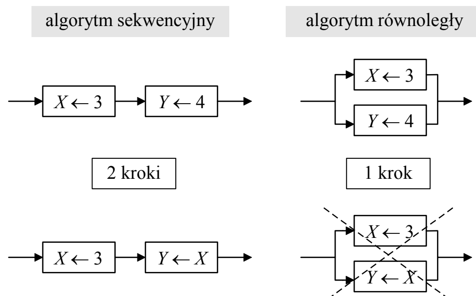
Obliczenia współbieżne

czyli zmiana założenia o sekwencyjnym działaniu procesora

- rozwiązywanie problemu algorytmicznego za pomocą współpracujących ze sobą wielu procesorów
- wykorzystanie komputerów równoległych, składających się z wielu rozłącznych elementów przetwarzających
- modele obliczeń i przetwarzania informacji w środowiskach rozproszonych (sieci telekomunikacyjne, systemy rezerwacji biletów lotniczych, długoterminowe prognozy pogody wyznaczane równolegle w wielu centrach obliczeniowych)

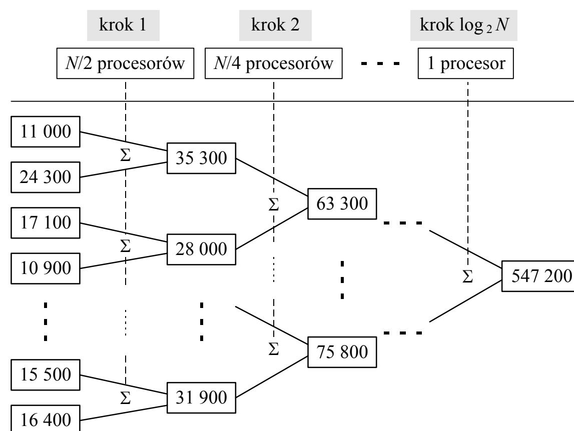


Sekwencyjność właściwa problemu algorytmicznego:

- stosując algorytm równoległy możemy wiele zyskać jeśli rozwiązywany problem ma bardzo małą sekwencyjność właściwą
- przy obliczaniu równoległym duża sekwencyjność właściwa problemu nie pozwala na wyraźne skrócenie czasu jego rozwiązywania

Przykład sumowania zarobków w czasie logarytmicznym

Naturalny algorytm sekwencyjny o koszcie $O(N)$: dodawanie N razy do sumy bieżącej
Algorytm równoległy o koszcie $O(\log N)$:



– 1000 pensji w 10 krokach
– 1 000 000 pensji w 20 krokach

O szybkości algorytmów równoległych, oczywiście poza liczbą dostępnych procesorów, decydują także struktury danych i metody komunikacji!

W algorytmie sumowania N liczb:

- dla osiągnięcia redukcji z $O(N)$ do $O(\log N)$ potrzebujemy $N/2$ procesorów
- mając do dyspozycji ustaloną liczbę procesorów poprawimy przetwarzanie tylko o stałą (np. 100 razy szybciej), ale nie o rząd wielkości
- uzyskanie poprawy rzędu wielkości wymaga rozszerzającej się równoległości tzn. liczba procesorów rośnie proporcjonalnie do N
- mając do dyspozycji $\sqrt{N}$ procesorów można go wykonać w czasie $O(\sqrt{N})$:
  - 1 000 000 pensji w 1000 krokach na 1000 procesorach

Sortowanie równoległe

Rozważmy sekwencyjny algorytm sortowania przez scalanie:

*procedura **sortuj-listę** L;*

- jeśli L zawiera tylko jeden element, to jest posortowana;
- w przeciwnym razie wykonaj co następuje:
  - ∗ podziel listę L na dwie połowy L_1 i L_2;
  - ∗ wywołaj ***sortuj-listę*** L_1;
  - ∗ wywołaj ***sortuj-listę*** L_2;
  - ∗ scal listy L_1 i L_2 w jedną posortowaną listę;
- wróć do poziomu wywołania.

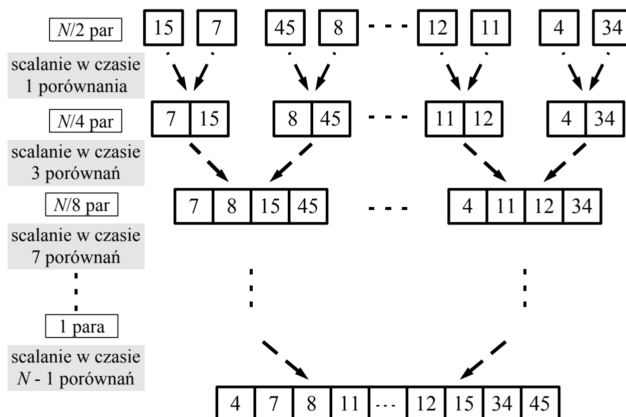
- złożoność czasowa $O(N \times \log N)$

Wersja równoległa:

*procedura **równolegle-sortuj-listę** L;*

- jeśli L zawiera tylko jeden element, to jest posortowana;
- w przeciwnym razie wykonaj co następuje:
  - ∗ podziel listę L na dwie połowy L_1 i L_2;
  - ∗ wywołaj równocześnie ***równolegle-sortuj-listę*** L_1 i ***równolegle-sortuj-listę*** L_2;
  - ∗ scal listy L_1 i L_2 w jedną posortowaną listę;
- wróć do poziomu wywołania.

Analiza złożoności:



zatem całkowita liczba porównań wyniesie:

$$1 + 3 + 7 + 15 + ... + (N - 1) \leq 2 \times N$$ - liczba rzędu $O(N)$

Złożoność iloczynowa: ***liczba procesorów* × *czas***

- złożoność „rozmiaru" algorytmu
- najlepsza złożoność iloczynowa nie będzie lepsza niż dolne ograniczenie sekwencyjnej złożoności problemu

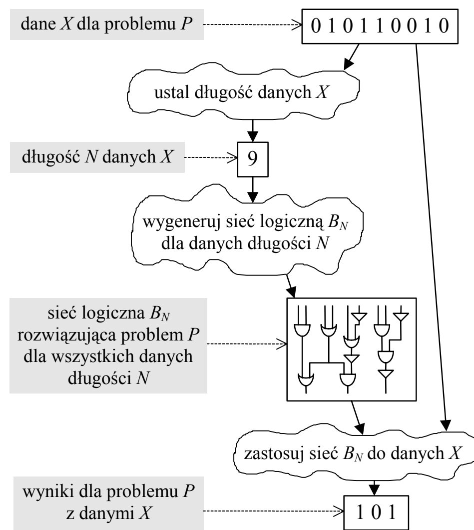
Rodzaj algorytmu	Nazwa algorytmu	Liczba procesorów (rozmiar)	Czas (najgorszy przypadek)	Iloczyn (rozmiar × czas)
sekwencyjny	sortowanie bąbelkowe	**1**	$O(N^2)$	$O(N^2)$
	sortowanie przez scalanie	**1**	$O(N \times \log N)$	$O(N \times \log N)$
równoległy	równoległe sortowanie przez scalanie	$O(N)$	$O(N)$	$O(N^2)$
	sieć sortująca parzysto-nieparzyście	$O(N \times (\log N)^2)$	$O((\log N)^2)$	$O(N \times (\log N)^4)$
	„optymalna" sieć sortująca	$O(N)$	$O(\log N)$	$O(N \times \log N)$

Czy istnieje „optymalna" sieć sortująca?

Modele współpracy procesorów pracujących równolegle

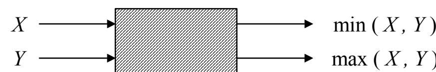
- pamięć dzielona
  - ∗ dzielenie dostępu tylko podczas odczytywania pamięci
  - ∗ dzielenie dostępu podczas zapisywania ⇒ rozwiązywanie konfliktów dostępu
  - ∗ nieograniczona pamięć dzielona jest praktycznie nie do zrealizowania ze względu na skomplikowane schematy wzajemnych połączeń elementów pamięci i procesorów
- sieci o ustalonej konfiguracji połączeń
  - ∗ każdy z procesorów może być połączony z co najwyżej pewną **stałą** (dla sieci) liczbą procesorów sąsiadujących
  - ∗ sieci są konstruowane często jako równoległe maszyny rozwiązujące szczególne problemy algorytmiczne
  - ∗ sieci logiczne (boolowskie) są dobrze znanym przykładem:
    procesory = bramki realizujące proste funkcje logiczne na bitach (AND, OR, NOT itp.)

Rozwiązywanie problemu algorytmicznego za pomocą sieci logicznej

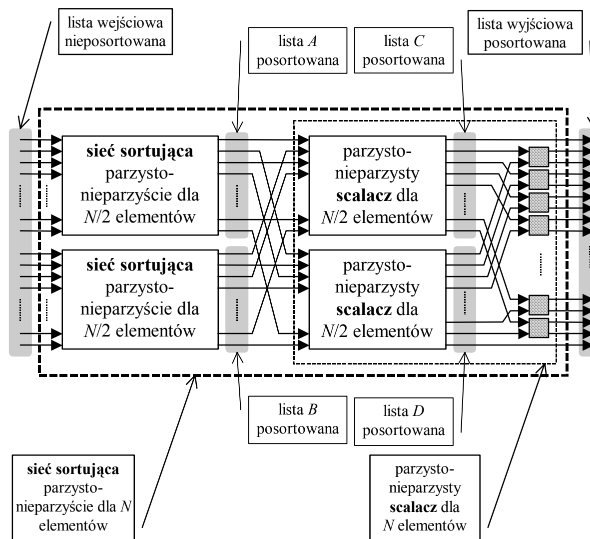


Sieć sortująca parzysto-nieparzyście

Użyty jest w niej jeden prosty typ procesora - **komparator**:



Reguła (rekurencyjna) konstruowania sieci:

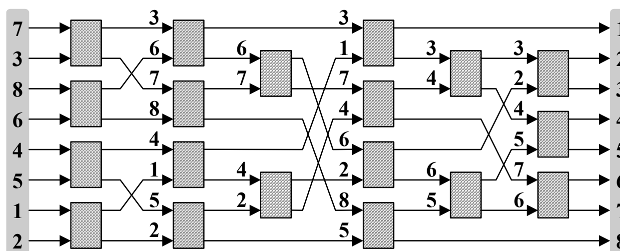


Czas sortowania: $\boldsymbol{O}((\log N)^2)$
Rozmiar sieci (liczba procesorów): $\boldsymbol{O}(N \times (\log N)^2)$
Złożoność iloczynowa: $\boldsymbol{O}(N \times (\log N)^4)$

Dla sieci sortującej parzysto-nieparzyście:

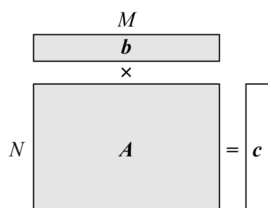
Długość sortowanej listy	czas sortowania	rozmiar sieci
100	ok. **25** porównań	ok. 1000 komparatorów
1000	ok. **55** porównań	ok. 23 000 komparator

Przykład: sieć sortująca parzysto-nieparzyście 8 elementów



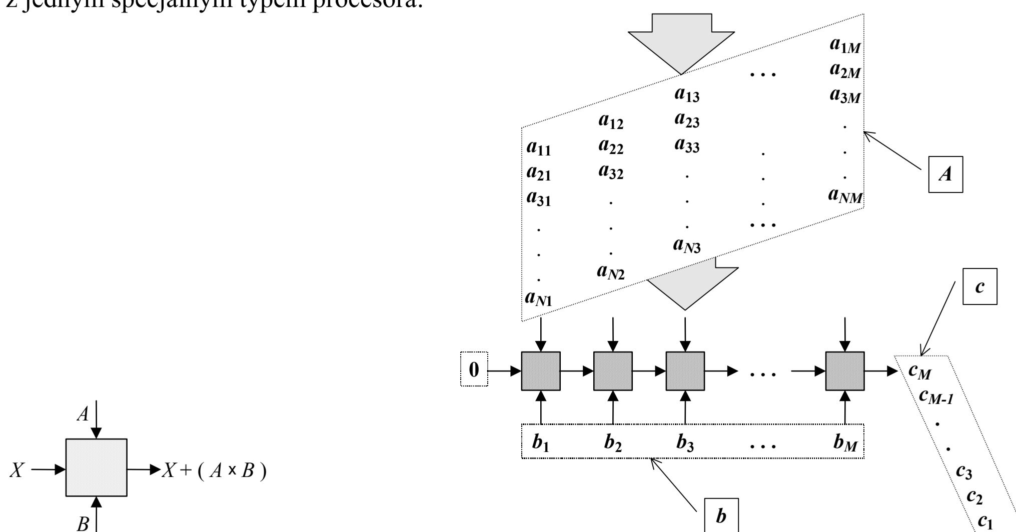
Sieci systoliczne (ang. systolic – skurczowy)

Przykład - problem mnożenia macierzy $N \times M$ przez wektor M elementowy

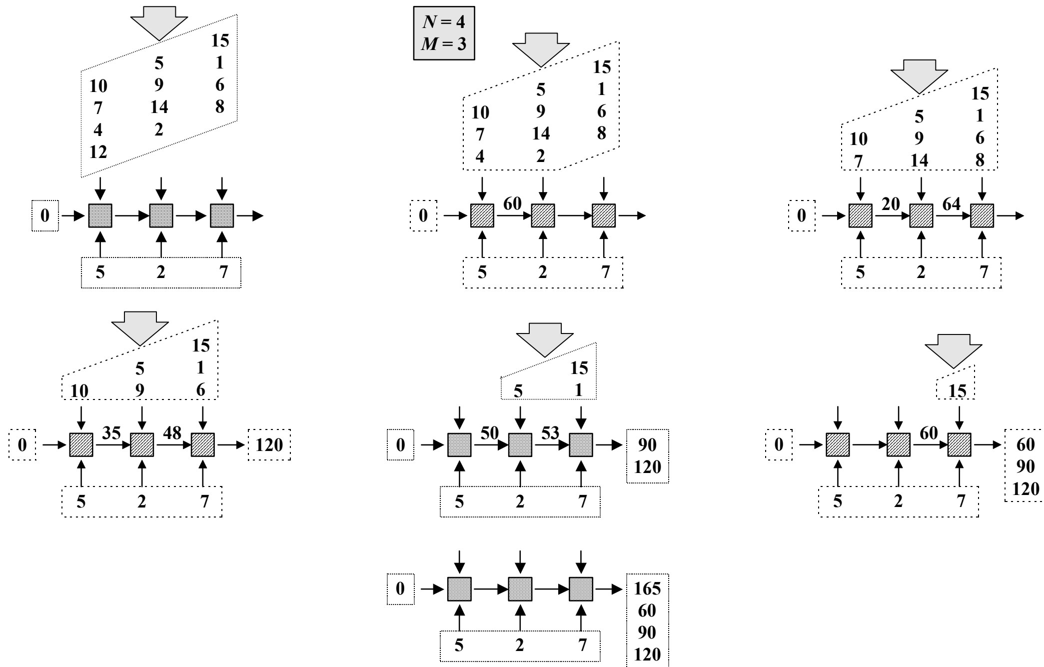


Złożoność sekwencyjnego algorytmu wyznaczania $c = A \times b$: $\boldsymbol{O}(N \times M)$

Sieć systoliczna do mnożenia macierzy przez wektor
z jednym specjalnym typem procesora:

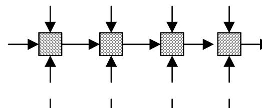


Algorytm realizowany przez tą sieć systoliczną ma złożoność $\boldsymbol{O}(N + M)$

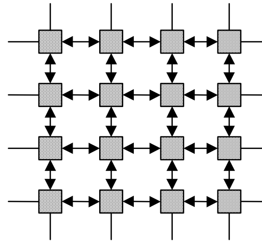


Typowe rozmieszczenia procesorów w sieciach systolicznych:

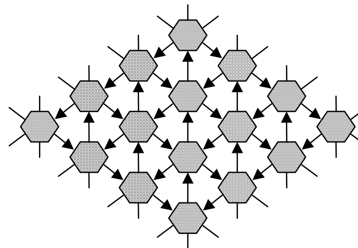
potok:



krata:



plaster miodu:



Co można, a czego nie można osiągnąć równoległością:

- wiele problemów można rozwiązać <u>szybciej</u> niż sekwencyjnie
- można niektóre problemy rozwiązywać <u>szybciej</u> nawet <u>o rząd</u> <u>wielkości</u>, jeśli da się zastosować rozszerzającą się równoległość
- dla problemów nierozstrzygalnych <u>nie da się</u> skonstruować algorytmu równoległego - klasa problemów rozwiązywalnych jest niewrażliwa na dodanie równoległości
- wszystkie problemy klasy *NP* <u>mają</u> rozwiązania równoległe znajdowane w czasie wielomianowym, ale
  - <u>liczba procesorów</u> potrzebnych do rozwiązania problemu *NP*-zupełnego w rozsądnym czasie rośnie <u>wykładniczo</u>
  - do końca nie wiadomo, czy problemy klasy *NP* są rzeczywiście trudno rozwiązywalne i trzeba szukać ratunku w równoległości
  - rzeczywiste komputery równoległe mają silne ograniczenia związane z przepustowością połączeń pomiędzy procesorami
- <u>nie wiadomo</u>, czy można zastosować równoległość, nawet z niewielomianową liczbą procesorów, do rozwiązania w czasie wielomianowym problemu o udowodnionej sekwencyjnej złożoności wykładniczej

Teza o obliczeniach współbieżnych

Wszystkie uniwersalne modele rozszerzającej się równoległości są równoważne w czasie wielomianowym. Każdy model można symulować innym tracąc co najwyżej czas wielomianowy.

+

Z dokładnością do różnic wielomianowych zapotrzebowanie na czas w modelu równoległym odpowiada zapotrzebowaniu na pamięć w modelu sekwencyjnym sekwencyjnym.

tzn.

1. Jeśli problem można rozwiązać algorytmem sekwencyjnym korzystając z $\boldsymbol{f(N)}$ pamięci, to można go rozwiązać algorytmem współbieżnym w czasie nie gorszym niż $\boldsymbol{W(f(N))}$.
2. Jeśli można rozwiązać problem algorytmem współbieżnym w czasie $\boldsymbol{f(N)}$, to można go rozwiązać algorytmem sekwencyjnym korzystając z ilości pamięci ograniczonej przez $\boldsymbol{W(f(N))}$.

⇓

sekwencyjna pamięć wielomianowa = równoległy czas wielomianowy

Klasa Nicka (*NC*)

Def.
Problem należy do klasy ***NC***, jeśli istnieje dla niego algorytm o czasie $\boldsymbol{O(W(\log N))}$, który wymaga tylko wielomianowej liczby procesorów.

Sumowanie zarobków, sortowanie, mnożenie macierzy przez wektor i wiele innych problemów należy do klasy *NC*.

- klasa *NC* nie jest wrażliwa na zmianę modelu obliczeń współbieżnych
- można pokazać, że $NC \subseteq P$, ale nie wiadomo, czy $P \subseteq NC$
  (np. podejrzewa się, że problem znajdowania NWD dla pary liczb całkowitych nie należy do klasy *NC*)
- także w klasie *NC* funkcjonuje pojęcie zupełności i warto udowadniać, że jakiś problem jest *NC*-zupełny

Jeśli jest tak, że

$$NC \subset P \subset NP \subset PSPACE \ (= \text{równolegle-}PTIME),$$

to:

- istnieją problemy rozwiązywalne sekwencyjnie z rozsądnym zapotrzebowaniem na pamięć (równolegle w rozsądnym czasie), których nie można rozwiązać sekwencyjnie w rozsądnym czasie, nawet korzystając z „magicznego" niedeterminizmu;
- istnieją problemy rozwiązywalne w rozsądnym czasie za pomocą „magicznego" niedeterminizmu, których nie można rozwiązać w takim czasie nie korzystając z niego
- istnieją problemy rozwiązywalne sekwencyjnie w rozsądnym czasie, których nie można rozwiązać równolegle w bardzo krótkim czasie na sprzęcie o rozsądnym rozmiarze pamięci

---

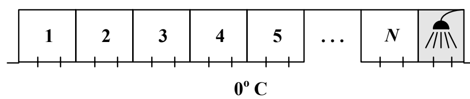
Systemy trwającej współbieżności - elementy składowe pracują współbieżnie i zachowują się w czasie w określony sposób, nie naruszając narzuconych wymagań

Systemy rozproszone - składniki współbieżne są fizycznie oddalone i komunikacja pomiędzy nimi musi być minimalizowana

Dwa przykładowe (trochę śmieszne) **problemy**, które wiele mówią o rzeczywistych problemach w systemach trwale współbieżnych i rozproszonych:

- dostęp do łazienki w tanim hotelu - problem „prysznica"
- wejście w posiadanie dwóch pałeczek przy stole - problem „chińskich filozofów"

Problem „prysznica"



Założenia:

- każdy chce w końcu wziąć prysznic (zasób krytyczny)

- nie można stać w kolejce na korytarzu
- goście nie potrafią się porozumiewać (ograniczenia komunikacyjne)

Rozwiązanie:
- tablica na zewnątrz łazienki (pamięć dzielona)
- po wyjściu każdy wymazuje numer wpisany na tablicy i wpisuje numer następny
- każdy podchodzi co jakiś czas do tablicy i sprawdza czy nie pojawił się na niej numer jego pokoju, jeśli tak, to bierze prysznic

Zagrożenia:
- jeden z pokoi jest pusty i już nikt więcej się nie umyje (zastój systemu)
- każdy jest zmuszony do brania prysznica w tym samym cyklu sprawdzeń
- jeden z gości po wyjściu wpisuje dowolne numery (lub w ogóle nie ma tablicy) ⇒ jeden lub więcej gości nie będzie mogło w ogóle się umyć („zagłodzenie” procesora)

Modyfikacja problemu:
- prysznic jest w każdym pokoju, ale na raz działa tylko jeden (słabe ciśnienie wody?)

Rozwiązanie (dla 2 pokoi):
- tablica na korytarzu z trzema polami:

Z	$\in \{1, 2\}$	, obaj czytają i zapisują	(zmienna dzielona)
X_1	$\in \{tak, nie\}$	, obaj czytają, 1 zapisuje	(zmienna rozproszona)
X_2	$\in \{tak, nie\}$	, obaj czytają, 2 zapisuje	(zmienna rozproszona)

wartości początkowe: $X_1, X_2 \leftarrow nie$; $Z \leftarrow$ cokolwiek

Postępowanie 1 gościa:

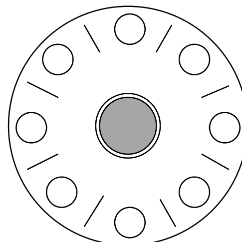
powtarza bez końca:
1. robi co chce, dopóki się nie ubrudzi
2. $X_1 \leftarrow tak$ (chce wziąć prysznic)
3. $Z \leftarrow 2$ (zaprasza drugiego gościa do wzięcia prysznicu)
4. czeka, aż $X_2 = nie$ lub $Z = 1$
5. bierze prysznic
6. $X_1 \leftarrow nie$ (jest wymyty)

Postępowanie 2 gościa:

powtarza bez końca:
1. robi co chce, dopóki się nie ubrudzi
2. $X_2 \leftarrow tak$ (chce wziąć prysznic)
3. $Z \leftarrow 1$ (zaprasza drugiego gościa do wzięcia prysznicu)
4. czeka, aż $X_1 = nie$ lub $Z = 2$
5. bierze prysznic
6. $X_2 \leftarrow nie$ (jest wymyty)

Z równoległością trzeba uważać: zamiana kolejności kroku 2. i 3. dla obu gości prowadzi do próby jednoczesnego użycia prysznica!

Problem „chińskich filozofów”



Założenia:
- do jedzenia są potrzebne dwie pałeczki (zasób krytyczny)
- filozof może do jedzenia użyć tylko najbliższych dwóch pałeczek
- filozofowie są zamyśleni i nie porozumiewają się, a czasami chcą zjeść trochę ryżu

Rozwiązanie:
- wprowadzamy szefa sali (planista)

- szef wyprasza najedzonych filozofów z sali
- wpuszcza ich na salę kiedy chcą jeść
- szef wie ilu jest filozofów (*N*) i pozwala na przebywanie w sali nie więcej niż *N* - 1 filozofom

⇓

co najmniej dwom filozofom przy stole brakuje sąsiada

⇓

co najmniej jeden filozof może jeść

Pytanie:
Czy problem „chińskich filozofów" można rozwiązać nie wprowadzając szefa sali lub jakiejś formy pamięci dzielonej?

NIE!

Nie istnieje rozwiązanie w pełni *rozproszone* (bez pamięci dzielonej ze wszystkimi zmiennymi rozproszonymi) i *symetryczne* (wszyscy filozofowie zachowują się identycznie)

Poważne zagadnienia:
- specyfikacja ograniczeń globalnych dla całego systemu (np. tylko jeden gość pod prysznicem)
- zapewnianie dostępu do zasobu krytycznego (np. prysznic lub pałeczki)
- zapobieganie zastojowi w systemie (np. wszyscy czekają na gościa z pustego pokoju)
- unikanie sytuacji „zagłodzenia" jednego lub kilku procesorów (np. filozof zmarł z głodu)
- narzucanie protokołów algorytmicznych sterujących zachowaniem się każdego z procesorów
- rozwiązywanie problemu wzajemnego wykluczania (np. jeśli filozof je, to jego sąsiedzi nie mogą)
- czekanie aktywne (np. gość ma ciągle sprawdzać, czy prysznic jest wolny)
- zmienne dzielone (każdy procesor może ją zmienić) i rozproszone (czytać je mogą wszystkie procesory, ale zmienić tylko upoważniony)
- ogólne kategorie wymagań:
  bezpieczeństwo (niepożądane stany nie pojawią się)
  i żywotność (pożądane stany w końcu się pojawią)
- rozważenie możliwości wprowadzenia planisty - procesora nadrzędnego (np. szef sali jadalnej filozofów)
- uczciwość planisty (protokół algorytmiczny jego pracy, który zapewnia jednakowe traktowanie wszystkich procesorów w systemie)
- ilościowe oceny dotyczące uczciwości planisty - uzgadnianie w czasie dostępu wolniejszych i szybszych procesorów
- organizowanie współbieżnych systemów czasu rzeczywistego

Języki programowania współbieżnego:
Simula 67, Concurrent Pascal, Ada, Occam, Modula 2, ...