Szkoła Gospodarstwa Wiejskiego
Wydział Zastosowań Informatyki i Matematyki

PARADYGMATY PROGRAMOWANIA

Jednostka 10 – Paradygmat współbieżny i równoległy

Kierunek: Informatyka, semestr 5
Prowadzący: dr inż. Krzysztof Malczewski

1. Cel i zakres zajęć

Celem zajęć jest:

- poznanie **paradygmatu współbieżnego i równoległego**,
- zrozumienie różnicy między **współbieżnością** (concurrency) a **równoległością** (parallelism),
- poznanie podstawowych mechanizmów: **wątki, procesy, synchronizacja**,
- zastosowanie tych mechanizmów w prostych programach,
- dostrzeżenie potencjalnych problemów: **sekcje krytyczne, wyścigi danych**.

Zakres tematyczny:

- modele współbieżności,
- wątki i procesy w Pythonie,
- komunikacja i synchronizacja,
- równoległe przetwarzanie danych,
- problemy i pułapki współbieżności.

---

2. Wprowadzenie teoretyczne

2.1 Współbieżność vs równoległość

- **Współbieżność** — program podzielony na wiele zadań, które mogą być wykonywane naprzemiennie (nawet na jednym procesorze).
  👉 Przykład: obsługa wielu klientów przez jeden serwer.
- **Równoległość** — rzeczywiste wykonywanie wielu zadań w tym samym czasie, np. na wielu rdzeniach CPU.
  👉 Przykład: dzielenie dużych danych między wiele wątków/procesów.

2.2 Modele współbieżności

- **Wątki (threads)** — współdzielą pamięć, lekki narzut, wymaga synchronizacji.
- **Procesy (processes)** — odrębne przestrzenie adresowe, brak współdzielonej pamięci, komunikacja przez IPC.
- **Asynchroniczność (async)** — współbieżność kooperatywna, bez blokujących operacji, często używana w serwerach i aplikacjach I/O.

2.3 Sekcje krytyczne i wyścigi danych

Jeżeli wiele wątków jednocześnie modyfikuje współdzielone dane, mogą wystąpić **błędy niedeterministyczne**.

Przykład błędu (Python):

```
import threading

licznik = 0

def zwieksz():
    global licznik
    for _ in range(100000):
        licznik += 1

t1 = threading.Thread(target=zwieksz)
t2 = threading.Thread(target=zwieksz)
t1.start(); t2.start()
t1.join(); t2.join()

print(licznik)  # wynik < 200000!
```

👉 Występuje **race condition** — dwa wątki jednocześnie modyfikują zmienną.

2.4 Synchronizacja – Lock

Rozwiązanie problemu: użycie **muteksów (Lock)**

```
import threading

licznik = 0
lock = threading.Lock()

def zwieksz():
    global licznik
    for _ in range(100000):
        with lock:
            licznik += 1

t1 = threading.Thread(target=zwieksz)
t2 = threading.Thread(target=zwieksz)
t1.start(); t2.start()
t1.join(); t2.join()

print(licznik)  # 200000
```

---

3. Tutorial 1 – Współbieżność z użyciem wątków (Python)

3.1 Podstawowy przykład

```
import threading
import time

def zadanie(n):
    print(f"Start zadania {n}")
    time.sleep(1)
    print(f"Koniec zadania {n}")

watki = []
for i in range(5):
    t = threading.Thread(target=zadanie, args=(i,))
    watki.append(t)
    t.start()

for t in watki:
    t.join()

print("Wszystkie zadania zakończone.")
```

👉 Wątki uruchamiają się „równolegle” — kolejność rozpoczęcia i zakończenia może być różna.

3.2 Przetwarzanie danych w wielu wątkach

```
import threading

def przetwarzaj_fragment(dane, start, end):
    fragment = dane[start:end]
    wynik = sum(fragment)
    print(f"Suma fragmentu {start}-{end}: {wynik}")

dane = list(range(1_000_000))
n = len(dane)//4
watki = []

for i in range(4):
    t = threading.Thread(target=przetwarzaj_fragment, args=(dane, i*n,
(i+1)*n))
    watki.append(t)
    t.start()

for t in watki:
    t.join()
```

---

4. Tutorial 2 – Równoległość z procesami (multiprocessing)

W Pythonie ze względu na GIL (Global Interpreter Lock), wątki nie zawsze dają prawdziwą równoległość obliczeń. Do tego celu używa się **modułu multiprocessing**.

```
from multiprocessing import Process, cpu_count
```

```
def policz(n):
    print(f"Proces {n}")
    s = sum(range(10_000_000))
    print(f"Proces {n} skończył")

procesy = []
for i in range(cpu_count()):
    p = Process(target=policz, args=(i,))
    procesy.append(p)
    p.start()

for p in procesy:
    p.join()

print("Wszystkie procesy zakończone.")
```

👉 Procesy działają naprawdę równolegle — wykorzystują wiele rdzeni CPU.

---

5. Zadania do samodzielnego wykonania

Zadanie 1 (obowiązkowe)

Napisz program, który:

- dzieli dużą listę liczb na fragmenty,
- uruchamia 4 wątki do równoległego liczenia sumy fragmentów,
- sumuje wyniki końcowe i wypisuje wynik całkowity.
  Zadbaj o synchronizację (Lock).

Zadanie 2 (obowiązkowe)

Napisz wersję zadania 1 z użyciem **procesów** zamiast wątków. Porównaj czasy działania.

Zadanie 3 (dodatkowe)

Zaimplementuj prostą „symulację serwera”:

- każdy wątek reprezentuje klienta,
- klienci wysyłają żądania (np. obliczenia),
- serwer odpowiada — użyj kolejki do komunikacji
  (np. `queue.Queue` lub `multiprocessing.Queue`).

---

6. Pytania kontrolne

1. Na czym polega różnica między współbieżnością a równoległością?
2. Jakie są różnice między wątkami a procesami?
3. Co to jest sekcja krytyczna?
4. Jak działa mechanizm Lock i dlaczego jest potrzebny?
5. Dlaczego w Pythonie multiprocessing daje prawdziwą równoległość, a threading nie zawsze?

---

7. Checklisty

Współbieżność / równoległość

- Użyto wątków lub procesów poprawnie.
- Zastosowano synchronizację w razie dostępu do danych współdzielonych.
- Program działa poprawnie dla wielu równoległych jednostek.
- Porównano czasy wykonania wątków i procesów.

---

8. Wzór sprawozdania

Dodatkowo:

- Fragmenty kodu wątków i procesów.
- Zrzuty wyników z różnych uruchomień.
- Porównanie wydajności i omówienie różnic.
- Opis potencjalnych problemów (np. race condition) i ich rozwiązania.

Zakończenie

Po wykonaniu ćwiczenia student powinien:

- rozumieć paradygmat współbieżny i równoległy,
- znać różnicę między wątkami a procesami,
- umieć uruchamiać wiele zadań jednocześnie,
- znać podstawy synchronizacji,
- rozumieć potencjalne problemy i ich rozwiązania.
