może zostać wywłaszczone przez zadanie o wysokim priorytecie. Wtedy, by zachować spójność danych, zadanie o wyższym priorytecie musi poczekać na ukończenie działania zadania niżej priorytetowego i zwolnienie zasobu.

Gdyby jeszcze w systemie istniało gotowe zadanie o średnim priorytecie, po zwolnieniu procesora (przez zadanie wysokopriorytetowe w oczekiwaniu na zwolnienie zasobu), może ono wywłaszczyć zadanie najniższego priorytetu. Zasób zostałby zwolniony znacznie później, a zadanie oczekujące na niego, pomimo swojego wysokiego priorytetu musiałoby poczekać na wykonanie wszystkich zadań.

Takie blokowanie może spowodować niedotrzymanie terminów, nawet przy małym użyciu wspólnych zasobów, dlatego wprowadzono protokoły dziedziczenia priorytetów i pułapu priorytetów [SRL90].

Dziedziczenie priorytetów

Kiedy zadanie blokuje inne zadania o wyższych priorytetach, dokonuje się wtedy zmiany priorytetu blokującego zadania. Jest on równy priorytetowi najwyższego zadania oczekującego na zwolnienie zasobu trzymanego przez zadanie (priorytet dziedziczy się).

Natychmiast po wyjściu blokującego zadania ze sekcji krytycznej (i odblokowaniu wyżej priorytetowych zadań) priorytet tego zadania jest przywrócony do poprzedniej wartości.

Głównymi założeniami tego protokołu są [SRL90]:

- na procesorze wykonywane jest zadanie o najwyższym priorytecie,
- zadanie przed wejściem do sekcji krytycznej, musi otrzymać (i zablokować) semafor,
- zadanie zrzeka się wykonywania na procesorze, jeżeli semafor, który chce zająć, jest już zablokowany przez inne zadanie,
- zadanie o niższym priorytecie, które blokuje zadanie o wyższym, dostaje priorytet najwyższego priorytetowego zadania, które blokuje. Po wyjściu z sekcji krytycznej, zadaniu zostaje przydzielony jego poprzedni priorytet.

Jednakże, taki protokół nie zapobiega zakleszczeniom, a nawet sam może prowadzić do zakleszczeń lub wielokrotnego blokowania. Dlatego wprowadzono inny protokół – pułapu priorytetów [LO12].

Pułap priorytetów

Protokół ten jest rozwinięciem dziedziczenia priorytetów i ma na celu zapobieganie zakleszczeniom oraz wielokrotnym blokowaniom.

Każdy zasób (semafor) ma przydzielony priorytet (tzw. pułap priorytetu). Jest on równy największemu priorytetowi zadania, które może zająć ten semafor. Wtedy, dostęp do sekcji krytycznej może być wstrzymany dla zadania, jeżeli istnieje semafor trzymany przez inne zadanie, którego pułap jest większy bądź równy priorytetowi bieżącego zadania. Zadanie jest wpuszczone do sekcji krytycznej, tylko jeżeli priorytet zadania jest wyższy niż wszystkie pułapy priorytetów semaforów zajęte przez inne zadania.

Głównymi założeniami tego protokołu są [SRL90]:

- na procesorze jest wykonywane zadanie o najwyższym priorytecie spośród zadań gotowych, zanim to zadanie wejdzie w sekcję krytyczną musi uzyskać semafor,

- dostęp do semaforu będzie odmówiony zadaniu, jeżeli inne zadanie ma przydzielony już ten semafor, albo jeżeli priorytet zadania nie będzie większy niż pułap priorytetu semaforu o najwyższym pułapie priorytetu zajęтым przez pozostałe zadania,
- zadanie to wtedy będzie zablokowane przez zadanie trzymające semafor o najwyższym pułapie priorytetu – nastąpi wtedy dziedziczenie priorytetów,
- w przeciwnym przypadku zadanie uzyska semafor, a zadanie o najwyższym priorytecie, które było przez nie zablokowane (jeżeli jest) – zostanie wybudzone.

Wykazano [Kop11], że harmonogram może zostać utworzony, gdy spełnione są następujące nierówności (gdzie B_i to maksymalny czas blokowania zadania i przez zadania o niższym priorytecie):

$$\left(\frac{c_1}{p_1} + \frac{c_2}{p_2} + \dots + \frac{c_i}{p_i} + \frac{B_i}{p_i} \right) \leq i \left(2^{\frac{1}{i}} - 1 \right), \quad i = 1, 2, \dots, n.$$

Rozdział 3.

Projekt systemu

System został napisany w języku C, aby uzyskany kod można było przenosić na inne platformy jak i aby był mniej wymagający względem zasobów sprzętowych.

Zaprojektowany system jest systemem wyłuszczającym priorytetowym. Podobnie jak FreeRTOS, program użytkownika jest kompilowany razem z systemem [RTE14].

Dla użytkownika jest dostarczony interfejs systemu, polecenia specyficzne dla poszczególnych architektur procesora są zawarte w funkcjach portu. Dzięki temu system jest przenośny na inne platformy, wymaga to uzupełnienia funkcji portu za pomocą kodu dla właściwego dla danej platformy. Do systemu zaimplementowano symulacyjny port na system Linux. Platforma PC została wybrana ze względu na prostotę wgrywania programu, testowania i badania oprogramowania [Lab02].

Na rysunku 3.1. przedstawiono schemat funkcjonalności dostarczanych przez system. System został podzielony na kilka modułów:

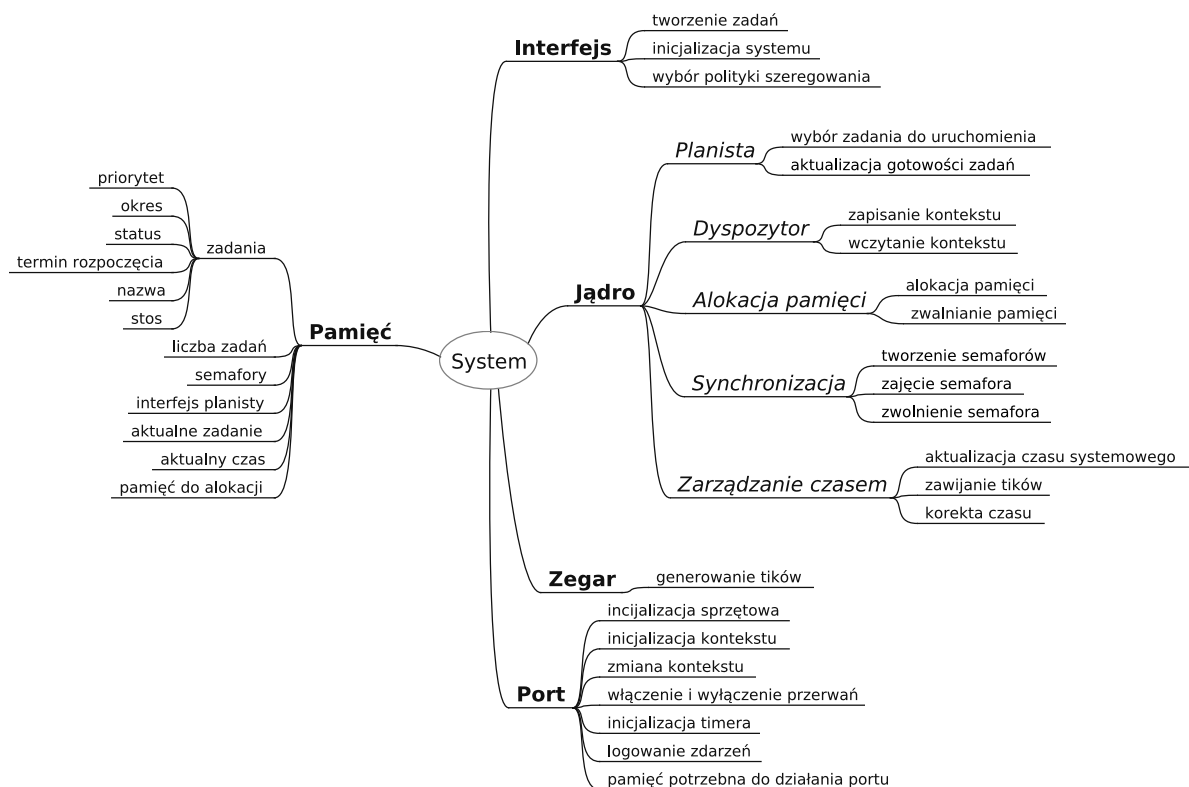
- moduł użytkownika, zawierający zadania użytkownika oraz program główny,
- interfejs dostępny dla użytkownika systemu, pozwalający na inicjalizację systemu, tworzenie zadań, wybór polityki szeregowania,
- moduł jądra, składający się z podmodułów: alokacji pamięci, dyspozytora, planisty, zarządzania czasem systemowym,
- moduł portu, czyli poleceń specyficznych dla architektury docelowego procesora, m.in. zmiany kontekstu, zablokowania przerwań, logowania zdarzeń w systemie.

3.1. Program główny

Użytkownik powinien napisać swoje zadania w nieskończonej pętli, a następnie: wywołać inicjalizację systemu, utworzyć zadania w systemie, rozpocząć planowanie i na koniec wejść w pustą nieskończoną pętlę. Pętla powinna być pusta, ponieważ program główny jest wykonywany jedynie do pierwszego przerwania zegarowego, potem nie będzie już planowany.

3.2. Interfejs użytkownika

Podstawowymi poleceniami, które też powinny być dostępne dla użytkownika systemu, są: inicjalizacja systemu, utworzenie zadania (wielkość stosu jest z góry określona w konfiguracji portu), włączenie i wyłączenie przerwań, uśpienie zadania, uruchomienie planisty.



Rysunek 3.1. Schemat funkcjonalności systemu

Algorytm 3.1. Program główny

- 1: Zainicjalizuj system.
- 2: Utwórz zadania.
- 3: Rozpocznij planowanie zadań.
- 4: **Dopóki** *prawda* **wykonaj:**
- 5: {pusta operacja}
- 6: **koniec.**

Algorytm 3.2. Inicjalizacja systemu

- 1: Zainicjalizuj sprzęt.
- 2: Zainicjalizuj pamięć systemową.
- 3: Zainicjalizuj planistę.
- 4: Utwórz zadanie bezczynności.
- 5: Zainicjalizuj timer.

Algorytm 3.3. Tworzenie zadania

Dane: uchwyt funkcji wykonywanej przez zadanie, priorytet, okres.

- 1: Uzyskaj początek pamięci systemowej.
 - 2: Wyłącz przerwania.
 - 3: Wczytaj liczbę zadań z pamięci systemowej.
 - 4: **Jeżeli** liczba zadań jest mniejsza niż maksymalna liczba zadań w systemie **to**:
 - 5: Zainicjalizuj stos zadania.
 - 6: Zapisz uchwyt funkcji wykonywanej przez zadanie.
 - 7: Zapisz priorytet i okres zadania.
 - 8: Zwiększ liczbę zadań.
 - 9: **koniec**.
 - 10: Włącz przerwania.
 - 11: **powrót:** zwróć liczbę zadań.
-

3.3. Jądro

W ramach inicjalizacji systemu, często należy dokonać inicjalizacji sprzętowej (procesora, timera) oraz programowej (inicjalizacja pamięci). Przy inicjalizacji systemu tworzy się zadanie bezczynności – jest to zadanie o najniższym priorytecie w systemie. To zadanie jest uruchamiane, kiedy żadne inne zadanie nie może być wykonywane w danej chwili na procesorze.

3.4. Pamięć

W celu zarządzania pamięcią, na jej obraz nałożono strukturę z języka C, zawierającą następujące pola:

- o tablicy informacji o zadaniach w systemie,
- o interfejsu planisty,
- o liczby zadań w systemie,
- o tablicy semaforów,
- o tablicy na alokację danych dynamicznych,
- o wskaźnika na pierwszy bajt pamięci dynamicznej,
- o wskaźnika na pierwszy wolny bajt pamięci dynamicznej.

Z kolei tablica informacji o zadaniu składa się ze stosu dla zadania (stałej wielkości), priorytetu zadania, okresu, terminu następnego rozpoczęcia oraz nazwy zadania.

3.5. Przerwania zegarowe, czas systemowy

W zaprojektowanym systemie obsługa przerwania zegarowego powiększa zmienną liczącą tiki (z możliwością zawijania) i uruchamia planistę. Istnieje możliwość ustawienia częstotliwości pojawiania się przerwania zegarowego w konfiguracji portu.

Zawinięcie licznika generuje nieciągłość, dlatego wszystkie operacje porównania tików, ich dodawania czy odejmowania, wykorzystują zaimplementowane funkcje uwzględniające to zjawisko.

Algorytm 3.4. Dodawanie tików zegarowych**Dane:** obecny tik, ilość tików do dodania.

- 1: nowy tik := obecny tik + ilość tików
- 2: **Jeżeli** obecny tik jest większy niż nowy tik **lub** nowy tik jest większy niż maksymalna liczba tików **to**:
- 3: nowy tik := ilość tików – (maksymalna liczba tików – obecny tik)
- 4: **koniec.**
- 5: **powrót:** zwróć wartość nowego tik.

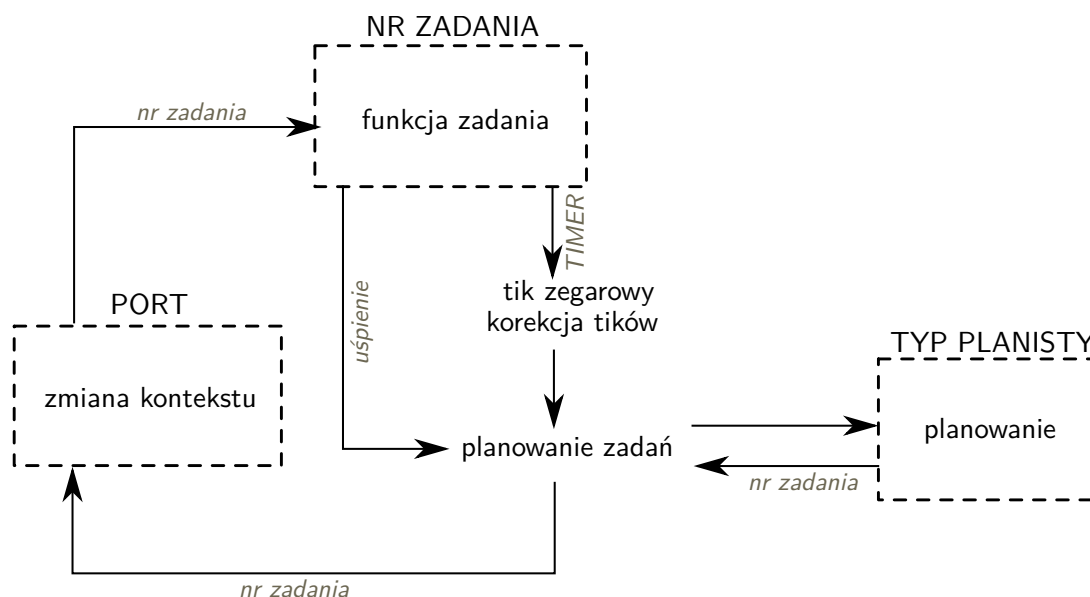
Algorytm 3.5. Przerwanie zegarowe

- 1: Uzyskaj początek pamięci systemowej.
- 2: Wczytaj obecną liczbę tików z pamięci systemowej.
- 3: Zwiększ obecną liczbę tików o ustaloną wartość tik (patrz: algorytm 3.4.).
- 4: Uruchom planistę.

3.6. Planista

W projekcie systemu głównym celem było dostarczenie interfejsu, który pozwala na implementację różnych polityk planowania zadań oraz działanie systemu niezależne od rodzaju polityki (algorytmy będą szerzej omówione w rozdziale 4.). Takie podejście pozwala też zmienić politykę szeregowania zadań już w trakcie pracy systemu.

Na rysunku 3.2. przedstawiono działanie planowania w systemie. Podczas działania systemu, w każdym tiku wywoływany jest planista, który zwraca informację jakie zadanie powinno być w danej chwili wykonywane na procesorze. Jeżeli jest to inne zadanie niż obecnie wykonywane, następuje przełączenie kontekstu. Samo uruchomienie planisty odbywa się wewnątrz obsługi przerwania zegarowego (a zatem nie można już wyłączyć przerw). Natomiast obsługa przerw jest przywracana albo po dokonaniu przełączenia kontekstu albo na koniec działania planisty.



Rysunek 3.2. Schemat działania planowania w systemie

W przypadku implementacji różnych algorytmów, zmieniany jest tylko uchwyt funkcji

obliczającej zadanie do wykonania na procesorze w danym momencie.

Algorytm 3.6. Włączenie planisty

- 1: Uzyskaj początek pamięci systemowej.
 - 2: Odczytaj numer obecnie wykonywanego zadania.
 - 3: Uruchom algorytm szeregujący i zwróć numer zadania, które powinno być wykonywane na procesorze. [patrz: algorytmy 4.4. i 4.5.]
 - 4: **Jeżeli** numery obecnego zadania oraz zadania, które powinno być wykonywane, różnią się **to**:
 - 5: Zapisz zadanie podane przez planistę, jako obecne.
 - 6: Przełącz kontekst jądra i włącz przerwania.
 - 7: **w przeciwnym wypadku:**
 - 8: Włącz przerwania.
 - 9: **koniec.**
-

3.7. Alokacja pamięci

W wielu systemach czasu rzeczywistego nie stosuje się dynamicznej alokacji pamięci, ze względu na niedeterminizm takiej operacji. Czas oczekiwania na przydział pamięci może się zmieniać w różnych wywołaniach, powoduje to zwiększenie i skomplikowanie kodu programu oraz możliwość braku przydziału pamięci dla zadania, przez co zadanie może nie wykonać się. [RTE14]. Kolejnym aspektem jest też zwalnianie pamięci: należy się zastanowić przy implementacji, czy należy zwalniać zaalokowaną pamięć. Jeżeli tak, w jaki sposób przydzielić innym zadaniom zwolnioną pamięć, jak radzić sobie z fragmentacją pamięci.

W zaprojektowanym systemie zaimplementowano alokację pamięci, ale bez możliwości zwalniania jej. Ze względu na uproszczenie obliczeń, przyjęto algorytm liniowej alokacji pamięci z możliwością ustawienia w konfiguracji portu jednostek alokacji pamięci (ang. *padding*).

Algorytm 3.7. Dynamiczna alokacja pamięci

Dane: rozmiar pamięci do alokacji.

- 1: Uzyskaj początek pamięci systemowej.
 - 2: Zapisz adres pamięci dla procesu.
 - 3: W pamięci powiększ adres wolnej pamięci o żądany rozmiar pamięci do alokacji.
 - 4: **Jeżeli** adres wolnej pamięci nie jest całkowitą wielokrotnością jednostki alokacji pamięci **to**:
 - 5: powiększ adres wolnej pamięci do następnej wielokrotności jednostki alokacji pamięci
 - 6: **koniec.**
 - 7: **powrót:** zwróć zapisany adres pamięci dla procesu.
-

3.8. Synchronizacja międzyprocesowa

Moduł synchronizacji i komunikacji między zadaniami nie został zaimplementowany. Dostarczono interfejs blokowania i odblokowywania przerwań systemowych.

3.9. Port na system Linux x86

Istnieją polecenia, które są kluczowe do działania systemu, a ich implementacja jest ściśle związana z architekturą procesora, na którym to system ma być uruchomiony. Należą do nich:

- włączenie i wyłączenie przerw,◦ obsługa timera,
- inicjalizacja sprzętowa procesora,
- inicjalizacja i zmiana kontekstu jądra,
- logowanie wydarzeń w systemie.

Aby system mógł być wieloplatformowy, implementacje tych funkcji przeniesiono do osobnego modułu.

W przypadku portu na system Linux x86, system nie może w pełni działać w czasie rzeczywistym ze względu na działające w tle jądro Linuxa, które nie jest w pełni deterministyczne. Jednakże, przy niewielkim obciążeniu systemu i dłuższym czasie pomiędzy tikami, pozwala on zasymulować taki system oraz przeprowadzić badanie i wizualizację. Ze względu na kontrolę uprawnień przez system, nie jest możliwe użycie poleceń assemblera służących do wyłączania i włączania przerw procesora, jak również do tworzenia i zmiany kontekstu. Do wykonania tej czynności uprawnione jest jedynie jądro Linuxa [Lov10]. Dlatego też w implementacji portu symulacyjnego na ten system skorzystano z gotowych mechanizmów dostarczanych przez system Linux.

3.9.1. Inicjalizacja sprzętowa

W zależności od architektury i implementacji, potrzebne są czynności umożliwiające inicjalizację sprzętową pamięci.

Przenosząc system na platformę PC, pamięć (wielkość struktury pamięci) została odwzorowana do pliku tymczasowego. Ustawione zostały odpowiednie atrybuty, aby wszystkie procesy mogły ją odczytywać – jest to odpowiednik pamięci RAM. Dalej, utworzono semafor binarny bazując na bibliotece `pthread` – potrzebny do włączania i wyłączania przerw systemowych (patrz: podsekcja 3.9.3.) oraz zainicjowano odpowiednio timer, szerzej omówiony w podsekcji 3.9.4.

3.9.2. Kontekst jądra

Zmiana kontekstu również jest silnie związana z architekturą procesora. Zamysłem było, aby inicjalizacja kontekstu zadania była rozwiązana poprzez utworzenie nowego procesu w systemie. Zaraz po utworzeniu wysłał on do siebie sygnał zatrzymania (SIGSTOP), a następnym poleceniem (które już wykona się dopiero po wznowieniu zadania) jest już wykonanie właściwej funkcji. Proces rodzica, po utworzeniu się nowego procesu, zapisuje w pamięci identyfikator procesu (PID).

Na podstawie danych do zmiany kontekstu szukany jest PID procesu, który ma być uruchomiony. Istotną rzeczą jest, że przełączanie kontekstu jest wywoływane z poziomu zadania obecnie wykonywanego, dlatego sygnał stopu jest wysyłany do siebie. Przełączanie kontekstu jądra jest wywoływane, gdy przerwania są wyłączone, na koniec wykonywania przełączania kontekstu przerwania są włączane.

Algorytm 3.8. Inicjalizacja kontekstu w porcie na system Linux x86

Dane: uchwyt funkcji do wykonania, wskaźnik na stos dla nowego zadania.

- 1: Sklonuj obecny proces.
 - 2: **Jeżeli** jest to proces potomny **to:**
 - 3: Zatrzymaj swoje wykonywanie.
 - 4: Zaczynj wykonywać podaną funkcję.
 - 5: **w przeciwnym wypadku:**
 - 6: Zapisz w pamięci identyfikator procesu potomnego.
 - 7: Zapisz w pamięci przydzielony mu stos.
 - 8: **koniec.**
-

Algorytm 3.9. Zmiana kontekstu w porcie na system Linux x86

Dane: wskaźnik na stos do zapisania, wskaźnik na stos do wczytania.

- 1: Na podstawie wskaźników na stosy, znajdź PID procesu, który ma być uruchomiony.
 - 2: Wyślij sygnał startu dla znalezionej PID.
 - 3: Włącz przerwania.
 - 4: Wyślij sygnał stopu do siebie *{obsługa przełączania kontekstu jest wywoływana z poziomu bieżącego zadania}*
-

3.9.3. Włączenie i wyłączenie przerwań

Włączenie i wyłączanie przerwań zaimplementowano w postaci dostępnego dla każdego zadania semaforu binarnego (odwzorowanego w pamięci i dodanego do struktury portu). Zablokowanie przerwań polega na zajęciu semaforu, a przywrócenie przerwań – zwolnieniu go. Próba zajęcia zajętego już semaforu powoduje oczekiwanie do momentu, gdy zostanie on zwolniony.

3.9.4. Inicjalizacja i uruchomienie timera

Aby przybliżyć działanie timera do działania sprzętowego, timer jest również uruchomiony jako osobny proces, cyklicznie wysyłający sygnał zegarowy. Sygnał ten jest wysyłany do zadania obecnie wykonywanego, informacja które zadanie jest wykonywane jest uzyskiwana z pamięci wspólnej. Co określony kwant czasu wysyłany jest sygnał zegarowy.

Algorytm 3.10. Uruchomienie i działanie timera w porcie na system Linux x86

Dane: uchwyt funkcji obsługującej przerwanie, czas trwania tik.

- 1: Sklonuj obecny proces.
 - 2: **Jeżeli** jest to proces potomny **to:**
 - 3: Zapisz uchwyt funkcji obsługującej przerwanie.
 - 4: **Dopóki** *prawda* **wykonaj:**
 - 5: Uśpij się na czas trwania tik.
 - 6: Wyłącz przerwania.
 - 7: Znajdź PID aktywnego procesu.
 - 8: Wyślij sygnał przerwania zegarowego do aktywnego procesu.
 - 9: **koniec.**
 - 10: **koniec.**
-

Rozdział 4.

Planista

Moduł planisty określa, jakie zadanie w danym momencie ma być wykonywane na procesorze. W systemie dostępny jest interfejs pozwalający na zastosowanie różnych polityk szeregowania, bez wprowadzania dodatkowych zmian w samym systemie. Planowanie jest uruchamiane co tik zegarowy.

W projekcie systemu zaimplementowano następujące polityki szeregowania:

- częstotliwościowe (RM),
- terminowe (EDF).

Możliwe jest rozszerzenie modułu planisty o własną politykę szeregowania.

4.1. Interfejs

Interfejs planisty składa się z uchwytu funkcji szeregującej oraz z wskaźnika na dane potrzebne planiście do wykonania uszeregowania. Dostępne metody szeregowania są w postaci typu wyliczeniowego, który jest podawany jako argument przy inicjalizacji planisty.

Algorytm 4.1. Inicjalizacja planisty

Dane: typ szeregowania, dane planisty

- 1: **W zależności od** typu szeregowania:
 - 2: **w przypadku** szeregowania częstotliwościowego::
 - 3: Zapisz uchwyt na funkcję szeregowania częstotliwościowego.
 - 4: Utwórz dane planisty.
 - 5: Zapisz wskaźnik na dane planisty.
 - 6: **w przypadku** szeregowania terminowego::
 - 7: Zapisz uchwyt na funkcję szeregowania terminowego.
 - 8: Utwórz dane planisty.
 - 9: Zapisz wskaźnik na dane planisty.
 - 10: **koniec.**
-

W zaimplementowanych algorytmach planista nie potrzebował dodatkowych danych do planowania. Jednakże, aby zapewnić możliwość rozszerzenia algorytmów planowania, jako argument wywołania funkcji planisty podawany jest wskaźnik na dane (przechowywany w pamięci).

4.2. Algorytmy szeregujące

Algorytm 4.2. Sprawdzenie gotowości zadania.

Dane: obecny tik, tik następnego pojawienia się zadania, okres zadania

- 1: Oblicz wartość tik za następny okres zadania. [patrz: algorytm 3.4.]
 - 2: **Jeżeli** obecny tik < wartość tik za następny okres zadania **to**:
 - 3: {nie nastąpi zawinięcie tik}
 - 4: **Jeżeli** obecny tik \geq tik następnego pojawienia się zadania **to**:
 - 5: **powrót:** zadanie jest gotowe
 - 6: **w przeciwnym wypadku:**
 - 7: **powrót:** zadanie nie jest gotowe
 - 8: **koniec.**
 - 9: **w przeciwnym wypadku:**
 - 10: {nastąpi zawinięcie tik}
 - 11: **Jeżeli** wartość tik za następny okres zadania < tik następnego pojawienia się zadania \leq obecny tik **to**:
 - 12: **powrót:** zadanie jest gotowe
 - 13: **w przeciwnym wypadku:**
 - 14: **powrót:** zadanie nie jest gotowe
 - 15: **koniec.**
 - 16: **koniec.**
-

4.2.1. Szeregowanie częstotliwościowe (RM)

Przy inicjalizacji planisty, zadaniom przydziela się priorytety w zależności od okresu – im mniejszy okres tym większy priorytet. Następnie spośród zadań gotowych wybierane jest zadanie o największym priorytecie.

Aby uniknąć przy inicjalizacji dodatkowych operacji sortowania, przyjmuje się założenie, że okres zadania nie będzie większy niż największy okres podany w konfiguracji portu. Wtedy można od wartości maksymalnej okresu odjąć okres bieżącego zadania, uzyskując priorytet danego zadania. Takie rozwiązanie pozwala na przydział priorytetów bez dodatkowego zużycia pamięci oraz obciążania procesora.

Ze względu na fakt, że w systemie jest określona maksymalna liczba zadań oraz że ta liczba jest znana projektantowi systemu, zdecydowano na liniowe przeglądanie zadań do uszeregowania. Dzięki temu nie trzeba implementować dodatkowych struktur i wykonywać kolejnych obliczeń. Uzyskuje się determinizm czasu i sposobu działania.

Algorytm 4.3. Inicjalizacja szeregowania częstotliwościowego (RM)

Dane: struktura danych planisty

- 1: Ustaw maksymalny priorytet jako największa wartość całkowitoliczbowa bez znaku.
 - 2: **Dla wszystkich** zadań, z pominięciem zadania bezczynności **wykonaj**:
 - 3: priorytet zadania := maksymalny priorytet – okres zadania
 - 4: **koniec.**
-

4.2.2. Szeregowanie terminowe (EDF)

Szeregowanie polega na wybraniu spośród zadań gotowych zadania o najbliższym terminie. Podobnie jak w przypadku szeregowania częstotliwościowego, zadania były przeglądane liniowo.

Algorytm 4.4. Szeregowanie częstotliwościowe (RM)

Dane: struktura danych planisty

- 1: Ustaw zapamiętany indeks na indeks zadania beczynności.
 - 2: Ustaw zapamiętany priorytet na priorytet zadania beczynności.
 - 3: **Dla wszystkich** zadań w systemie z wyłączeniem zadania beczynności **wykonaj:**
 - 4: Sprawdź gotowość zadania [patrz: algorytm 4.2.].
 - 5: **Jeżeli** bieżące zadanie jest gotowe **to:**
 - 6: **Jeżeli** priorytet bieżącego zadania jest większy od zapamiętanego priorytetu **to:**
 - 7: Ustaw zapamiętany indeks na indeks bieżącego zadania.
 - 8: Ustaw zapamiętany priorytet na priorytet bieżącego.
 - 9: **koniec.**
 - 10: **koniec.**
 - 11: **koniec.**
 - 12: **powrót:** zwróć indeks zadania o największym priorytecie
-

Pojawia się jednak utrudnienie w postaci obliczania i porównywania czasu. W powodu zawijania tików, nie można porównywać samych wartości następnego tików zadania. Tik tuż przed zawinięciem jest wcześniejszy niż tik tuż po zawinięciu, a jednak ten wcześniejszy ma większą wartość.

Widać też, że w przedstawionym algorytmie, jeżeli dwa zadania mają ten sam termin, to wybierane jest te, które jest porównywane jako pierwsze. w praktyce oznacza to, że pierwsze uruchamiane będzie zadanie, które zostało utworzone w systemie wcześniej (i ma mniejszy identyfikator zadania).

Algorytm 4.5. Szeregowanie terminowe (EDF)

Dane: struktura danych planisty

- 1: Ustaw zapamiętany najniższy tik na maksymalną liczbę tików w systemie.
 - 2: Ustaw zapamiętany indeks na indeks zadania beczynności.
 - 3: **Dla wszystkich** zadań w systemie z wyłączeniem zadania beczynności **wykonaj:**
 - 4: Sprawdź gotowość zadania [patrz: algorytm 4.2.]
 - 5: **Jeżeli** zadanie jest gotowe **to:**
 - 6: **Jeżeli** tik następnego pojawienia się zadania jest mniejszy niż zapisany najmniejszy tik (z uwzględnieniem zawijania tików) **to:**
 - 7: Ustaw zapamiętany indeks na indeks bieżącego zadania.
 - 8: Ustaw zapamiętany najniższy tik na tik następnego pojawienia się bieżącego zadania.
 - 9: **koniec.**
 - 10: **koniec.**
 - 11: **koniec.**
 - 12: **powrót:** zwróć indeks zadania o najbliższym terminie
-

Rozdział 5.

Rezultaty

5.1. Wizualizacja

Aby móc przedstawić uzyskane harmonogramy w poszczególnych strategiach szeregujących, zaimplementowano program wizualizujący przebieg zadań w systemie. Na podstawie pliku generowanego przez system podczas zmiany kontekstu, tworzony jest obraz w formacie TEX. Pozwala to na proste załączenie obrazów do dokumentacji. Sposób działania programu do wizualizacji oraz instrukcja użytkownika opisane jest w sekcjach 6.1. i 6.2.

5.2. Testy

Do celów testów zaimplementowano dodatkową funkcję, pozwalającą na symulację obliczeń na procesorze przez określony czas. Musiała ona być w pełni wywłaszczalna, aby zmiana kontekstu nie zmieniała czasu wykonywania.

5.2.1. Test 1. Niewielkie obciążenie procesora

Wykonano symulację działania systemu, przy zbiorze zadań umieszczonych w tabeli 5.1. Współczynnik użycia procesora wyniósł 0,73.

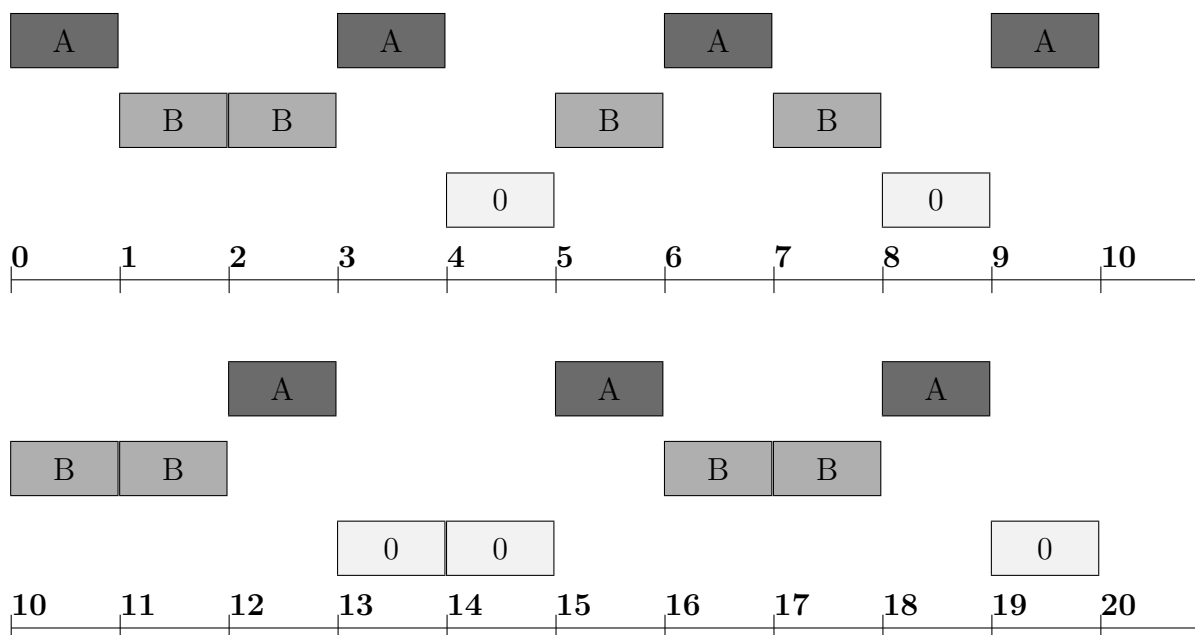
Tabela 5.1. Zbiór zadań w 1. teście

τ_i	A	B	0
p_i	3	5	—
c_i	1	2	—

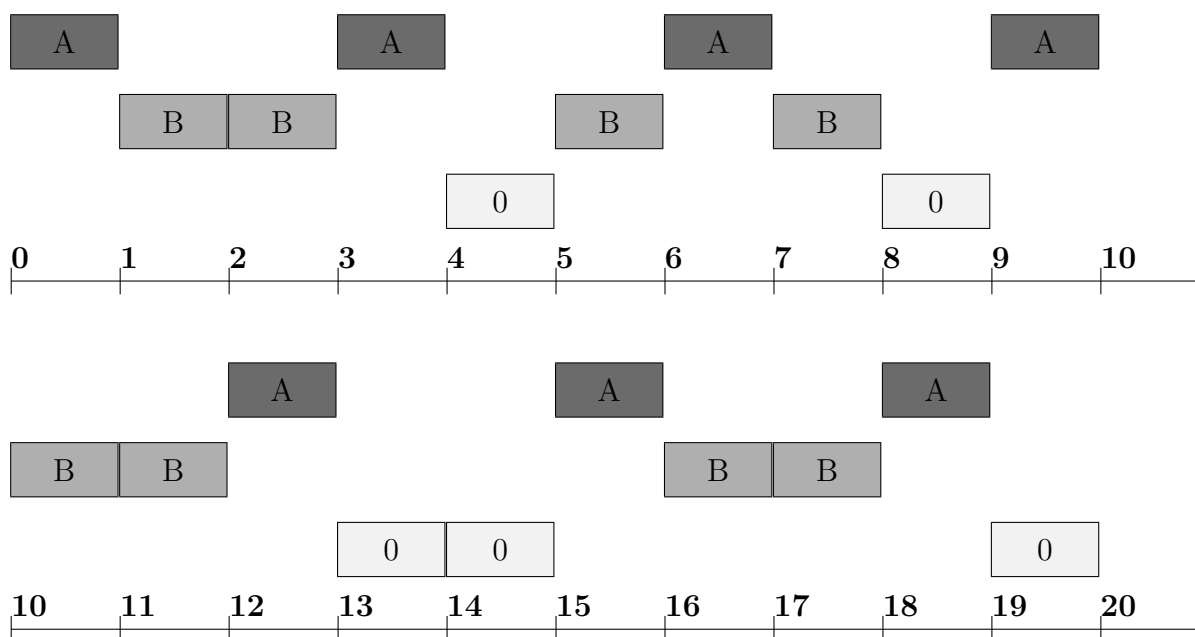
Wynik uszeregowania jest ten sam dla obydwóch algorytmów. Wszystkie zadanie dotrzymały swoich terminów wykonania.

5.2.2. Test 2. Znaczne obciążenie procesora

Następnie przetestowano działanie algorytmów, przy większej liczbie zadań oraz większym współczynniku wykorzystania procesora. Wyniósł on około 0,94. Parametry zadań



Rysunek 5.1. Wizualizacja uszeregowania zadań dla algorytmu RM dla 1. testu

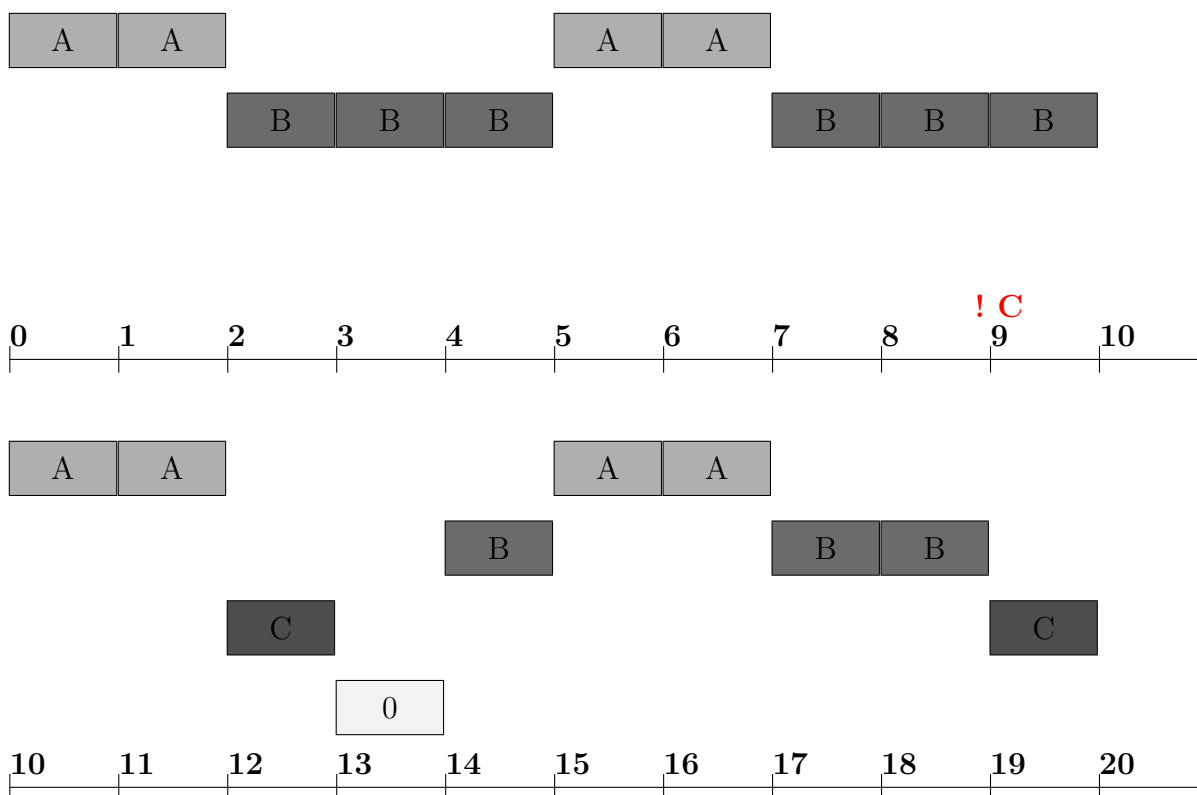


Rysunek 5.2. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 1. testu

zostały przedstawione w tabeli 5.2.

Tabela 5.2. Zbiór zadań w 2. teście

τ_i	A	B	C	0
p_i	5	7	9	—
c_i	2	3	1	—

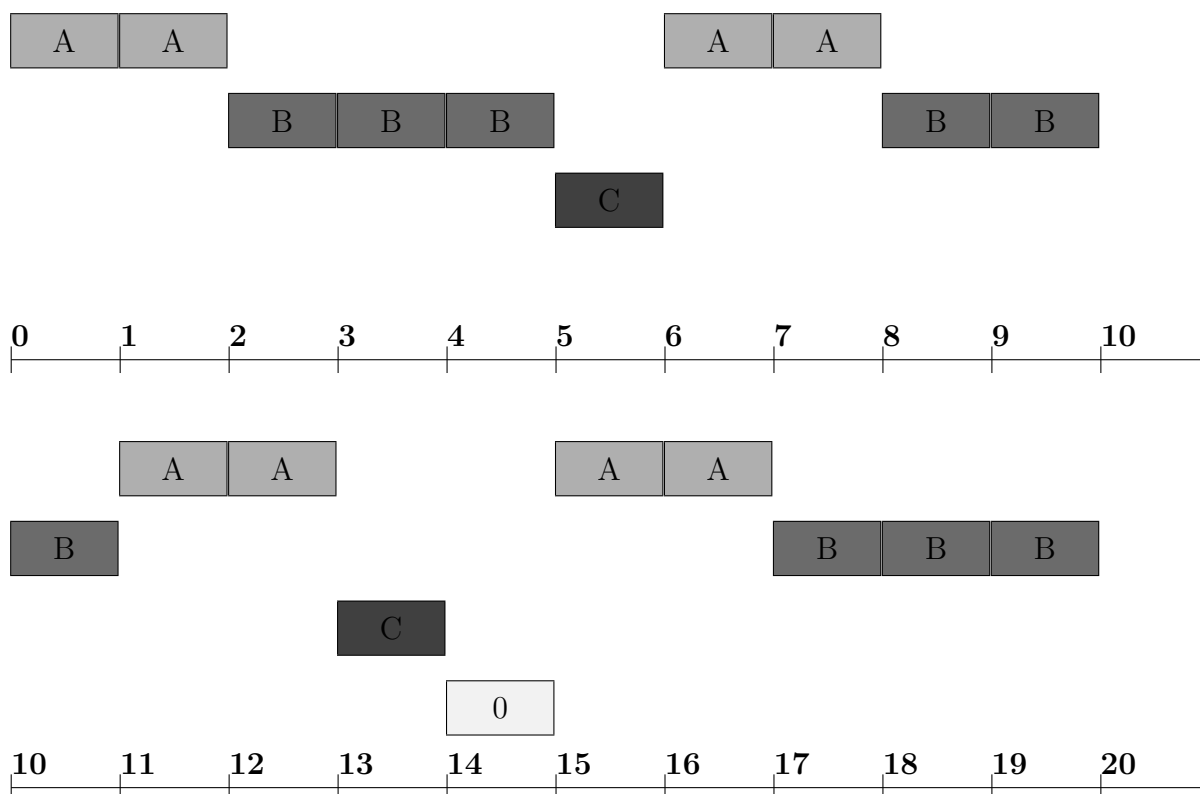


Rysunek 5.3. Wizualizacja uszeregowania zadań dla algorytmu RM dla 2. testu

Można zauważyć, że uszeregowania się różnią. W przypadku stosowania algorytmu szeregowania częstotliwościowego, zadanie C (o najdłuższym okresie) nie dotrzymuje swojego terminu zakończenia w 9. tik. Szeregowanie terminowe pozwala na terminowe wykonanie się wszystkich zadań. W 6. tik, pomimo gotowego zadania A, uruchamiane jest zadanie C (ma bliższy termin). Dzięki temu uzyskuje się harmonogram pozwalający na terminowe wykonanie wszystkich zadań.

5.2.3. Test 3. Zadania harmoniczne, pełne wykorzystanie procesora

W tym teście sprawdzono sposób uszeregowania zadań, gdy okresy zadań są harmoniczne – każdy z nich jest całkowitą wielokrotnością najkrótszego okresu zadania. Współ-



Rysunek 5.4. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 2. testu

czynniki użycia procesora wyniosły $U = 1$. Okresy zadań oraz ich czasy wykonania zostały przedstawione w tabeli 5.3.

Tabela 5.3. Zbiór zadań w 3. teście

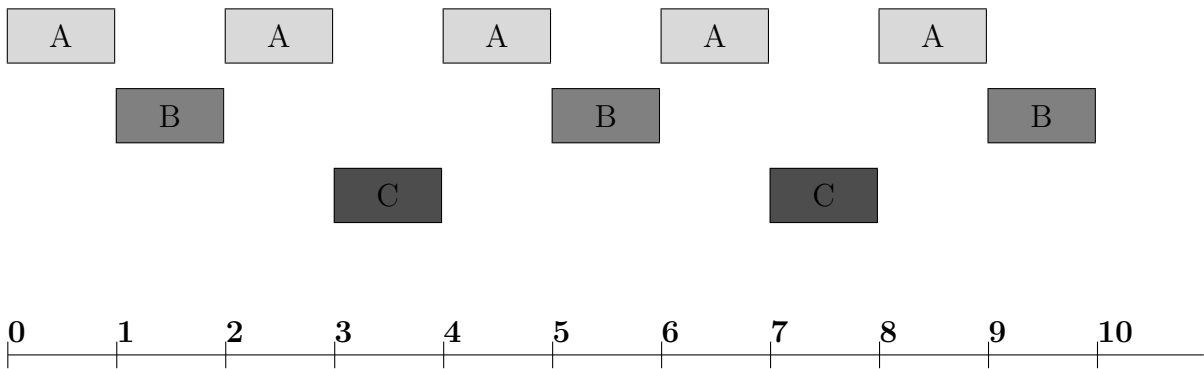
τ_i	A	B	C	0
p_i	2	4	8	—
c_i	1	1	2	—

Można zauważyć, że obydwa harmonogramy spełniają terminy zadań, przy pełnym obciążeniu systemu.

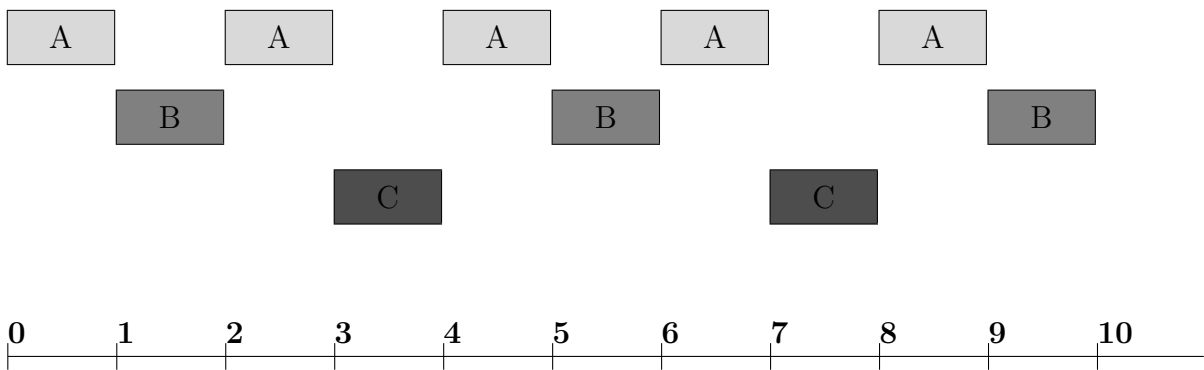
5.2.4. Test 4. Przeciążenie procesora

Następnie zbadano zachowanie systemu przy przeciążeniu zadaniami. Może ono wynikać z np. ze złego projektu systemu, wydłużenia czasu obliczeń planisty lub blokowania się zadań. Parametry zadań przedstawione zostały w tabeli 5.4. W tym teście współczynnik wykorzystania procesora $U > 1$, był on równy 1,11.

W przypadku przeciążenia systemu, zadania w każdym uszeregowaniu nie dotrzymują swoich terminów. Kiedy zadania uszeregowano algorytmem częstotliwościowym, zadaniem, które zdarzało się niedotrzymywanie terminu, było zadanie D, o najniższym prio-



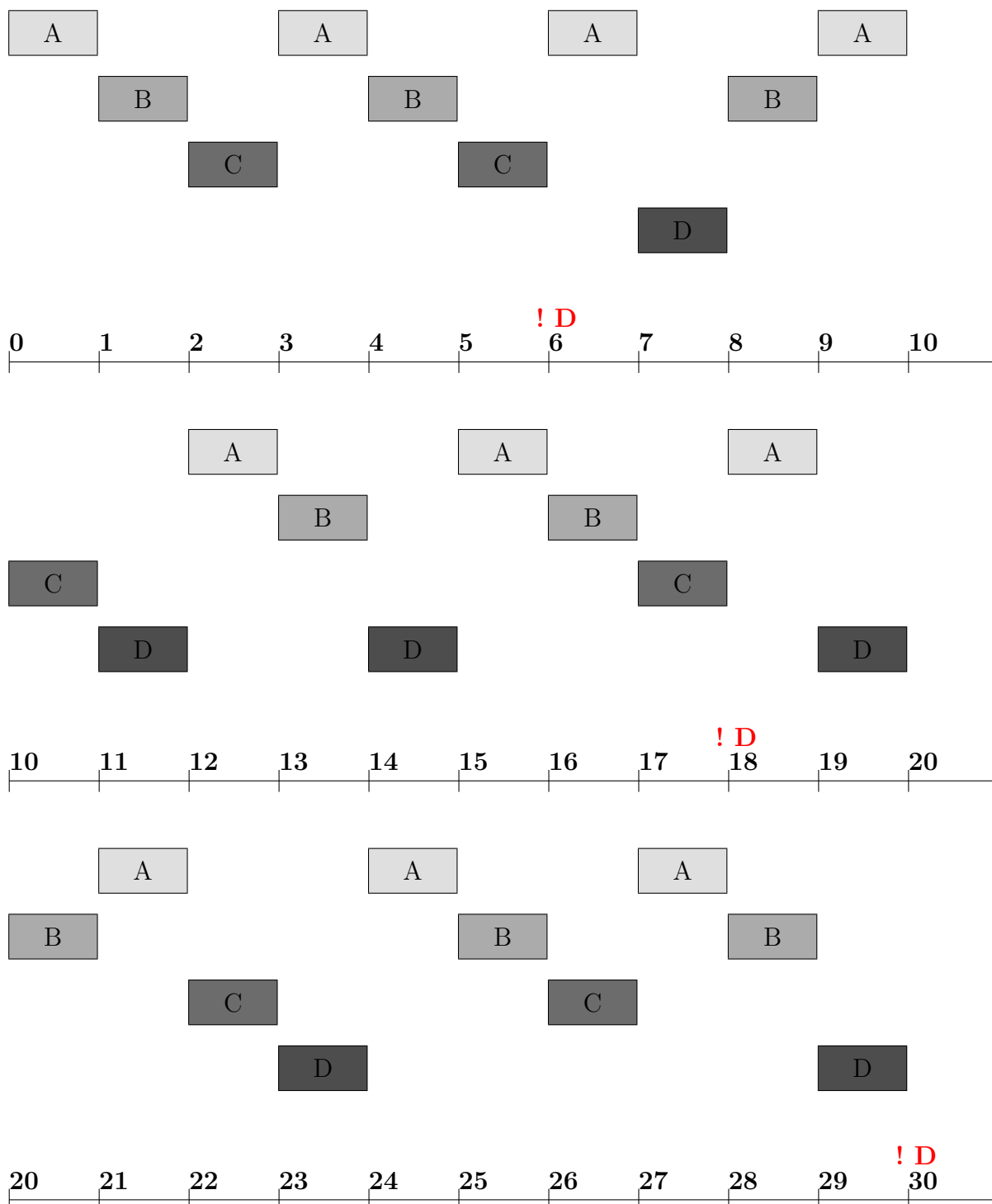
Rysunek 5.5. Wizualizacja uszeregowania zadań dla algorytmu RM dla 3. testu



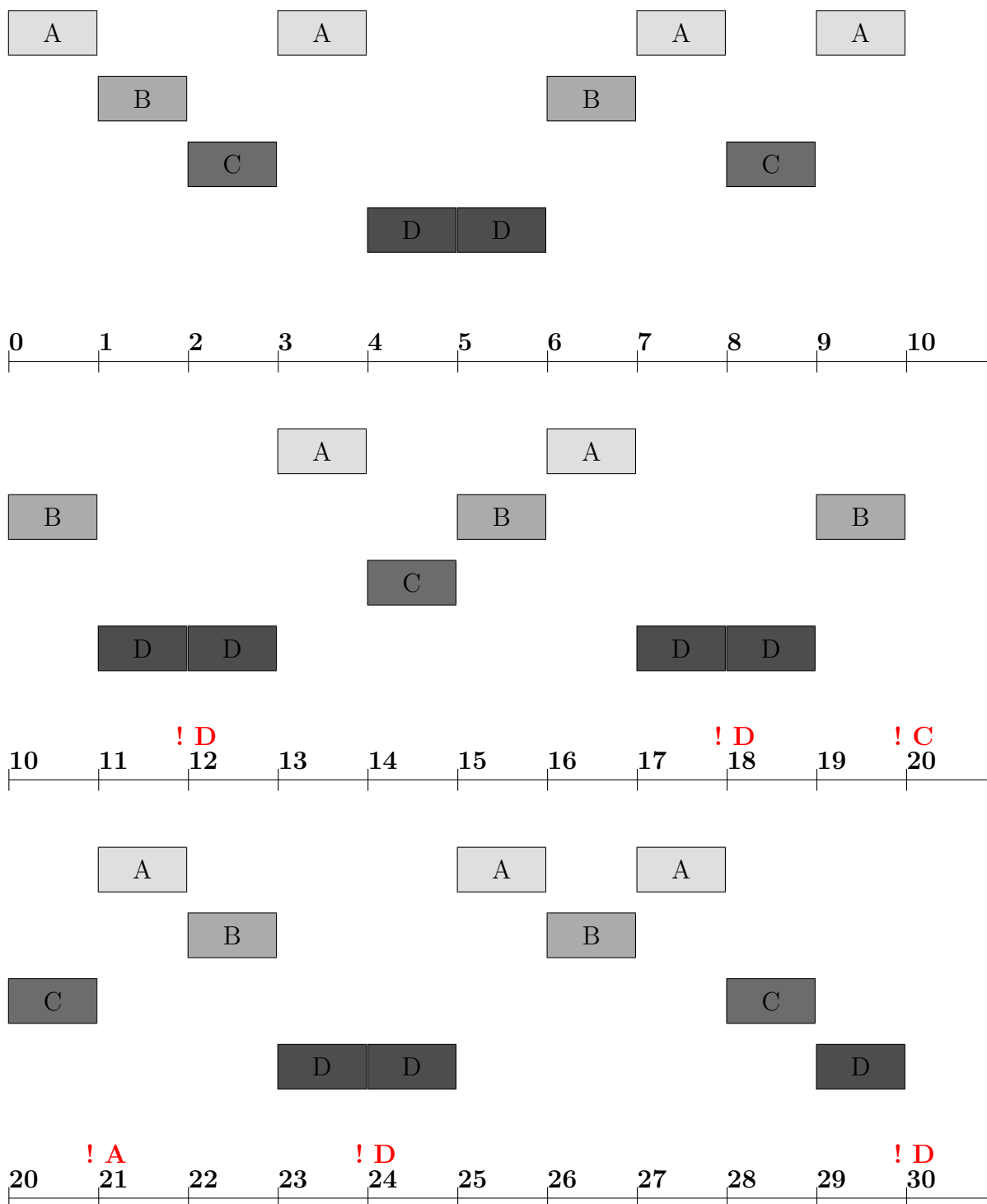
Rysunek 5.6. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 3. testu

Tabela 5.4. Zbiór zadań w 4. teście

τ_i	A	B	C	D	0
p_i	3	4	5	5	—
c_i	1	1	1	2	—



Rysunek 5.7. Wizualizacja uszeregowania zadań dla algorytmu RM dla 4. testu



Rysunek 5.8. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 4. testu

rytecie. Nie spełniło ono swojego terminu w tiku 6, w tiku 18 i 30 nie dokończyło swoich obliczeń.

Szeregowanie algorytmem terminowym było zaimplementowane tak, że w przypadku zadań o tym samym terminie wybierane było pierwsze z nich (tzw. leniwe wybierane). Szeregowanie algorytmem terminowym spowodowało, że różne zadania nie dotrzymywały terminów. W tikach 12, 18, 24, 30 zadanie D nie dokończyło na czas swoich obliczeń, oprócz tego w tiku 20 zadanie C nie dotrzymało terminu, a w tiku 21. – zadanie A.

Rozdział 6.

Implementacja

6.1. Algorytm wizualizacji

Program do wizualizacji miał za zadanie stworzenie obrazu kolejności i czasu wykonania zadań, na podstawie odpowiedniego pliku z danymi. Dane analizowane są na podstawie wejściowego pliku z danymi.

Informacje o kolejnych zadaniach są zapisane w pliku, gdzie każda nowa linia to nowy wpis dotyczący pojawiania się zadań w systemie. Format wygląda następująco:

```
czas_s   czas_ms   nr_tiku   id_zadania   priorytet_zadania
```

Dane wynikowe są przetwarzane do obrazu w formacie TEX, który można załączyć w odpowiednim dokumencie i wygenerować do formatu PDF. Umożliwia to szybkie tworzenie dokumentacji. Algorytm 6.1. pokazuje sposób przetwarzania obrazu.

We wstępnym przetwarzaniu zadań, tworzona jest tablica numerów zadań i odpowiadającym im priorytetom. Następnie tablica ta jest sortowana według priorytetów zadań. Dzięki takiemu zabiegowi łatwiej jest w trakcie rysowania ocenić położenie zadania, jak również uniknąć nienaturalnego przeskoku, gdyby pomiędzy wartościami priorytetów zadań był przeskok.

Aby zachować czytelność obrazu i możliwość wczytania większej liczby zadań do analizy, zadania są dzielone w partie, ilość zadań w partii jest określana przez użytkownika.

Wartości: szerokości obrazu, odstepu pionowego i poziomego pomiędzy zadaniami, liczby zadań w partii, wysokości bloku zadania, minimalnej oraz maksymalnej jasności są ustalane z góry przez użytkownika programu.

Schemat rysowania obrazu do formatu TEX został przedstawiony w kodzie 6.1.

Kod 6.1. Schemat generowanego pliku z wizualizacją zadań

```
1 \begin{picture}(w_{obrazu},h_{obrazu}) % rozpoczęcie obrazu
2 \setlength\fbboxsep{0pt}
3 % rysowanie zadania
4 \put(x_{zadania},y_{zadania}){\colorbox{kolor}{\framebox{w_{zadania},h_{zadania}}{nr_{zadania}}}}
5
6 \end{picture} % zakończenie obrazu
```

Algorytm 6.1. Generowanie obrazu.**Dane:** zadania w pliku tekstowym.

Wstępnie przetwórz zadania.

Dla wszystkich zadań wykonaj:**Jeżeli** obecne zadanie rozpoczyna nową partię zadań **to:**

Oblicz wysokość i szerokość nowego obrazu. [1]

Rozpocznij nowy obraz.

Narysuj oś OX.

Zaznacz wartości na osi.

koniec.

Oblicz rozmiary zadania. [2]

Oblicz położenie zadania na diagramie. [3]

Oblicz jasność zadania. [4]

Narysuj zadanie na diagramie.

Jeżeli obecne zadanie kończy daną partię **to:**

Zakończ obraz.

koniec.**koniec.****Jeżeli** partia nie została zakończona **to:**

Zakończ obraz.

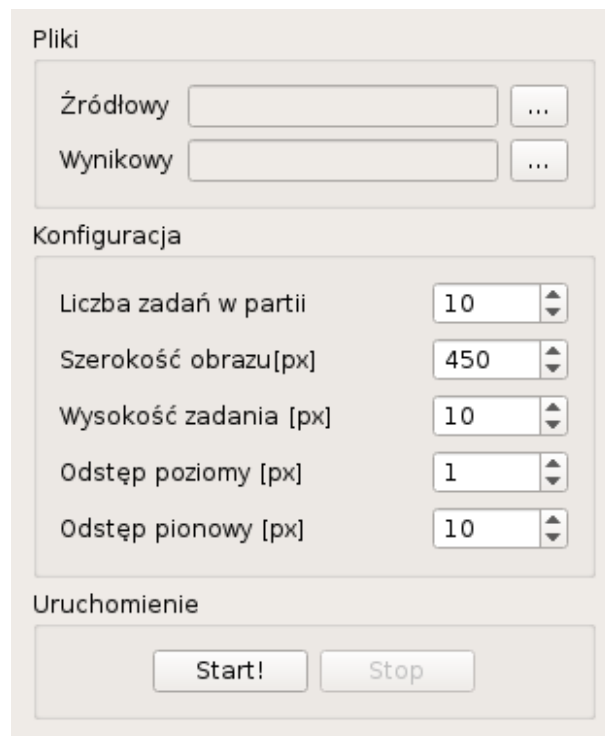
koniec.

Tabela 6.1. Zależności do wizualizacji zadań.

Nazwa		Wzór
rozmiar obrazu	[1]	$h_{\text{obrazu}} = (2 + m_{\text{priorytetów}}) \cdot h_{\text{odstępu}} + m_{\text{priorytetów}} \cdot h_{\text{zadania}}$ $w_{\text{obrazu}} = \text{const}$
rozmiar zadania	[2]	$h_{\text{zadania}} = \text{const}$ $w_{\text{zadania}} = \frac{w_{\text{obrazu}}}{n_{\text{max_zadań}} + 1}$
położenie zadania	[3]	$y_{\text{zadania}} = h_{\text{odstępu}} + j_{\text{idx_priorytetu}} (h_{\text{zadania}} + h_{\text{odstępu}})$ $x_{\text{zadania}} = x_{\text{poprz_zadania}} + w_{\text{zadania}} + w_{\text{odstępu}}$
jasność zadania	[4]	$k_{\text{jasność}} = k_{\text{min_jasność}} + \frac{k_{\text{max_jasność}}}{n_{\text{zadań}}} \cdot i_{\text{idx_zadania}}$

6.2. Wizualizacja przebiegu zadań

Po uruchomieniu programu pojawia się okno główne programu przedstawione na rys. 6.1.



Rysunek 6.1. Okno główne programu wizualizacji po uruchomieniu.

- o *Pliki*

- **Źródłowy** pozwala na wybór logu z danymi zadań. Po kliknięciu pojawi się okno wyboru pliku. Plik źródłowy powinien zawierać w kolejnych liniach wpisy dotyczące zadań:

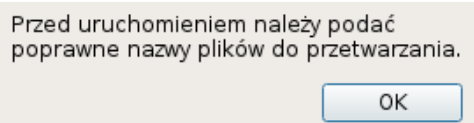
```
czas_s  czas_ms  nr_tiku  id_zadania  priorytet_zadania
```

- **Wynikowy** umożliwia na wybór pliku docelowego, gdzie ma być wygenerowany obraz w formacie TEX. Po kliknięciu pojawi się okno wyboru położenia i nazwy pliku. Po wybraniu istniejącego już pliku, należy potwierdzić jego nadpisanie.

- o *Konfiguracja*

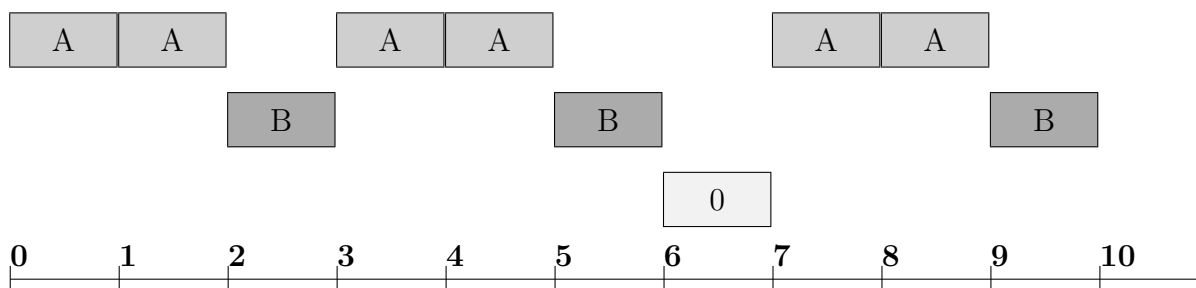
- **Liczba zadań w partii** pozwala na wybór liczby zadań w pojedynczej partii.
- **Szerokość obrazu** umożliwia dobranie szerokości obrazu wynikowego w pikselach.
- **Wysokość zadania** specyfikuje wysokość pojedynczego bloku z zadaniem w pikselach.

- **Odstęp poziomy** ustawia odstęp w poziomie pomiędzy zadaniami w pikselach.
- **Odstęp pionowy** ustawia odstęp w pionie pomiędzy zadaniami w pikselach.
- *Uruchomienie*
 - **Start** uruchamia przetwarzanie. W przypadku braku nazwy pliku lub nieprawidłowego pliku źródłowego wyświetlony zostanie komunikat przedstawiony na rys. 6.2.
 - **Stop** zatrzymuje dalsze przetwarzanie, dane które już zostały przetworzone, są zapisane w pliku.

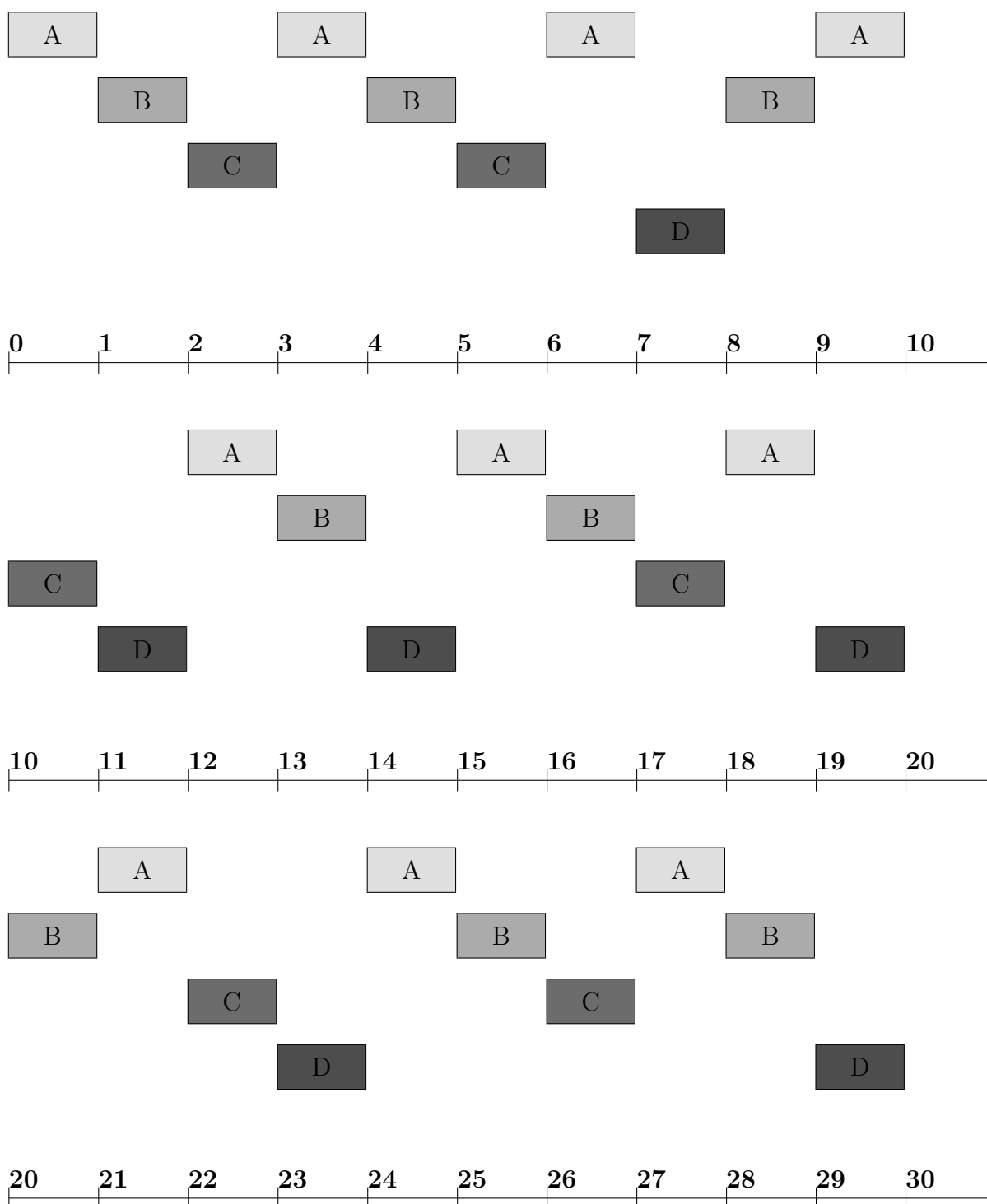


Rysunek 6.2. Komunikat nieprawidłowości w plikach w programie wizualizacji

Na rys. 6.3. oraz rys. 6.4. przedstawiono przykładowe wyniki działania programu.



Rysunek 6.3. Przykładowy harmonogram dla dwóch zadań, 10 tików trwania



Rysunek 6.4. Przykładowy harmonogram dla czterech zadań, 30 tików trwania

6.3. Implementacja kluczowych elementu systemu

6.3.1. Program główny

Kod 6.2. przedstawia przykładowy program główny z programem użytkownika `jobA()`, tworzący zadania i uruchamiający system.

Kod 6.2. Przykładowy program główny

```

1  #include "OS_core/inc/OS_core.h"
2  #include "OS_core/inc/OS_port.h"
3
4  void jobA() {
5      while( 1 == 1 ) {
6          OS_LOG( "task A" );
7          OS_sleep();
8      }
9  }
10
11 int main() {
12     OS_init_system( );
13     OS_create_task( jobA, "JobA", 0, 3 );
14     OS_start_scheduling( OS_SCHED_RM );
15     while( 1 == 1 ) {
16         ; /* NOOP */
17     }
18     return 0;
19 }
```

6.3.2. Pamięć systemowa

Strukturę pamięci przedstawiono w kodzie 6.3. Wskaźnik `malloc_memory` oznacza pierwszy bajt w pamięci, który nie jest zajmowany przez struktury systemowe. Pamięć zaczynająca się w tym miejscu może być wykorzystywana jako sterta dla podprogramów użytkownika. Podanie w tym miejscu prawdziwej wielkości pamięci nie jest potrzebne, ponieważ język C nie sprawdza zakresów tablic. Rozmiar pamięci do użycia będzie wyliczony na podstawie konfiguracji portu i adresów w strukturze.

Kod 6.3. Pamięć systemowa

```

1  struct OS_mem {
2      struct OS_task_info tasks[OS_MAX_TASKS_CNT];
3      unsigned int tasks_cnt;
4      struct OS_scheduler_API scheduler;
5      unsigned int current_tick;
6      unsigned int current_task;
7      char * free_mem;
8      char malloc_memory[1];      /* ostatni w strukturze */
9  };
```

6.3.3. Jądro systemu

W kodzie 6.4. dokonana jest inicjalizacja systemu. Najpierw wykonywana jest inicjalizacja sprzętowa, potem inicjalizacja pamięci. Dalej tworzone jest zadanie bezczynności i uruchamiany timer. Początek pamięci systemowej (`OS_MEM_START`) jest makrem implementowanym w porcie. Z kolei `OS_TICK` jest jedną z danych konfiguracji portu, wskazuje ona ile milisekund zajmuje tik systemowy.

Kod 6.5. przedstawia tworzenie zadań w systemie. Przy tworzeniu należy podać uchwyt funkcji wykonywanej przez zadanie, nazwę zadania, priorytet oraz okres zadania (w tikach).

W kodzie 6.6. przedstawione jest planowanie zadań. Przed wywołaniem pierwszego tik, należy wyłączyć przerwania i wywołać tik zegara.

Inicjalizacja pamięci, przedstawiona w kodzie 6.7. polega zainicjowaniu odpowiednich pól struktury pamięci systemu.

Kod 6.4. Inicjalizacja systemu

```

1 void OS_init_system() {
2     _OS_INIT_HARDWARE();
3     _OS_init_mem();
4     struct OS_mem * mem = OS_MEM_START;
5     OS_create_task( _OS_idle_task, "idle", 0, 0 );
6     _OS_TIMER_START( _OS_timer_tick, OS_TICK );
7 }

```

Kod 6.5. Tworzenie zadań

```

1 int OS_create_task( void (*fun)( void ), const char * fun_name,
2     unsigned int priority, unsigned int period ) {
3     struct OS_mem * mem = OS_MEM_START;
4     int ret = 0;
5
6     OS_interrupts_disable();
7
8     if( mem->tasks_cnt < OS_MAX_TASKS_CNT ) {
9         _OS_CONTEXT_INITIALISE(
10             &(mem->tasks[mem->tasks_cnt].stack), fun );
11         mem->tasks[mem->tasks_cnt].priority = priority;
12         mem->tasks[mem->tasks_cnt].period = period;
13         mem->tasks[mem->tasks_cnt].state = OS_READY;
14         mem->tasks[mem->tasks_cnt].next_tick =
15             mem->current_tick + OS_TICK;
16
17         OS_strcpy( fun_name, mem->tasks[mem->tasks_cnt].name );
18
19         ++( mem->tasks_cnt );
20         ret = mem->tasks_cnt;
21     }
22
23     OS_interrupts_enable();
24     return ret;
25 }

```

Kod 6.6. Rozpoczęcie planowania

```

1 void OS_start_scheduling( enum OS_sched_type sched_type ) {
2     struct OS_mem * mem = OS_MEM_START;
3     OS_sched_init( sched_type, &( mem->scheduler ) );
4     OS_interrupts_disable();
5     _OS_timer_tick();
6 }

```

Kod 6.7. Inicjalizacja pamięci

```

1 int _OS_init_mem() {
2     struct OS_mem * mem = OS_MEM_START;
3     mem->tasks_cnt = 0;
4     mem->current_task = 0;
5     mem->free_mem = mem->malloc_memory;
6     mem->current_tick = 0;
7
8     return 1;
9 }

```

6.3.4. Czas systemowy

Obsługa przerwania zegarowego (kod 6.8.) polega na obliczeniu nowej wartości tików (z uwzględnieniem zawijania) oraz na wywołaniu planisty.

Kod 6.8. Obsługa przerwania zegarowego

```

1 void _OS_timer_tick() {
2     struct OS_mem * mem = OS_MEM_START;
3     unsigned int old_tick = mem->current_tick;
4     mem->current_tick = _OS_sum_ticks( old_tick, OS_TICK );
5
6     _OS_schedule();
7 }

```

6.3.5. Planista

Kod 6.9. przedstawia mechanizm wyboru zadania do uszeregowania i zmiany zadania aktualnie wykonywanego, jeżeli zachodzi taka potrzeba. Po wywołaniu algorytmu planującego i podaniu przez niego zadania, które ma być aktualnie wykonywane, jeżeli nie jest to obecnie wykonywane zadanie, następuje zmiana kontekstu i odblokowanie przerwań.

W kodzie 6.10. pokazana jest zmiana stanu zadania. Zadanie może być w trzech stanach: **OS_SLEEPING**, **OS_READY**, **OS_RUNNING**, czyli odpowiednio: uśpione, gotowe, uruchomione. Dla zadań utworzonych przez użytkownika, sprawdzane jest na podstawie tików, w którym ma znów wystąpić zadanie, czy już jest gotowe. Algorytm ten uwzględnia zawijanie tików.

W kodzie 6.11. przedstawiona została inicjalizacja oraz planowanie algorytmem częstotliwościowym. W pętli sprawdzane są zadania użytkownika, jeżeli żadne nie jest gotowe, to zaplanowane zostanie zadanie bezczynności.

Kod 6.12. pokazuje planowanie algorytmem terminowym. Podobnie jak w przypadku algorytmu częstotliwościowego, w pętli sprawdzane są jedynie zadania użytkownika. Jeżeli

zadanie jest gotowe, wyliczany jest jego termin. Uszeregowane jest zadanie o najbliższym terminie.

Kod 6.9. Planowanie i dyspozycja zadań

```

1 void _OS_schedule() {
2     struct OS_mem * mem = OS_MEM_START;
3     unsigned int task_to_sched;
4     unsigned int curr_task = mem->current_task;
5
6     _OS_set_task_state();
7
8     /* uruchomienie algorytmu szeregującego */
9     task_to_sched =
10         mem->scheduler.OS_schedule( mem->scheduler.data );
11
12     if( task_to_sched != curr_task ) { /* zmiana kontekstu */
13         mem->tasks[task_to_sched].next_tick = _OS_sum_ticks(
14             mem->tasks[task_to_sched].next_tick,
15             mem->tasks[task_to_sched].period*OS_TICK );
16
17         mem->current_task = task_to_sched;
18         mem->tasks[task_to_sched].state = OS_RUNNING;
19         if( mem->tasks[curr_task].state != OS_SLEEPING ) {
20             mem->tasks[curr_task].state = OS_READY;
21         }
22         _OS_CONTEXT_SWITCH_AND_ENABLE_INTERRUPTS(
23             &( mem->tasks[task_to_sched].stack ),
24             &( mem->tasks[curr_task].stack ) );
25     } else { /* dalsze wykonywanie zadania */
26         OS_INTERRUPTS_ENABLE();
27     }
28 }

```

Kod 6.10. Ustalenie stanu gotowości zadania

```

1 void _OS_set_task_state() {
2     struct OS_mem * mem = OS_MEM_START;
3
4     /* tylko zadania użytkownika */
5     for( int i=1; i < mem->tasks_cnt; ++i ) {
6         if( mem->tasks[i].state == OS_SLEEPING ) {
7             if( _OS_cmp_ticks_is_task_ready(
8                 mem->current_tick,
9                 mem->tasks[i].next_tick,
10                 mem->tasks[i].period*OS_TICK
11             ) ) {
12                 mem->tasks[i].state = OS_READY;
13             }
14         }
15     }
16 }

```

Kod 6.11. Inicjalizacja i planowanie zadań algorytmem częstotliwościowym (RM)

```

1 void _OS_schedule_init_RM( struct OS_sched_RM_data * data ) {
2     struct OS_mem * mem = OS_MEM_START;
3
4     /* tylko zadania użytkownika */
5     for( int i=1; i < mem->tasks_cnt; ++i ) {
6         data->priority[i] = OS_MAX_PRIO - mem->tasks[i].period;
7     }
8 }
9
10 unsigned int _OS_schedule_RM( struct OS_sched_RM_data * data ) {
11     struct OS_mem * mem = OS_MEM_START;
12     unsigned int max_prio = 0;
13     unsigned int max_prio_idx = 0; /* indeks z. bezczynności */
14
15     for( int i=0; i < mem->tasks_cnt; ++i ) {
16         if( ( mem->tasks[i].state == OS_READY )
17             || ( mem->tasks[i].state == OS_RUNNING ) ) {
18             if( data->priority[i] > max_prio ) {
19                 max_prio_idx = i;
20                 max_prio = data->priority[i];
21             }
22         }
23     }
24     return max_prio_idx;
25 }

```

Kod 6.12. Planowanie zadań algorytmem terminowym (EDF)

```

1 unsigned int _OS_schedule_EDF( struct OS_sched_EDF_data * data ) {
2     struct OS_mem * mem = OS_MEM_START;
3     unsigned int min_diff = OS_MAX_TICKS;
4     unsigned int diff = OS_MAX_TICKS;
5     unsigned int min_tick_idx = 0; /* indeks z. bezczynności */
6
7     /* tylko zadania użytkownika */
8     for( int i=1; i < mem->tasks_cnt; ++i ) {
9         if( ( mem->tasks[i].state == OS_READY )
10            || ( mem->tasks[i].state == OS_RUNNING ) ) {
11             /* obliczenie terminu */
12             diff = _OS_diff_ticks(
13                 mem->current_tick, mem->tasks[i].next_tick );
14             if( diff < min_diff ) {
15                 min_tick_idx = i;
16                 min_diff = diff
17             }
18         }
19     }
20     return min_tick_idx;
21 }

```

6.3.6. Alokacja pamięci

Pamięć przydzielana jest liniowo, alokuje się kolejne bloki pamięci, bez możliwości zwracania. Zastosowane jest wyrównywanie przydzielonych bloków pamięci, decyduje o tym zmienna w konfiguracji portu `OS_MEM_PADDING`.

Kod 6.13. Alokacja pamięci

```

1 void * OS_malloc( unsigned char size ) {
2     struct OS_mem * mem = OS_MEM_START;
3     void * ret_mem;
4     ret_mem = mem->free_mem;
5     mem->free_mem += ( (size-1)/OS_MEM_PADDING + 1 ) * OS_MEM_PADDING;
6
7     return ret_mem;
8 }

```

6.4. Implementacja portu symulacyjnego na system Linux

6.4.1. Inicjalizacja sprzętowa

Inicjalizacja sprzętowa składa się z kilku etapów, inicjalizacji: pamięci systemowej, przerwań systemowych oraz timera. Aby zaimplementować pamięć systemową, odwzorowano strukturę pamięci współdzielony plik. Dodatkowo, na potrzeby portu stworzono dodatkową strukturę, zawierającą potrzebne dane do działania portu (adresy PID procesów, semafor). Włączenie i wyłączenie przerwań systemowych polega na zajęciu lub zwolnieniu semafora binarnego z biblioteki `pthread`. Aby semafor był dostępny dla różnych procesów należało go odpowiednio zainicjować. Inicjalizacja timera polegała na przypisaniu obsługi przerwania zegarowego do sygnału wysyłanego przez zegar (patrz podsekcja: 6.4.3.)

Kod 6.14. Inicjalizacja sprzętowa

```

1 void _OS_HARDWARE_INIT() {
2     /* inicjalizacja RAM */
3     char mem_file[] = "/tmp/OS_mem.XXXXXX";
4     int map_fd = mkstemp( mem_file );
5     ftruncate( map_fd, sizeof( struct OS_PORT ) );
6     PORT = mmap(
7         NULL, sizeof( struct OS_PORT ),
8         PROT_READ | PROT_WRITE, MAP_SHARED, map_fd, 0 );
9     port_linux_mem_start = &( PORT->mem );
10
11     PORT->processes_cnt = 0;
12     PORT->current_process = 0;
13
14     /* inicjalizacja przerwań systemowych */
15     pthread_mutexattr_t attr;
16     pthread_mutexattr_init( &attr );
17     pthread_mutexattr_setpshared( &attr, PTHREAD_PROCESS_SHARED );
18     pthread_mutexattr_settype( &attr, PTHREAD_MUTEX_RECURSIVE );
19     pthread_mutex_init( &PORT->interrupt_lock, &attr );

```

```

20
21     /* inicjalizacja timera */
22     if( signal( SIGRTMIN, _OS_TIMER_HANDLER ) == SIG_ERR ) {
23         OS_LOG( "[Error] Hardware init" );
24         exit( 1 );
25     }
26 }

```

6.4.2. Kontekst jądra

Kontekst jądra, jak zostało to szerzej opisane w sekcji 3.9. jest tworzony poprzez jądro Linuxa. W zaimplementowanym systemie każde zadanie to osobny proces systemu Linux. Inicjalizacja kontekstu zadania przedstawiona jest kodzie 6.15.. Zadania są tworzone poprzez polecenie `fork()`, ich numery PID są zapisywane w pamięci portu.

Zmiana kontekstu (kod 6.16.) polega na wysłaniu sygnału uruchomienia (`SIGCONT`) do nowego zadania oraz sygnału zatrzymania do siebie (`SIGSTOP`). Sygnał zatrzymania jest wysyłany do siebie, ponieważ przerwanie zegarowe jest uruchamiane z poziomu bieżącego zadania.

Kod 6.15. Inicjalizacja kontekstu

```

1 void _OS_CONTEXT_INITIALISE(
2     void * stack, void (*thread_fun)( void ) ) {
3     pid_t pid = fork();
4     if( pid ) { /* rodzic */
5         PORT->processes[PORT->processes_cnt].pid = pid;
6         PORT->processes[PORT->processes_cnt].stack = stack;
7         ++( PORT->processes_cnt );
8     } else { /* dziecko */
9         raise( SIGSTOP ); /* zatrzymanie siebie */
10        ( *thread_fun )();
11    }
12 }

```

Kod 6.16. Zmiana kontekstu

```

1 void _OS_CONTEXT_SWITCH_AND_ENABLE_INTERRUPTS(
2     void * stack_to_load, void * stack_to_save ) {
3     (void) stack_to_save;
4
5     pid_t pid_to_start = 0;
6     for( int i=0; i < PORT->processes_cnt; ++i ) {
7         if( PORT->processes[i].stack == stack_to_load ) {
8             pid_to_start = PORT->processes[i].pid;
9             PORT->current_process = i;
10            break;
11        }
12    }
13    if( kill( pid_to_start, SIGCONT ) == -1 ) { /* wznowienie */
14        OS_LOG( "[Error] Context switch" );
15        exit( 1 );
16    }

```



```

17     OS_INTERRUPTS_ENABLE();
18     raise( SIGSTOP ); /* zatrzymanie siebie */
19 }

```

6.4.3. Przerwania zegarowe

Timer został zaimplementowany jako osobny proces, aby nie bazować na mechanizmie systemu gospodarza, utworzenie i jego działanie przedstawia kod 6.17.

Jako pierwszy argument wywołania podawany jest uchwyt na funkcję obsługującą przerwanie (`void * timer_function(void)`), zapisywany potem w strukturze portu. Drugim argumentem wywołania jest czas trwania tiku w milisekundach (`unsigned ↵ ↵ int ticks_ms`). W nieskończonej pętli, proces timera po odmierzeniu tiku generuje sygnał SIGRTMIN do obecnie wykonywanego zadania.

Kod 6.17. Uruchomienie timera

```

1 void _OS_TIMER_START( void (*timer_function)( void ),
2     unsigned int ticks_ms ) {
3
4     pid_t pid = fork();
5     if( !pid ) { /* dziecko */
6         PORT->timer_function = timer_function;
7         while( 1 == 1 ) { /* pętla timera */
8             usleep( ticks_ms * 1000 );
9             OS_LOG( "Timer timeout" );
10            OS_INTERRUPTS_DISABLE();
11            pid_t pid = PORT->processes[PORT->current_process].pid;
12
13            if( kill( pid, SIGRTMIN ) == -1 ) {
14                OS_LOG( "[Error] Timer signal" );
15                exit( 1 );
16            }
17        }
18    }
19 }

```


Rozdział 7.

Podsumowanie

W pracy przedstawiono prototyp systemu czasu rzeczywistego. Opisano rolę planisty w takim systemie, algorytmy tworzenia harmonogramu oraz niektóre zagrożenia i niebezpieczne zjawiska mogące się pojawić w takim systemie. Powstał projekt systemu czasu rzeczywistego, dostarczający podstawowych usług, jak uruchamianie zadań, planowanie, zmienianie kontekstu, blokowanie i odblokowywanie przerwań. Za pomocą zaprojektowanego systemu zbadano własności dwóch algorytmów szeregowania: częstotliwościowego i terminowego.

7.1. Plany rozwoju

W projekcie nie zaimplementowano dodatkowych usług systemowych synchronizacji i komunikacji, jak semaforey, kolejki komunikatów. Z tego względu nie można było też sprawdzić działania systemu i algorytmów szeregowania zadań zależnych. Do planów rozwojowych projektu można też włączyć przeniesienie systemu na inne architektury.

7.2. Wnioski

Projekt systemu oraz jego port na system Linux powstał w celu badania systemu czasu rzeczywistego. Jednakże, aby rozwijać oprogramowanie, zwykle pisze się port, który nie działa w trybie rzeczywistym (na przykład port systemu FreeRTOS na system Linux [RTE14]). Wtedy, do każdego zadania przypisuje się wątki użytkownika, które działają we wspólnej pamięci adresowej. Spełnienie warunku pracy w trybie rzeczywistym testuje się dopiero na platformie docelowej. Takie rozwiązanie ułatwia pracę nad oprogramowaniem, dostarcza więcej narzędzi do testowania [Lab02].

Ze względu na specyfikę systemu Linux, zadania były testowane na luźnym czasie. Tik zegarowy następował co sekundę, aby zmiany kontekstu nie były zaburzane przez wywołania systemowe.

W zaimplementowanym systemie, algorytmy szeregowania częstotliwościowego i terminowego zaobserwowano właściwości zgodne z literaturą [Buz97], [LO12], [Kop11], [SRL90].

- o Harmonogram stworzony algorytmem częstotliwościowym może w pewnych sytuacjach nie spełnić wymaganych terminów, a szeregowanie terminowe jest w stanie taki harmonogram utworzyć. Taka sytuacja wystąpiła, gdy współczynnik wykorzystania procesora U był większy niż 69% ale mniejszy niż 100%. Wtedy zadanie o najniższym priorytecie nie spełniło swojego terminu wykonania.

- Istnieją też sytuacje, kiedy można utworzyć harmonogram za pomocą algorytmu częstotliwościowego spełniające terminy, nawet gdy $U = 1$. Zaszło to wtedy, kiedy okresy zadań były harmoniczne względem siebie.
- W przypadku przeciążenia systemu zadaniami, gdy harmonogram tworzony był za pomocą algorytmu częstotliwościowego, to zadania o najniższym priorytecie nie dotrzymywały terminów. W przypadku algorytmu szeregowania terminowego, na procesorze wykonywane były najpierw zadania, które miały najbliższy termin zakończenia. Najwyższy priorytet miały zadania których termin już minął. Skutkowało to lawinowym niedotrzymywaniem terminów.

Na tej podstawie można wyciągnąć wniosek, że planista częstotliwościowy daje większy determinizm działania w porównaniu do planisty terminowego. Takie własności tłumaczą, dlaczego powszechnie używane systemy czasu rzeczywistego jak FreeRTOS czy Micrium MicroC/OS-II wykorzystują ten rodzaj planowania albo wręcz wymagają ręcznego ustawienia priorytetów zadaniami. Determinizm, w przypadku rygorystycznych systemów czasu rzeczywistego jest bardzo istotnym aspektem działania. Pomimo, że taka polityka nakłada dodatkowe ograniczenia na projekt systemu, to pozwala ona, np. w przypadku awarii podjąć odpowiednie działania.

Bibilografia

- [Buz97] G. C. Buzzaro: *Hard real-time computing systems. Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, Pisa 1997.
- [Bro04] H. Broeders: *2014 Embedded market study. Then, now: what's next?* UBM Tech 2014, <http://bd.eduweb.hhs.nl/es/2014-embedded-market-study-then-now-whats-next.pdf>, [do-step: 02.09.2015 r.].
- [JRZ10] L. Jie, G. Ruifeng, S. Zhinxiang: *The research of scheduling algorithms in real-time system*. 2010 International Conference on Computer and Communication Technologies in Agriculture Engineering, 2010.
- [Kop11] H. Kopetz: *Real-time systems. Design principles for distributed embedded applications*. Springer, Wien 2011, 2nd edition.
- [Lab02] J. J. Labrosse: *MicroC/OS-II. The real-time kernel*, CMP Books, Kansas 2002, 2nd Edition.
- [LO12] P. A. Laplante, S. J. Ovaska: *Real-time systems design and analysis. Tools for the practitioner*. IEEE Press 2012, 4th edition.
- [LL73] C. L. Liu, J. W. Layland: *Scheduling algorithms for multiprogramming in hard real-time environment*. Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973.
- [LW83] J. Leung, J. W. Whithead: *On the complexity of fixed priority scheduling of periodic real-time operating system*. Performance Evaluation 2(4), 1982.
- [Lov10] R. Love: *Linux kernel development*. Pearson Education, 2010.
- [Mok93] A. K. Mok: *Fundamental design problems of distributed systems for the hard real time environment*. PhD Thesis, Massachusetts Institute of Technology.
- [SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky: *Priority inheritance protocols: An approach to real-time synchronization*. IEEE Transaction on computers, vol. 39 NO. 9, September 1990.
- [SR04] J. A. Stankovic, R. Rajkumar: *Real-time operating systems*. Kluwer Academic Publishers, 2004.
- [RTE14] Real Time Engineers Ltd.: *Free RTOS*, <http://www.freertos.org>, [do-step: 07.09.2015 r.].

Spis rysunków

2.1. Schemat interakcji systemu czasu rzeczywistego	5
3.1. Schemat funkcjonalności systemu	16
3.2. Schemat działania planowania w systemie	18
5.1. Wizualizacja uszeregowania zadań dla algorytmu RM dla 1. testu	28
5.2. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 1. testu	28
5.3. Wizualizacja uszeregowania zadań dla algorytmu RM dla 2. testu	29
5.4. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 2. testu	30
5.5. Wizualizacja uszeregowania zadań dla algorytmu RM dla 3. testu	31
5.6. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 3. testu	31
5.7. Wizualizacja uszeregowania zadań dla algorytmu RM dla 4. testu	32
5.8. Wizualizacja uszeregowania zadań dla algorytmu EDF dla 4. testu	33
6.1. Okno główne programu wizualizacji po uruchomieniu.	37
6.2. Komunikat nieprawidłowości w plikach w programie wizualizacji	38
6.3. Przykładowy harmonogram dla dwóch zadań, 10 tików trwania	38
6.4. Przykładowy harmonogram dla czterech zadań, 30 tików trwania	39

Spis algorytmów

3.1. Program główny	16
3.2. Inicjalizacja systemu	16
3.3. Tworzenie zadania	17
3.4. Dodawanie tików zegarowych	18
3.5. Przerwanie zegarowe	18
3.6. Włączenie planisty	19
3.7. Dynamiczna alokacja pamięci	19
3.8. Inicjalizacja kontekstu w porcie na system Linux x86	21
3.9. Zmiana kontekstu w porcie na system Linux x86	21
3.10. Uruchomienie i działanie timera w porcie na system Linux x86	21
4.1. Inicjalizacja planisty	23

4.2. Sprawdzenie gotowości zadania.	24
4.3. Inicjalizacja szeregowania częstotliwościowego (RM)	24
4.4. Szeregowanie częstotliwościowe (RM)	25
4.5. Szeregowanie terminowe (EDF)	25
6.1. Generowanie obrazu.	36

Spis tablic

5.1. Zbiór zadań w 1. teście	27
5.2. Zbiór zadań w 2. teście	29
5.3. Zbiór zadań w 3. teście	30
5.4. Zbiór zadań w 4. teście	31
6.1. Zależności do wizualizacji zadań.	36

Spis kodów

6.1. Schemat generowanego pliku z wizualizacją zadań	35
6.2. Przykładowy program główny	40
6.3. Pamięć systemowa	40
6.4. Inicjalizacja systemu	41
6.5. Tworzenie zadań	41
6.6. Rozpoczęcie planowania	42
6.7. Inicjalizacja pamięci	42
6.8. Obsługa przerwania zegarowego	42
6.9. Planowanie i dyspozycja zadań	43
6.10. Ustalenie stanu gotowości zadania	43
6.11. Inicjalizacja i planowanie zadań algorytmem częstotliwościowym (RM) . . .	44
6.12. Planowanie zadań algorytmem terminowym (EDF)	44
6.13. Alokacja pamięci	45
6.14. Inicjalizacja sprzętowa	45
6.15. Inicjalizacja kontekstu	46
6.16. Zmiana kontekstu	46
6.17. Uruchomienie timera	47