Notatki do egzaminu – **Systemy Czasu Rzeczywistego (SCR)**

> **UWAGA:** Poniższe notatki łączą w sobie treści z trzech wykładów (1, 2 i 3). Znajdziesz tu **rozszerzone wyjaśnienia** w prostym języku, **przykłady** oraz **tabele**, które pomogą Ci lepiej zrozumieć zagadnienia związane z systemami czasu rzeczywistego.
> Dodatkowo **najważniejsze** fragmenty są **pogrubione**.
> Używaj spisu treści, by łatwo nawigować między sekcjami.

Spis treści

Wykład 1: Wprowadzenie do SCR

1.1. Definicja systemu czasu rzeczywistego

System czasu rzeczywistego (ang. *Real-Time System* – RTS) to komputerowy system, który **musi reagować** na zdarzenia z otoczenia (np. sygnały z czujników) **w ściśle określonym czasie**.
Jeśli reakcja będzie zbyt wolna, może to prowadzić do nieprzewidywalnych i niepożądanych skutków, w tym krytycznych błędów.

> **Przykład**
>
> - **System kontroli lotu**: w samolocie dane z czujników (wysokość, prędkość) muszą być przetwarzane natychmiast, by wprowadzać poprawki w sterowaniu.

1.2. Czas reakcji

- **Czas reakcji** (ang. *Response Time*) to **odstęp czasu** między wystąpieniem zdarzenia w systemie (np. przerwania) a momentem, w którym system podejmuje działanie w odpowiedzi na to zdarzenie.
- Jest kluczową miarą jakości systemu czasu rzeczywistego.

> **Ważne**: Czas reakcji musi być **deterministyczny**, czyli znany i gwarantowany z góry, np. 10 ms, a nie „w okolicach 10 ms".

1.3. Rodzaje systemów SCR

1. **Hard Real-Time (twarde)**
    - Opóźnienie powoduje **katastrofę**.
    - **Przykład**: System kontroli reaktora jądrowego, system sterowania ABS w samochodach.
2. **Firm Real-Time (solidne)**
    - Wynik spóźniony staje się bezużyteczny, ale nie powoduje natychmiastowej katastrofy.
    - **Przykład**: Aplikacje finansowe obsługujące transakcje w konkretnym czasie.
3. **Soft Real-Time (miękkie)**
    - Opóźnienie **nie niszczy** całego procesu, ale obniża jego jakość.

- **Przykład**: System transmisji wideo (lag w streamingu obniża komfort oglądania, ale go nie uniemożliwia).

1.4. Funkcja zysku (Profit Function)

- Określa, **jaką korzyść** (zysk) przynosi zakończenie zadania w danym przedziale czasowym.
- Jeśli zadanie nie zostanie ukończone do pewnego czasu – zysk może być zerowy lub nawet ujemny.

> **Prosty wykres**:
> Wyobraź sobie, że oś X to czas, a oś Y to wartość zysku. Im dalej w czasie, tym zysk spada.

1.5. Projektowanie i implementacja SCR – wyzwania

1. **Reakcja na zdarzenia zewnętrzne** – krótki czas na obsługę.
2. **Synchronizacja i komunikacja zadań** – wiele zadań jednocześnie współpracuje.
3. **Spełnianie ograniczeń czasowych** – algorytmy planowania muszą uwzględniać priorytety i terminy.

> **Przykład**: W systemie kontroli przemysłowej (sterownik PLC) w fabryce – program musi w określonym czasie pobrać dane z czujników, przetworzyć je i wysłać sygnały sterujące do maszyn.

1.6. Cechy systemów SCR

1. **Zależność od otoczenia**
   - System ciągle śledzi sygnały zewnętrzne.
2. **Współbieżność**
   - Równoczesne wykonywanie się wielu zadań/procesów.
3. **Punktualność**
   - Reakcje muszą być **na czas**.
4. **Ciągłość działania**
   - System powinien pracować **bez przerw**.
5. **Przewidywalność**
   - Z góry wiadomo, w jaki sposób system zareaguje na zdarzenie (czasowo i funkcjonalnie).

1.7. Komunikacja i synchronizacja procesów

- **Komunikacja**: wymiana danych między procesami (lub wątkami).
- **Synchronizacja**: ustalanie kolejności dostępu do zasobów, by uniknąć konfliktów.

Modele komunikacji

1. **Pamięć dzielona (Shared Memory)**
   - Procesy współdzielą określoną przestrzeń pamięci.
2. **Przesyłanie komunikatów (Message Passing)**
   - **send** (nadaj) i **receive** (odbierz) – procesy wysyłają i odbierają komunikaty.

> **Tabela porównawcza**

Model komunikacji	Zalety	Wady
Pamięć dzielona	Szybka wymiana danych	Wymaga mechanizmów synchronizacji
Przesyłanie komunikatów	Łatwiejsza kontrola dostępu do danych	Może wymagać dodatkowego narzutu czasowego

1.8. Klasyczne problemy synchronizacji

1. **Producent i konsument**
    - Producent wytwarza dane i umieszcza w buforze.
    - Konsument pobiera dane z bufora.
    - **Problemy**: bufor może być pusty (konsument czeka) lub pełny (producent czeka).
2. **Czytelnicy i pisarze**
    - Wiele procesów czyta zasób, a tylko jeden pisze w danym momencie.
    - **Zasada**: kilku czytelników może czytać jednocześnie, ale pisarz musi mieć **wyłączność**.
3. **Filozofowie przy stole**
    - Każdy filozof potrzebuje 2 widelców, by jeść.
    - Gdy wszyscy jednocześnie podniosą jeden widelec, mogą się zablokować, bo żaden nie ma obu narzędzi.

> **Ćwiczenie praktyczne**: Spróbuj zaprojektować mechanizm semaforów dla problemu filozofów tak, by **uniknąć zakleszczenia**.

1.9. Problem zakleszczenia (Deadlock)

- **Zakleszczenie (deadlock)**: sytuacja, w której każdy z kilku procesów czeka na zasób (lub sygnał) od innego procesu i żaden nie może ruszyć dalej.
- **Przykład**: Dwa procesy, z których każdy zajmuje inny zasób i próbuje zająć zasób drugiego. Blokada jest wzajemna.

> **Najważniejsze warunki powstawania deadlocka** (tzw. Kołodziej Chandy'ego/Misry lub Coffmana):
>
> 1. Wzajemne wykluczanie
> 2. Przytrzymywanie i czekanie
> 3. Brak wywłaszczania
> 4. Cykliczne czekanie

1.10. Mechanizmy komunikacji między procesami (IPC)

1. **Pliki i blokady** – procesy współdzielą pliki, blokady chronią przed jednoczesnym zapisem.
2. **Sygnały** – proste powiadomienia (np. SIGINT).
3. **Potoki (pipes)** – jednokierunkowe kanały komunikacyjne.
4. **Kolejki komunikatów** – dane przesyłane w trybie FIFO.

5. **Semafory** – do kontroli dostępu do zasobów (np. sekcja krytyczna).
6. **Pamięć dzielona** – współdzielona przestrzeń danych.

1.11. Semafory

Semafor to mechanizm synchronizacji procesów oparty na liczniku, posiadający dwie operacje:

- **P (proberen)** – **opuszczenie** semafora, czyli dekrementacja licznika. Jeżeli licznik wynosi 0, proces się blokuje.
- **V (verhogen)** – **podniesienie** semafora, czyli inkrementacja licznika. Odblokowuje proces czekający w kolejce.

> **Przykład**:
>
> - Semafor binarny (wartość 0 lub 1) – może służyć do ochrony sekcji krytycznej.
> - Semafor liczący (wartość >1) – pozwala określić maksymalną liczbę procesów w sekcji krytycznej.

Wykład 2: Planowanie zadań i kolejki procesów

2.1. Kolejkowanie procesów

- **Kolejka zadań**: zawiera wszystkie procesy obecne w systemie.
- **Kolejka procesów gotowych**: procesy, które mogą być wykonywane w danej chwili (czekają na przydział CPU).
- **Kolejka urządzeń**: procesy oczekujące na zakończenie operacji We/Wy (np. dysk).
- **Kolejka procesów oczekujących na sygnał**: procesy, które czekają na zdarzenie (np. sygnał od innego procesu).

> **Ilustracja** (schemat blokowy kolejkowania):
>
> ```
> [Kolejka zadań] -> [Kolejka gotowych] -> [CPU] -> [Kolejka urządzeń] -> ...
> ```

2.2. Kontekst i przełączanie kontekstu

Kontekst procesu zawiera:

- Zawartość rejestrów,
- Licznik rozkazów,
- Informacje o pamięci (np. tablica stron),
- Dane o priorytetach.

Przełączanie kontekstu (context switch) ma miejsce, gdy system operacyjny zawiesza wykonywanie jednego procesu i rozpoczyna wykonywanie innego.

- **Czas przełączania** jest **stracony** z punktu widzenia procesów użytkownika, bo nic produktywnego się nie dzieje.

> **Metafora**: Wyobraź sobie, że proces to „osoba", a kontekst to jej „stan umysłu" (co pamięta). Przełączanie kontekstu to moment, gdy jedna osoba wychodzi, a inna wchodzi na scenę i musi sobie przygotować wszystkie rekwizyty.

2.3. Planowanie zadań (Scheduling) – ogólne pojęcia

1. **Tryb decyzji**:
    - **Wywłaszczeniowy**: proces może zostać przerwany przez inny (o wyższym priorytecie).
    - **Niewywłaszczeniowy**: proces wykonuje się do końca lub do momentu oddania procesora.
2. **Funkcja priorytetu**:
    - Określa, któremu procesowi przydzielić CPU w danym momencie.
3. **Reguła arbitrażu**:
    - Rozstrzyga kolejność w sytuacjach, gdy dwa procesy mają ten sam priorytet (np. FIFO).

2.4. Priorytety w systemach SCR

Priorytet to wartość liczbowa wskazująca ważność zadania:

- **Wyższy priorytet** → **większa szansa** na szybkie przydzielenie CPU.
- **Niższy priorytet** → **zadanie może czekać dłużej**.

> **Przykład**:
>
> - W systemie Linux:
>   - **RTPRIO** (1–99) – priorytety czasu rzeczywistego.
>   - **nice** (-20 do +19) – priorytety dla zwykłych zadań (im niższa liczba, tym wyższy priorytet).

2.5. Czasy oczekiwania i cyklu przetwarzania

- **Czas oczekiwania (T_o)**: ile dany proces spędził w kolejce gotowych, zanim zaczął się wykonywać.
- **Czas cyklu (T_p)**: łączny czas od momentu zgłoszenia procesu w systemie do zakończenia jego działania (obejmuje czas oczekiwania + czas wykonania).

> **Wzory**:
> $$T_{o} = \frac{\sum (\text{czasy oczekiwania wszystkich procesów})}{\text{liczba procesów}}$$
> $$T_{p} = T_{o} + \text{czas obsługi}$$

2.6. Miary efektywności

1. **Efektywność jednego procesu**:
   [\eta = \frac{\text{czas obsługi}}{\text{czas całkowity w systemie}}]
2. **Średnia efektywność**: średnia wartość (\eta) dla wszystkich procesów.

2.7. Algorytmy szeregowania zadań

Bez wywłaszczenia

1. **FCFS (First Come First Served)**
   - Procesy obsługiwane w kolejności zgłoszeń.
   - **Zaleta**: prostota.
   - **Wada**: może powodować długi czas oczekiwania dla później przychodzących zadań.
2. **SJF (Shortest Job First)**
   - Najpierw wykonywane są najkrótsze zadania.
   - **Wada**: trudność w przewidzeniu czasu trwania zadania.

Z wywłaszczeniem

1. **RR (Round Robin)**
   - Każdy proces dostaje CPU na krótki kwant czasu (np. 10 ms), następnie trafia na koniec kolejki.
   - **Zaleta**: dobra dla interakcyjnych systemów.
2. **SRT (Shortest Remaining Time)**
   - Wybiera zawsze zadanie o **najkrótszym pozostałym czasie**.
   - Może często przełączać kontekst, jeśli przyjdzie nowe, jeszcze krótsze zadanie.

2.8. Planowanie w Linux i Windows

Linux

- **SCHED_FIFO**: bez wywłaszczenia, gdy proces działa, nie jest przerywany.
- **SCHED_RR**: round-robin z wywłaszczeniem.
- **SCHED_OTHER** (zwykle CFS – Completely Fair Scheduler): domyślny dla procesów użytkownika.

Windows

- **Czas rzeczywisty**: wątki mają **stały** priorytet – wysoki i niezmieniany przez system.
- **Priorytet zmienny**: system może **dynamicznie** dostosowywać priorytety wątków (np. by zrekompensować długie oczekiwanie).

2.9. GPOS vs RTOS

Cecha	GPOS (General Purpose OS)	RTOS (Real-Time OS)

Cecha	GPOS (General Purpose OS)	RTOS (Real-Time OS)
Cel nadrzędny	Równomierna, sprawiedliwa obsługa	Deterministyczne reakcje
Obsługa priorytetów	Możliwe, ale często dynamiczne	**Ścisłe** przestrzeganie
Gwarancja czasu odpowiedzi	Brak	**Tak**, czas gwarantowany
Zastosowania	PC, serwery, urządzenia domowe	Branża medyczna, lotnicza, przemysł

2.10. Jądro i mikrojądro

- **Jądro** (kernel) systemu RTOS zawiera:
    1. **Planowanie** zadań (kto i kiedy dostaje CPU),
    2. **Dyspozycję** (przekazanie sterowania procesowi),
    3. **Komunikację i synchronizację** (mechanizmy IPC).
- **Mikrojądro**:
    - Bardzo zminimalizowane, zawiera tylko najważniejsze mechanizmy (planowanie, zarządzanie przerwaniami).
    - Dodatkowe usługi (np. system plików) są realizowane w **modułach zewnętrznych**.

2.11. Czas i jego odmierzanie

1. **Czas bezwzględny**: rzeczywista data i godzina (np. do obsługi zegara RTC).
2. **Zegar RTC (Real Time Clock)**: może działać nawet przy wyłączonym zasilaniu, podtrzymywany baterią.
3. **Timer**: generuje przerwania co pewien odcinek czasu (np. co 1 ms), co służy do odmierzania kwantów czasu w systemie.

2.12. Przykładowe systemy RTOS

1. **QNX Neutrino**
    - Architektura mikrojądra, duża skalowalność.
    - Popularny w systemach embedded (np. w przemyśle samochodowym).
2. **VxWorks**
    - Jedno z najpopularniejszych komercyjnych RTOS, stabilne i wydajne.
    - Stosowane w lotnictwie i automatyce przemysłowej.
3. **RTLinux**
    - Rdzeń czasu rzeczywistego + jądro Linux.
    - Dobre wsparcie dla systemów otwartoźródłowych.

4. **Windows CE**
    - Bardziej „soft real-time".
    - Stosowany w urządzeniach przenośnych i wbudowanych (np. handheldy).
5. **FreeRTOS**
    - Darmowy, otwartoźródłowy, popularny w mikrokontrolerach (ARM, AVR).
    - Mała liczba wymagań sprzętowych.

Wykład 3: Zaawansowane metody planowania (RMS, EDF) i inwersja priorytetów

3.1. Planowanie oparte na priorytetach

Planowanie priorytetowe: każde zadanie ma **liczbę** oznaczającą priorytet. Procesor **zawsze** przydziela czas zadaniu o najwyższym priorytecie.

> **Ważne**: Prawidłowe ustalenie priorytetów jest kluczowe, zwłaszcza w systemach *hard real-time*.

3.2. Priorytety statyczne i dynamiczne

- **Statyczne**: ustalone raz, np. przez programistę – **nie zmieniają się** w trakcie działania.
- **Dynamiczne**: system może je modyfikować na podstawie obciążenia, czasu oczekiwania itp.

3.3. Kolejki wielopoziomowe

Technika, gdzie mamy kilka **kolejek** o różnych priorytetach.

- **Procesy interakcyjne**: wysoka waga priorytetowa, np. planowane round-robin.
- **Procesy obliczeniowe** (zadania tła): niższa waga, np. planowane SJF.

> **Przykład**:
>
> - Kolejka 1 (wysoki priorytet, kwant 10 ms),
> - Kolejka 2 (średni priorytet, kwant 20 ms),
> - Kolejka 3 (niski priorytet, np. SJF).

3.4. Planowanie statyczne i dynamiczne

- **Statyczne**: plan jest przygotowany **przed** uruchomieniem (np. w systemach wbudowanych z dobrze znanymi zadaniami).
- **Dynamiczne**: priorytety przydzielane **w trakcie** działania, zmieniają się wraz z sytuacją (np. algorytm EDF).

3.5. Zadania okresowe

Zadania, które wykonują się **cyklicznie** co pewien okres (T).

- Np. odczyt czujnika co 10 ms, przetwarzanie i wysłanie wyniku dalej.
- **C** – czas wykonania zadania w każdym cyklu,
- **T** – okres (co ile czasu zadanie się powtarza).

3.6. Algorytm RMS (Rate Monotonic Scheduling)

Rate Monotonic to **statyczny** algorytm planowania, w którym:

- Zadania o **krótszym okresie (T)** dostają **wyższy priorytet**.
- **Założenia** (klasyczny model Liu & Layland):
    1. Wszystkie zadania są **okresowe**.
    2. Brak zależności (synchronizacji) między zadaniami.
    3. Procesor zawsze uruchamia zadanie o najwyższym priorytecie (brak blokad).
    4. **Priorytety są niezmienne** w czasie.

> **Przykład**:
> Zadanie A: (T_A = 5 \text{ ms}, C_A = 1 \text{ ms})
> Zadanie B: (T_B = 10 \text{ ms}, C_B = 2 \text{ ms})
>
> - Zadanie A ma mniejszy okres (5 ms), więc otrzyma priorytet wyższy niż B.
> - RMS zapewnia, że A będzie zawsze planowane przed B.

3.7. Współczynnik wykorzystania procesora (U)

Dla jednego zadania:
[U = \frac{C}{T}]

- **C** – czas wykonania
- **T** – okres

Dla (n) zadań:
[U_c = \sum_{i=1}^{n} \frac{C_i}{T_i}]

3.8. Granica szeregowalności RMS

Dla algorytmu RMS istnieje **teoretyczna granica**:
[U_c \le n \left(2^{\frac{1}{n}} - 1\right)]
Wraz ze wzrostem (n), wyrażenie dąży do **(\ln(2) \approx 0{,}693)**.

> **Interpretacja**:
>
> - Jeśli całkowite obciążenie procesora (suma (C_i / T_i)) jest **mniejsze** niż ok. 69.3%, system powinien dać radę wykonać wszystkie zadania w czasie.
> - Jeśli jest większe, RMS może nie gwarantować terminów.

3.9. Algorytm EDF (Earliest Deadline First)

Earliest Deadline First to **dynamiczny** algorytm, w którym:

- W danym momencie wybierane jest zadanie z **najbliższym terminem ukończenia** (tzw. deadline).
- Może przyznawać lub odbierać priorytet w zależności od tego, jak zadania zmieniają swoje terminy.

> **Twierdzenie**:
> Jeśli (\sum (C_i/T_i) \le 1), to EDF gwarantuje, że wszystkie zadania okresowe **zakończą się w terminie**.

3.10. Porównanie RMS i EDF

Cecha	RMS (statyczny)	EDF (dynamiczny)
Typ priorytetów	Stałe (wynikające z okresu)	Zmienne (od deadline)
Teoretyczna granica (U)	(\approx 0{,}693)	1 (100%)
Złożoność implementacji	Stosunkowo prosta	Bardziej złożona
Gwarancja terminów	Do ~69.3% wykorzystania CPU	Do 100% (przy pewnych warunkach)

3.11. Inwersja priorytetów

Sytuacja, w której **zadanie o wysokim priorytecie** jest blokowane przez zadanie o **niskim priorytecie**, bo to drugie trzyma zasób (np. semafor). Dodatkowo może wejść zadanie o priorytecie **średnim**, które „przeskakuje" w kolejce.

> **Przykład** (W-S-N):
>
> - **N** (niski priorytet) zajmuje semafor i jest wywłaszczone przez **S** (średni priorytet).
> - **W** (wysoki priorytet) próbuje zdobyć semafor, ale jest blokowane przez N.
> - System czeka na to, aż N odzyska procesor (co się nie dzieje szybko, bo S ciągle ma wyższy priorytet).

3.12. Dziedziczenie priorytetów

By **zminimalizować** inwersję priorytetów, stosuje się:

1. **Priority Inheritance (PI)**
   - Zadanie o niskim priorytecie **dziedziczy** (tymczasowo) priorytet najwyższego procesu, który może być blokowany, dopóki nie zwolni zasobu.
2. **Priority Ceiling (pułap priorytetów, ICPP)**
   - Każdy zasób ma **priorytet pułapu** – najwyższy możliwy priorytet spośród zadań, które mogą go używać.
   - Kiedy zadanie blokuje ten zasób, **zostaje podniesione** do poziomu „pułapu".

> **Przykład (Priority Inheritance)**:

> - Zadanie N (niski priorytet) zajmuje semafor.
> - Zadanie W (wysoki priorytet) też go chce.
> - N odzyskuje procesor z priorytetem W, by jak najszybciej dokończyć i zwolnić zasób.
> - Po zwolnieniu zasobu N wraca do swojego normalnego priorytetu.

Podsumowanie

1. **Systemy czasu rzeczywistego** są zaprojektowane tak, by gwarantować **terminowe** reakcje.
2. **Projektowanie** wymaga zrozumienia mechanizmów synchronizacji (semafory, kolejki, pamięć dzielona) i **unikania zakleszczeń**.
3. **Planowanie zadań** (RMS, EDF, priorytety statyczne i dynamiczne) jest kluczowe dla **dotrzymania** ograniczeń czasowych.
4. **Inwersja priorytetów** może być niebezpieczna i wymaga stosowania **mechanizmów dziedziczenia priorytetów**.
5. W praktyce systemy RTOS (np. **FreeRTOS**, **QNX**, **VxWorks**) wykorzystują opisane strategie, zapewniając **deterministyczne** działanie.