# Programowanie współbieżne

Lista zadań nr 1

Na ćwiczenia 19. października 2022

**Zadanie 1.** Moc zużywana przez pracujący procesor dana jest wzorem

$$P = CV^2 f,$$

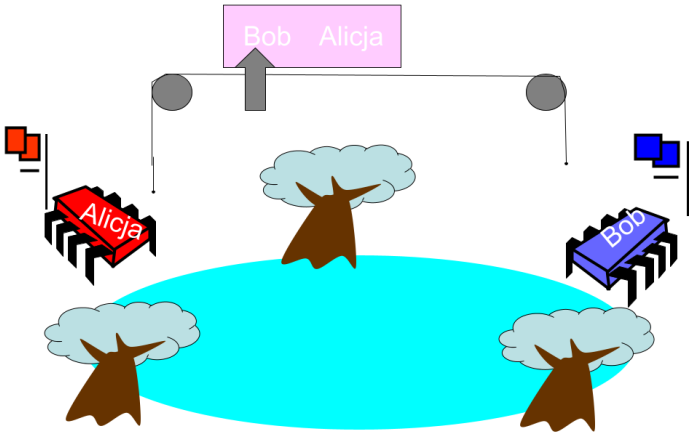
gdzie

- $C$ - pojemność elektryczna układu
- $V$ - napięcie pracy
- $f$ - częstotliwość pracy

Wiadomo, że wartość $C$ liniowo zależy od liczby tranzystorów w procesorze (im więcej tranzystorów tym większa pojemność układu), a napięcie $V$ liniowo zależy od częstotliwości $f$[1]. Na wzorcowym jednordzeniowym procesorze uruchamiasz sekwencyjnie (w jednym wątku) pewien algorytm. Załóżmy, że można ten algorytm zrównoleglić i uruchomić na dwóch rdzeniach uzyskując dwukrotne przyspieszenie. Ty jednak wolałbyś procesor zużywający znacznie mniej mocy, ale wykonujący algorytm tak samo szybko jak wzorcowy. Czy można taki zaprojektować?

**Zadanie 2.** Zdefiniuj problem **sekcji krytycznej**. Jako przykładem posłuż się historyjką o smokach w jeziorze opowiedzianą na wykładzie. Rozwiązanie tego problemu z użyciem flag spełnia warunki: **wzajemnego wykluczania** (w jeziorze znajdzie się co najwyżej jeden smok) i **niezakleszczenia**. Ten drugi warunek oznacza, że jeśli jeden smok chce wejść do pustego jeziora, to wchodzi. A jeśli chcą wejść dwa naraz to wejdzie smok Alicji. To niestety oznacza, że smok Boba może zostać **zagłodzony** (nigdy nie wejdzie do jeziora mimo tego, że chce). By rozwiązać ten problem zainstalowano wskaźnik połączony ze sznurkiem jak na rysunku poniżej.

[1] To założenie jest uproszczeniem.




Teraz, oprócz posługiwania się flagami, Alicja i Bob mogą ciągnąć za końce sznurka. Jeśli Alicja pociągnie swój koniec, wskaźnik zacznie wskazywać na słowo Bob, i na odwrót. Stan wskaźnika jest widoczny dla obydwojga. Popraw protokół tak, by żaden ze smoków nigdy nie był zagłodzony.

**Zadanie 3.** Zdefiniuj problem **producenta-konsumenta.** Jako przykładem posłuż się historyjką o karmieniu smoków opowiedzianą na wykładzie. Rozwiązaniem tego problemu był następujący protokół wykorzystujący pojedynczą puszkę znajdującą się na parapecie u Alicji.

Alicja:
1. Czeka aż puszka zniknie z parapetu.
2. Wypuszcza smoki.
3. Gdy zwierzęta wrócą, Alicja sprawdza, czy zjadły wszystko. Jeśli tak, to ustawia puszkę z powrotem na miejscu.

Bob:
1. Czeka aż puszka stanie na parapecie.
2. Zostawia jedzenie w jeziorze.
3. Pociąga za sznurek i strąca puszkę.

Protokół ten spełnia warunki: **wzajemne wykluczanie** (Bob i smoki nie przebywają jednocześnie w jeziorze), **niezagłodzenie** (smoki będą jadły nieskończenie często, pod warunkiem że Alicja i Bob są zawsze gotowi je obsłużyć) oraz **producenta-konsumenta** (zwierzęta nie wchodzą do jeziora w którym nie ma jedzenia, a Bob nie dokłada jedzenia, jeśli poprzednie nie zostało jeszcze zjedzone). Niestety ma też wadę: Bob musi mieć balkon Alicji w zasięgu wzroku, by widzieć w jakim stanie jest puszka. Zaproponuj protokół rozwiązujący ten problem przy użyciu puszek, jeśli Alicja i Bob nie widzą nawzajem swoich balkonów (całe jezioro zasnuwa mgła).

**Zadanie 4**. Które z poniższych zdań definiują własności **bezpieczeństwa** ("nigdy nic złego się nie stanie") a które **żywotności** ("kiedyś stanie się coś pożądanego").

1. Klienci są obsługiwani zgodnie z kolejnością przybycia.
2. Co idzie do góry, musi zejść na dół.
3. Jeśli co najmniej dwa wątki oczekują na wejście do sekcji krytycznej, to przynajmniej jednemu to się udaje.
4. Jeśli nastąpi przerwanie, to w ciągu sekundy drukowany jest komunikat.
5. Jeśli nastąpi przerwanie, to drukowany jest komunikat.
6. Koszt życia nigdy nie spada.
7. Dwie rzeczy są pewne: śmierć i podatki.

**Zadanie 5**. W historyjkach o smokach każda puszka lub flaga reprezentowała pojedynczy bit informacji współdzielonej przez Alicję i Boba. Potrzebowaliśmy jednego lub dwóch takich bitów by rozwiązać proste problemy z historyjek zadowalająco efektywnie. W tym zadaniu pokażesz, że nawet jeden współdzielony bit wystarczy do rozwiązania bardziej złożonego problemu. Oto ono.

Jesteś jednym z grupy niedawno aresztowanych. Strażnik, obłąkany informatyk, wygłasza następujące oświadczenie:

*Dzisiaj możecie się spotkać i zaplanować strategię działania, ale jutro znajdziecie się w odosobnionych celach i nie będziecie mogli się ze sobą komunikować.*

*Zbudowałem "przełączalnię", czyli pokój, w którym znajduje się przełącznik mogący być w jednym z dwóch stanów: włączony albo wyłączony. Przełącznik nie jest z niczym połączony.*

*Od czasu do czasu będę wybierał więźnia, aby wchodził do tej "przełączalni". Więzień będzie mógł zmienić stan przełącznika (włączyć lub wyłączyć), będzie mógł też niczego nie zmienić. Nikt oprócz więźniów nie będzie tam wchodził.*

*Każdy więzień odwiedzi przełączalnię dowolną liczbę razy. Dokładniej, dla dowolnego $N$, każdy z was odwiedzi przełączalnię przynajmniej $N$ razy. No chyba że któryś z was kiedyś oświadczy: "każdy z nas przynajmniej raz był w przełączalni". Jeśli stwierdzenie to będzie prawdziwe, jesteście wolni. W przeciwnym razie rzucę was na pożarcie smokom.*

Opracuj skuteczną strategię uwolnienia się, wiedząc że początkowo przełącznik jest wyłączony. Wskazówka: nie wszyscy więźniowie muszą robić to samo.

**Zadanie 6.** Opracuj skuteczną strategię dla problemu z zadania 5. nie wiedząc w jakim stanie jest początkowo przełącznik.

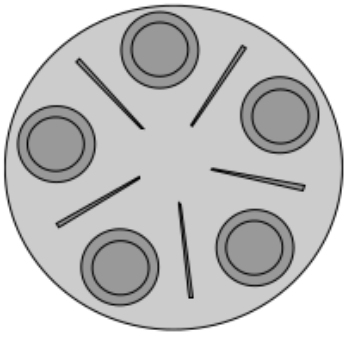
**Zadanie 7.** Dany jest program P w którym M jest jedyną metodą której nie da się zrównoleglić. Program początkowo uruchamiamy na maszynie z jednym procesorem, a następnie jego wersję zrównoleglonąP' uruchamiamy na maszynie z n procesorami. Na podstawie prawa Amdahla odpowiedz na poniższe pytania.

1. Załóżmy, że wykonanie metody M zajmuje 40% czasu wykonania programu P. Jaki jest limit całkowitego przyspieszenia, które można osiągnąć gdy n zmierza do nieskończoności?

Wydajność programu P' okazała się niewystarczająca, dlatego decydujemy się na zaprogramowanie szybszego (ale wciąż szeregowego) algorytmu dla M, w wyniku czego otrzymujemy program P''.

2. Załóżmy, że metoda M zajmuje 30% czasu programu P. Jakie przyspieszenie w działaniu M musimy uzyskać, by cały program P'' wykonywał się dwukrotnie szybciej niż P'?
3. Załóżmy, że umiemy zaprogramować trzykrotnie szybszy algorytm dla M. Jaki ułamek całościowego czasu wykonania programu P musi zajmować M, aby P'' wykonywał się dwa razy szybciej od P'?

**Zadanie 8.** Problem ucztujących filozofów został sformułowany przez E.W.Dijkstrę, pioniera w dziedzinie współbieżności. Wyobraźmy sobie N filozofów, którzy przez całe życie tylko myślą i ucztują. Siedzą na N krzesłach przy okrągłym stole, na którym stoją ich talerze. Jednak, jak widzimy na rysunku poniżej (dla N = 5), jest tylko N pałeczek (widelców). Każdy filozof myśli, a kiedy poczuje głód chwyta najpierw jedną a później drugą pałeczkę, które znajdują się po obu stronach jego talerza. Jeśli uda mu się je obydwie pochwycić, to będzie przez chwilę jadł. Kiedy skończy jeść, odkłada pałeczki i znów zaczyna myśleć.



Chcemy napisać program symulujący zachowanie filozofów. W programie tym każdy filozof to wątek, a pałeczki to współdzielone obiekty. Należy zapobiec sytuacji, w której dwóch filozofów będzie trzymało na raz tę samą pałeczkę.

Oto implementacja klasy Fork. Potraktuj ją jako czarną skrzynkę (nie musisz rozumieć, jak działa). Oprócz konstruktora ma ona dwie metody:

**public void get()**

poczekaj aż pałeczka zostanie odłożona, a wtedy ją weź. Ta funkcja wykonuje się w sposób **atomowy**, tzn. jeśli filozof zobaczył odłożoną pałeczkę to bierze ją w tej samej chwili (więc nie może tego zrobić sąsiad).

**public void put()**

odkłada wziętą uprzednio pałeczkę

```
class Fork {
	boolean taken;
	int id ;
	public Fork(int myID) {
		id = myID;
	}
	public synchronized void get() throws InterruptedException {
		while (taken) {
			wait ();
		}
		taken = true;
	}
	public synchronized void put() {
		taken = false;
		notify ();
	}
}
```

Oto implementacja klasy **Philosopher**, która dziedziczy z klasy **Thread**. Klasa reimplementuje odziedziczoną metodę **run()**, w której znajduje się kod wątku, w tym przypadku jest to symulacja działania pojedynczego filozofa.

```
class Philosopher extends Thread {
	int id;
	Fork left;
	Fork right;

	public Philosopher(int myID, Fork myLeft, Fork myRight) {
		id = myID;
		left = myLeft;
		right = myRight;
	}

	public void run() {
		Random random = new Random();
		while (true) {
			try {
				sleep(random.nextInt(1000));
				sleep(100);
				System.out.println("Philosopher " + id + " is hungry");
				left.get();
```

```
                        right.get();
                        left.put();
                        right.put();
                    } catch (InterruptedException ex) {
                        return;
                    }
                }
            }
}
```

Poniżej znajduje się główna klasa symulatora. Widelce i filozofowie zapamiętani są w tablicach. Każdy nowo utworzony filozof staje się wątkiem, który należy uruchomić metodą **start()**. W efekcie wykona się metoda **run()** wątku. Program ma N+1 wątków: wątek główny (ten, w którym wykonuje się funkcja **main()**) oraz N wątków filozofów. W typowym programie wielowątkowym wątek główny powinien poczekać na zakończenie pozostałych wątków. Służy do tego metoda **join()**, wywoływana przez wątek główny na wątku, na który czekamy. W tym programie wątki filozofów nigdy się nie zakończą, ale użycie metody **join()** sprawi, że nie zakończy się też wątek główny, co mogłoby pociągnąć za sobą zamknięcie całego programu przez system operacyjny.

```
public class Main {
	static final int N = 10;

	public static void main(String[] args) throws InterruptedException {
		System.out.println("Starting...");

		Fork forks[] = new Fork[N];
		Philosopher philos[] = new Philosopher[N];

		for (int i = 0; i < N; i++)
			forks[i] = new Fork(i);

		Philosopher p;

		for (int i = 0; i < N; i++) {
			p = new Philosopher(i,
					forks[i],
					forks[(i + 1) % N]);
			philos[i] = p;
			p.start();
		}

		for (int i = 0; i < N; i++) {
			System.out.printf("Waiting for philosopher %d to finish\n", i);
			philos[i].join();
		}
	}
}
```

Uruchom program złożony z klas Fork, Philosopher i Main (plik z kodem możesz ściągnąć ze SKOSa). Upewnij się, że rozumiesz

jak działają dwie ostatnie klasy. Następnie pokaż, że istnieje przebieg programu prowadzący do **zakleszczenia** (żaden filozof nigdy nie będzie w stanie zjeść). Następnie tak zmodyfikuj pętlę **while** w metodzie **run()** filozofa, żeby rozwiązać ten problem (wskazówka: nie wszyscy filozofowie muszą podnosić pałeczki w tej samej kolejności). Możesz założyć[2], że wątki czekające na zakończenie metody **get()** wywołanej na dowolnym widelcu nie zostaną zagłodzone.

[2] To założenie jest uproszczeniem. Wytłumaczę się z niego podczas omawiania monitorów w Javie.