Notatki do Egzaminu z Podstaw Programowania Współbieżnego

> **UWAGA:**
>
> - Poniższe notatki są **obszerną** wersją materiałów do nauki.
> - Zawierają **dodatkowe wytłumaczenia**, **przykłady** i **omówienia**, aby ułatwić zrozumienie zagadnień.
> - Możesz je skopiować do pliku `.md` i wygenerować na ich podstawie PDF.

SPIS TREŚCI

1. Wstęp do Programowania Współbieżnego
2. Abstrakcja Współbieżności
    1. Model Komputera Współbieżnego (PRAM)
    2. Podstawowe Definicje i Pojęcia
    3. Rodzaje Programów
    4. Przełączanie Kontekstu (Context Switching)
3. Problemy Synchronizacyjne i Narzędzia Synchronizacji
    1. Problem Wzajemnego Wykluczania
    2. Algorytmy Rozwiązujące Problemy Synchronizacji
        - Algorytm Dekkera
        - Algorytm Piekarniany (Bakery Algorithm)
    3. Narzędzia Synchronizacji
        - Semafory
        - Mutexy
        - Monitory
    4. Problem 5 Filozofów
4. Programowanie Współbieżne w Systemie Linux
    1. Proces UNIX-owy
    2. Mechanizmy Komunikacji Międzyprocesowej (IPC)
    3. Wątki
    4. Planowanie (Scheduling) w Systemie Linux
5. Komunikacja Międzyprocesowa - Omówienie Szczegółowe
    1. Komunikacja Synchroniczna i Asynchroniczna
    2. Przykład Problemu Producenta i Konsumenta
6. Wprowadzenie do Obliczeń Równoległych
    1. Rodzaje Równoległości
    2. Modele Architektury Równoległej (SIMD, MIMD)
7. Dodatkowe Koncepcje i Praktyki
    1. Unikanie Zakleszczeń (Deadlock Avoidance)
    2. Wykrywanie i Usuwanie Zakleszczeń (Deadlock Detection & Recovery)
    3. Zagłodzenie (Starvation)
    4. Wyścigi Danych (Race Conditions)

5. Debugowanie Programów Współbieżnych
8. Podsumowanie
9. Najważniejsze Pojęcia

---

1. Wstęp do Programowania Współbieżnego

Programowanie współbieżne (ang. concurrent programming) zajmuje się problemami wynikającymi z **równoczesnego** wykonywania wielu obliczeń (np. przez wiele procesów lub wątków). Dzięki wykorzystaniu współbieżności możemy:

- Skrócić czas wykonania (poprzez **podział pracy** między wiele jednostek obliczeniowych).
- Zwiększyć **responsywność** aplikacji (np. w interfejsie graficznym można równolegle wykonywać zadania w tle).
- Symulować **systemy rozproszone** i naturalnie współbieżne (np. obsługa żądań w serwerach).

Korzyści i Wyzwania

- **Korzyści**: Lepsze wykorzystanie zasobów, możliwość obsługi wielu użytkowników na raz, przyspieszenie obliczeń.
- **Wyzwania**: Trudniejsza analiza poprawności (m.in. trzeba dbać o synchronizację), potencjalne błędy trudne do wykrycia (np. wyścigi danych, zakleszczenia).

---

2. Abstrakcja Współbieżności

Model Komputera Współbieżnego (PRAM)

PRAM (Parallel Random Access Machine) to **teoretyczny model** równoległego komputera, w którym:

- Istnieje wiele procesorów, pracujących **synchronicznie** (wspólny zegar).
- Wszystkie procesory komunikują się poprzez **wspólną pamięć**.
- Dostęp do pamięci ma (w teorii) **stały czas** – niezależnie od liczby procesorów (idealizacja).
- Jednoczesny zapis do tej samej komórki pamięci jest **niedozwolony** w najprostszym wariancie PRAM (lub wymaga dodatkowych założeń – np. wariant **CREW**: concurrent read, exclusive write).

Zalety PRAM

- Pozwala na **łatwą analizę** teoretyczną złożoności równoległej.
- Umożliwia proste wyobrażenie równoległego przetwarzania wielu danych.

Wady PRAM

- Nie uwzględnia **opóźnień komunikacji**, **asynchroniczności** rzeczywistych systemów i **hierarchii pamięci** (cache, RAM, dysk itp.).

Podstawowe Definicje i Pojęcia

- **Proces**: Uruchomiony program, posiadający **przestrzeń adresową**, rejestry, zasoby systemowe.

- **Wątek**: Lżejsza jednostka wykonania działająca w obrębie jednego procesu (współdzieląca z nim zasoby i pamięć).
- **Sekcja krytyczna**: Fragment kodu, w którym następuje dostęp do **wspólnego zasobu** (np. zmienne globalnej, pliku). W sekcji krytycznej musi być zapewnione **wzajemne wykluczanie**.

> **Pamiętaj**: Jeśli wiele wątków/ procesów jednocześnie modyfikuje wspólny zasób i nie jest to kontrolowane, mogą wystąpić **błędy danych** i niespójność.

Rodzaje Programów

1. **Sekwencyjny**: Kod wykonywany **kolejno**, przez jeden proces – nie występuje równoległość.
2. **Współbieżny (konkurencyjny)**: Składa się z co najmniej dwóch wykonujących się (logicznie lub fizycznie) równolegle podprogramów.
3. **Równoległy**: Specyficzny przypadek współbieżności, w którym **wiele jednostek obliczeniowych** faktycznie pracuje w tym samym momencie (np. różne rdzenie procesora).
4. **Instrukcja atomowa**: Instrukcja, której **nie da się** przerwać – w całości wykonywana między przełączeniami kontekstu (np. instrukcja `xchg` w assemblerze x86).

Przełączanie Kontekstu (Context Switching)

- **Definicja**: Proces, w którym system operacyjny wstrzymuje jeden proces/wątek i przełącza procesor do wykonywania innego.
- **Koszty**: Konieczne jest zapisanie stanu rejestrów, licznika rozkazów (PC), i odtworzenie stanu innego procesu.
- **Częstotliwość przełączeń** zależy od **algorytmu planowania** (scheduler). Zbyt częste przełączenia zwiększają narzut na wydajność.

> **Wniosek**: Im większa współbieżność, tym potencjalnie więcej przełączeń kontekstu, co może prowadzić do spadku efektywności, jeśli nie jest to odpowiednio zarządzane.

---

3. Problemy Synchronizacyjne i Narzędzia Synchronizacji

Problem Wzajemnego Wykluczania

- **Opis**: Wiele procesów chce jednocześnie korzystać z **wspólnego zasobu** (np. drukarki, pliku, zmiennej globalnej).
  Musimy zagwarantować, że w danym momencie **tylko jeden** proces może być w sekcji krytycznej.
- **Warunki** (wg Dijkstry):
  1. **Wzajemne wykluczanie (Mutual Exclusion)**: Nigdy dwa procesy jednocześnie w sekcji krytycznej.
  2. **Postęp**: Gdy nikt nie jest w sekcji krytycznej, a jakieś procesy chcą do niej wejść, to jeden z nich powinien móc to zrobić bez nieograniczonego opóźnienia.
  3. **Ograniczone czekanie**: Każdy proces czekający na sekcję krytyczną powinien w końcu do niej wejść (brak zagłodzenia).

Algorytmy Rozwiązujące Problemy Synchronizacji

a) Algorytm Dekkera

Jeden z **pierwszych** algorytmów (dla dwóch procesów) zapewniający wzajemne wykluczanie bez użycia dodatkowego sprzętu (tylko instrukcje sekwencyjne i zmienne).

Podstawowe założenia:

- Dwie zmienne logiczne: `chceP` i `chceQ` (informują, czy proces P/Q chce wejść do sekcji krytycznej).
- Zmienna `kolej`, która rozstrzyga priorytet.

Szkic działania:

1. Proces P ustawia `chceP = true` i ustawia `kolej = Q`, by dać „prawo do pierwszeństwa" procesowi Q.
2. P czeka w pętli, dopóki `chceQ == true` i `kolej == Q`.
3. Gdy wyjdzie z tej pętli, może wejść do sekcji krytycznej.
4. Po zakończeniu sekcji krytycznej ustawia `chceP = false`.

Dowody:

- **Bezpieczeństwo**: Nie ma sytuacji, że oba procesy jednocześnie wejdą do sekcji krytycznej, ponieważ w przypadku konfliktu `kolej` przyznaje priorytet jednemu procesowi.
- **Żywotność**: Każdy proces po pewnym czasie otrzyma dostęp do sekcji krytycznej (nie ma permanentnego blokowania, jeśli proces działa w sekcji reszty).

---

b) Algorytm Piekarniany (Bakery Algorithm)

Rozszerzenie idei do **N** procesów. Wyobraź sobie **piekarnię**, w której każdy pobiera **bilet** (liczbę) i ustawia się w kolejce rosnącej numeracji:

1. **Proces** pobiera bilet o wartości `1 + max(...ticket innych...)`.
2. Jeśli w dwóch procesach bilety są równe, decyduje **mniejszy identyfikator** procesu.
3. Po „obsłużeniu" (sekcja krytyczna) proces oddaje bilet (ustawiając `ticket[i] = 0`).

Zalety:

- Teoretycznie **zapewnia** wzajemne wykluczanie.
- Gwarantuje **brak zagłodzenia**, gdyż kolejność jest zdeterminowana numerami i ID procesów.

Wady:

- Numerki mogą **rosnąć w nieskończoność**.
- W praktyce wymaga **atomicznych** odczytów i zapisów pewnych zmiennych, co bywa trudne w realnych systemach.

Narzędzia Synchronizacji

a) Semafor

- **Semafor** to zmienna całkowita z dwoma operacjami *atomowymi*:

- `wait()` (lub `P()`): zmniejsza wartość semafora o 1; jeśli wartość jest **poniżej zera**, proces się blokuje.
- `signal()` (lub `V()`): zwiększa wartość semafora o 1; jeśli jacyś procesy są w kolejce, jeden z nich zostaje wznawiany.

Przykład Kod w C (POSIX):

```c
#include <semaphore.h>

sem_t sem;

int main() {
    sem_init(&sem, 0, 1); // semafor binarny (wartość początkowa = 1)

    // ...
    // wątek 1:
    sem_wait(&sem);
    // sekcja krytyczna
    sem_post(&sem);

    // ...
    return 0;
}
```

> **Uwaga**: Semafor może mieć wartość > 1, co oznacza, że może pozwolić na jednoczesny dostęp np. do kilku identycznych zasobów.

---

b) Mutexy

- **Mutex** (ang. *mutual exclusion*) to zwykle semafor binarny z regułą: tylko **posiadacz** mutexu może go zwolnić.
- Używane w bibliotekach wątków (np. `pthread_mutex_t` w POSIX).
- Operacje: `pthread_mutex_lock(&m)`, `pthread_mutex_unlock(&m)`.

Prosty przykład:

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* threadFunc(void* arg) {
    pthread_mutex_lock(&lock);
    // sekcja krytyczna
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

> **Przykład**: Kod powyżej zapewnia, że tylko jeden wątek na raz wykona instrukcje wewnątrz blokady.

---

c) Monitory

- **Monitor** łączy zmienne i operacje w jedną strukturę, która zapewnia automatyczne **wzajemne wykluczanie** podczas wywoływania metod.
- W języku Java – słowo kluczowe `synchronized`:

```
public synchronized void increment() {
    // ta metoda jest sekcją krytyczną
    counter++;
}
```

- Monitory często udostępniają **zmienne warunkowe** (ang. *condition variables*), aby proces/wątek mógł **czekać** wewnątrz monitora na określony stan.

Problem 5 Filozofów

- **Opis**: 5 filozofów siedzi przy stole z 5 widelcami (pomiędzy każdym dwoma filozofami znajduje się 1 widelec). Aby zjeść, filozof potrzebuje **2 widelców**.
- **Cel**: Nie dopuścić do sytuacji, w której każdy filozof chwyci jeden widelec i czeka na drugi → **deadlock**.

Proste rozwiązanie z semaforami

```
semaphore widelec[5]; // każdy widelec - semafor binarny

void filozof(int i) {
    while(true) {
        // myślenie
        wait(widelec[i]);             // podniesienie "lewego" widelca
        wait(widelec[(i + 1) % 5]);   // podniesienie "prawego" widelca

        // jedzenie (sekcja krytyczna)

        signal(widelec[i]);           // odłożenie "lewego" widelca
        signal(widelec[(i + 1) % 5]); // odłożenie "prawego" widelca
    }
}
```

> **Zauważ**: Ta prosta wersja może prowadzić do zakleszczenia, gdy wszyscy filozofowie jednocześnie podniosą lewy widelec i czekają na prawy.
> Rozwiązania unikają tego poprzez np. ograniczenie liczby filozofów, którzy mogą *jednocześnie* sięgać po widelce, lub przyjęcie hierarchii (filozof o niższym ID zawsze podnosi najpierw niżej ponumerowany widelec).

---

4. Programowanie Współbieżne w Systemie Linux

Proces UNIX-owy

- **Definicja**: Program w trakcie wykonania, posiadający **PID**, własną przestrzeń adresową, plik opisujący kontekst (np. rejestry, deskryptory plików).
- **Tworzenie nowych procesów**: `fork()`.
  - Proces macierzysty i proces potomny początkowo niemal identyczne (kopiowana przestrzeń adresowa).
  - `exec()` zastępuje kod procesu nowym programem.

Demony: Procesy działające w tle (np. serwer SSH, demon poczty), często bez terminala, zarządzane przez system init (np. `systemd`).

Mechanizmy Komunikacji Międzyprocesowej (IPC)

- **Pamięć współdzielona (Shared Memory)**: Dwa (lub więcej) procesy mapują ten sam obszar pamięci w swojej przestrzeni adresowej.
- **Potoki (Pipes)**: Jednokierunkowe kanały do przesyłania danych między procesem rodzicem a potomnym.
- **Kolejki komunikatów (Message Queues)**: Pozwalają wysyłać/odbierać wiadomości (rekordy).
- **Gniazda (Sockets)**: Komunikacja proces-proces, również po sieci (może być używana lokalnie przez `AF_UNIX`).

Wątki

- Wątki w Linux (np. biblioteka pthread) współdzielą:
  - Kod, segment danych, pliki, zasoby wewnątrz jednego procesu.
- Każdy wątek ma **własny stos**, rejestry i **licznik rozkazów**.
- Tworzenie wątków: `pthread_create(&tid, NULL, funkcja, arg)`.
- Zakończenie wątku: `pthread_exit(...)`.
- Synchronizacja wątków: mutexy, semafory, monitory, bariery (`pthread_barrier_t`), itd.

> **Zaleta**: Wątki są „lżejsze” od procesów, mają wspólną pamięć, co ułatwia dzielenie danych.
> **Wada**: Łatwiej o błędy wyścigów i zakleszczeń, bo współdzielenie pamięci następuje *domyślnie*.

Planowanie (Scheduling) w Systemie Linux

- **CFS (Completely Fair Scheduler)**: Domyślny algorytm planowania w nowszych jądrach Linux.
- **Zasady**:
  - Próba równego podziału czasu CPU (czasami z priorytetami).
  - Przydziela małe kwanty czasu różnym procesom.
- **Priorytety**: Procesy rzeczywiste (nicetime + priorytety) oraz procesy w czasie rzeczywistym (RT – *real-time*).
- **Polityki**:
  - *SCHED_OTHER* (domyślna),
  - *SCHED_FIFO*, *SCHED_RR* (czas rzeczywisty).

---

5. Komunikacja Międzyprocesowa - Omówienie Szczegółowe

Komunikacja Synchroniczna i Asynchroniczna

- **Synchroniczna** (ang. *blocking*): Nadawca czeka, aż odbiorca odbierze komunikat.
  - Prostota w analizie, ale może powodować **czekanie** (blokadę).
- **Asynchroniczna** (ang. *non-blocking*): Nadawca wysyła komunikat do bufora, nie czeka na odbiór.
  - Wymaga bufora, może występować problem z jego przepełnieniem.

Przykład Problemu Producenta i Konsumenta

Opis:

- Producent wytwarza dane (np. liczby, obiekty) i chce je umieszczać w **buforze**.
- Konsument pobiera dane z bufora.
- **Warunek**: Konsument nie może pobrać z **pustego** bufora, a producent nie może wstawić do **pełnego** bufora.

Implementacja z Semaforami

```
semaphore Puste = N;   // liczba wolnych miejsc
semaphore Pelne = 0;   // liczba zapełnionych miejsc
semaphore Mutex = 1;   // chroni sekcję krytyczną

int bufor[N];
int in = 0, out = 0;

void producent() {
    while(true) {
        int item = produceItem(); // generuj dane

        wait(Puste);  // sprawdź, czy jest wolne miejsce
        wait(Mutex);  // sekcja krytyczna (dostęp do 'bufor')

        bufor[in] = item;
        in = (in + 1) % N;

        signal(Mutex);
        signal(Pelne); // jedno miejsce więcej jest zapełnione
    }
}

void konsument() {
    while(true) {
        wait(Pelne); // czekaj, aż będzie coś w buforze
        wait(Mutex); // sekcja krytyczna

        int item = bufor[out];
        out = (out + 1) % N;
```

```
            signal(Mutex);
            signal(Puste); // zwolnij miejsce

            consumeItem(item);
        }
    }
```

> **Komentarz**:
>
> - `Puste` i `Pelne` pilnują zasobów (wolne vs. zajęte miejsca).
> - `Mutex` chroni **jednoczesny dostęp** do indeksów `in` i `out` oraz tablicy `bufor`.
> - Zapobiega wyścigowi na zmiennych tablicy.

---

6. Wprowadzenie do Obliczeń Równoległych

Obliczenia równoległe polegają na wykonywaniu programów na wielu jednostkach obliczeniowych (rdzeniach, procesorach) jednocześnie.

- **Cel**: Skrócić czas wykonania, zwiększyć przepustowość.
- **Architektury**: Wielordzeniowe CPU, klastery komputerów, GPU (karty graficzne) do obliczeń masowo równoległych.

Rodzaje Równoległości

1. **Równoległość danych (Data Parallelism)**: Tę samą operację wykonujemy na różnych porcjach danych (np. obróbka pikseli obrazu).
2. **Równoległość zadań (Task Parallelism)**: Różne zadania (funkcje, podprogramy) są wykonywane w tym samym czasie, często współdzieląc wyniki.

Modele Architektury Równoległej (SIMD, MIMD)

- **SIMD (Single Instruction, Multiple Data)**:
  - Jedna instrukcja wykonywana równolegle na wielu elementach danych.
  - Przykłady: SSE, AVX, GPU (obliczenia strumieniowe).
- **MIMD (Multiple Instruction, Multiple Data)**:
  - Każdy procesor/wątek ma własny zestaw instrukcji i dane.
  - Przykład: Współczesne wielordzeniowe CPU (każdy rdzeń może wykonywać inny kod).

---

7. Dodatkowe Koncepcje i Praktyki

Unikanie Zakleszczeń (Deadlock Avoidance)

- **Definicja zakleszczenia**: Sytuacja, w której kilka procesów/wątków czeka na zasób trzymany przez inny proces. Koło się zamyka i żaden proces nie może postąpić dalej.

- **Warunki wystąpienia (wg Coffmana)**:
    1. Wzajemne wykluczanie (mutual exclusion).
    2. Zatrzymanie i czekanie (hold and wait).
    3. Brak wywłaszczania (no preemption).
    4. Czekanie cykliczne (circular wait).

Strategie unikania:

- Zastosowanie **hierarchii zasobów**.
- Protokół Bankiera (Dijkstra) do przydziału zasobów tylko jeśli system pozostaje w stanie bezpiecznym.
- Ustalanie kolejności zamawiania zasobów (np. zawsze alokować w rosnącej kolejności numerów zasobów).

Wykrywanie i Usuwanie Zakleszczeń (Deadlock Detection & Recovery)

- **Wykrywanie**: System okresowo sprawdza, czy istnieje cykl oczekiwania procesów na zasoby.
- **Usuwanie**: Zabijanie jednego z procesów w cyklu (lub wywłaszczanie zasobów) i ponawianie prób, aż cykl się przerwie.

Zagłodzenie (Starvation)

- **Opis**: Proces czeka bardzo długo (lub w nieskończoność) na zasoby, podczas gdy inne procesy wielokrotnie przechodzą.
- Może wystąpić przy niesprawiedliwych algorytmach planowania lub przy priorytetach (proces z niskim priorytetem ciągle odkładany).

Rozwiązania:

- **Priorytety dynamiczne** (zwiększanie priorytetu procesu, który długo czeka).
- Mechanizmy gwarantujące **wprowadzenie** do sekcji krytycznej po określonym czasie.

Wyścigi Danych (Race Conditions)

- **Definicja**: Błąd, który pojawia się, gdy wynik obliczeń zależy od **kolejności przeplotu** (ang. *interleavings*) operacji wątków/procesów.
- **Rozwiązanie**: Synchronizacja (muteksy, semafory, itd.), aby nie dopuszczać do niekontrolowanego przeplatania.

Debugowanie Programów Współbieżnych

- **Trudność**: Błędy współbieżne często **nie powtarzają się** w przewidywalny sposób.
- **Narzędzia**:
    - Debuggery z funkcjami wątkowymi (np. GDB).
    - **Analiza statyczna** (wykrywanie wzorców niepoprawnej synchronizacji).
    - **Narzędzia do wykrywania wyścigów** (np. ThreadSanitizer).

Podsumowanie

Programowanie współbieżne i równoległe to **kluczowy obszar** w nowoczesnym oprogramowaniu. Aby efektywnie z niego korzystać, musimy:

1. **Rozumieć** mechanizmy systemu operacyjnego: procesy, wątki, IPC.
2. **Stosować** prymitywy synchronizacyjne: semafory, muteksy, monitory, bariery.
3. **Zapewnić** bezpieczeństwo i żywotność: unikać deadlocków, dbać o brak zagłodzenia.
4. **Testować** i debugować programy w różnych scenariuszach (bo błędy współbieżności często są trudne do wykrycia).

Najważniejsze Pojęcia

- **Współbieżność (Concurrency)**: Zdolność do wykonywania wielu zadań *logicznie* w tym samym czasie.
- **Równoległość (Parallelism)**: Fizyczne wykonywanie wielu zadań *równocześnie* (np. na wielu rdzeniach).
- **Sekcja krytyczna (Critical Section)**: Fragment kodu wymagający wyłącznego dostępu do zasobu wspólnego.
- **Wzajemne wykluczanie (Mutual Exclusion)**: Gwarancja, że tylko jeden wątek/proces naraz przebywa w sekcji krytycznej.
- **Zakleszczenie (Deadlock)**: Sytuacja, w której procesy/wątki oczekują na zasoby w cyklu i żaden nie może kontynuować pracy.
- **Zagłodzenie (Starvation)**: Proces (lub wątek) nigdy nie otrzymuje zasobów (lub nie wchodzi do sekcji krytycznej).
- **Wyścigi Danych (Race Condition)**: Niepoprawna sytuacja, gdy końcowy rezultat zależy od przypadkowego przeplotu wątków.
- **Planowanie (Scheduling)**: Decydowanie, który proces lub wątek otrzymuje czas procesora w danym momencie.
- **Algorytmy synchronizacji**: Dekker, Piekarniany, Peterson, itd.
- **Narzędzia synchronizacji**: Semafory, muteksy, monitory, bariery, zmienne warunkowe.
