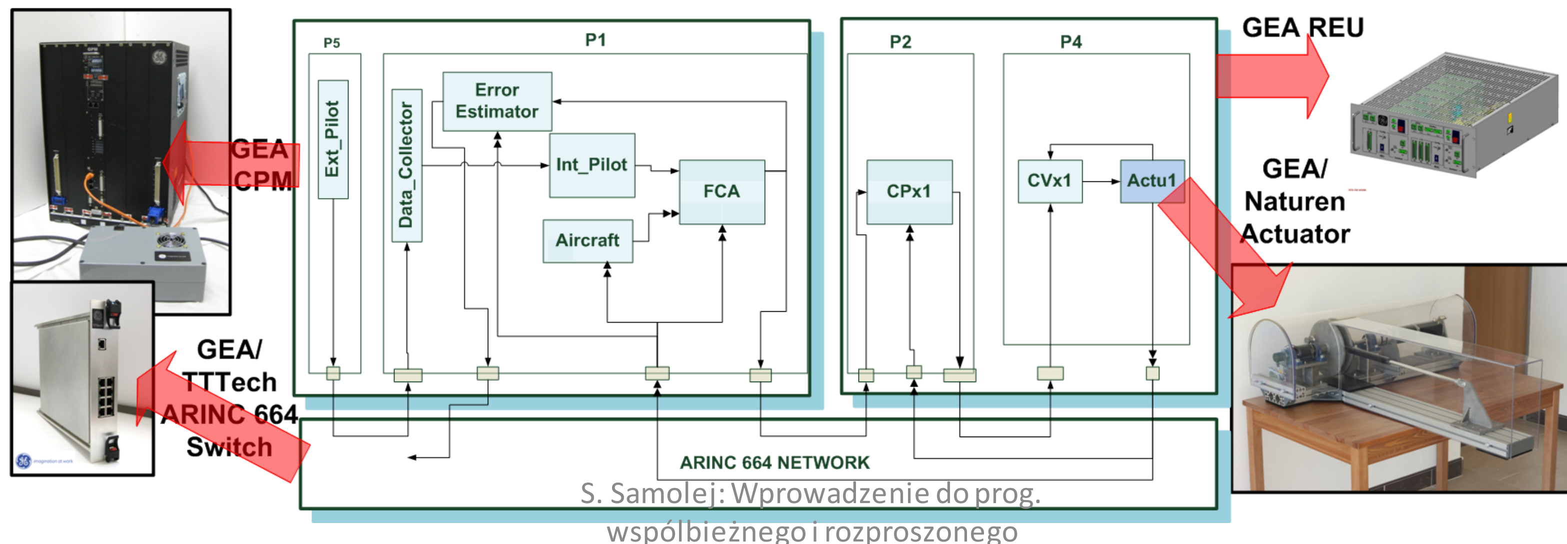
Przegląd zagadnień programowania współbieżnego

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na: http://wazniak.mimuw.edu.pl/

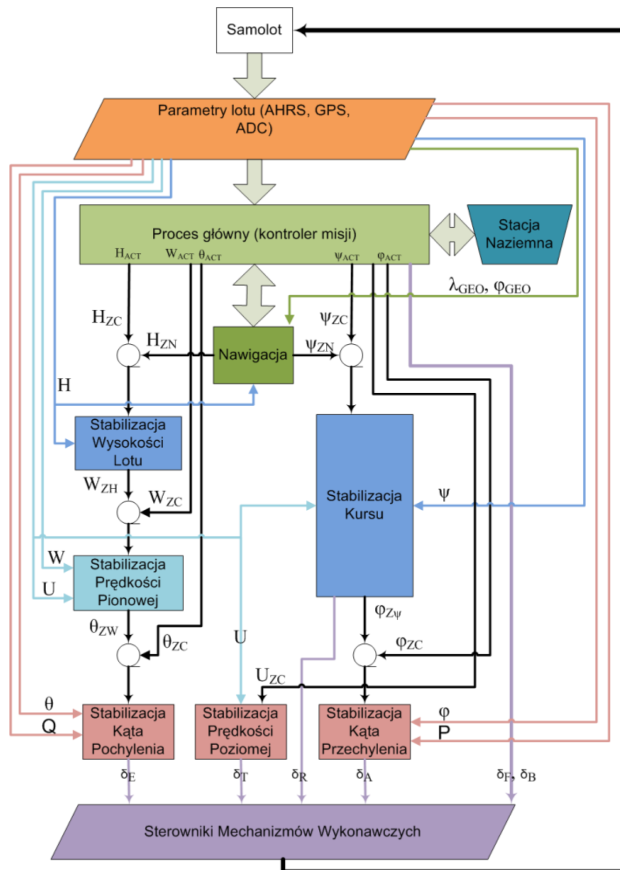
Kompetencje wykładowcy

- Doktorat z inżynierii oprogramowania systemów czasu rzeczywistego (informatyka)
- Współpraca z Airbus i General Electric Aviation w dziedzinie wytwarzania rozproszonych systemów sterowania lotem
- Współpraca z Katedrą Awioniki i Sterowania PRz - grant „Latający Obserwator Terenu” – wdrożenie oprogramowania autonomicznego sterowania lotem dla samolotu bezzałogowego na system operacyjny czasu rzeczywistego



Współpraca z Airbus i GE

LOT – stanowisko laboratoryjne



LOT – aplikacja

Samolot bezzałogowy

- Możliwość autonomicznego lotu
- Zdolność do obserwacji terenu
- Zdolność do strumieniowego przesyłania danych
- Możliwość zdalnego sterowania ze stacji naziemnej

Dwustronna Komunikacja

Mobilna stacja naziemna

- Możliwość monitorowania lotu
- Możliwość „przejęcia" samolotu
- Akwizycja danych

LITERATURA

- http://wazniak.mimuw.edu.pl/
- **Linux : programowanie / Neil Matthew, Richard Stones, RM, 1999**
- **Linux. Niezbędnik programisty/ John Fusco, Helion 2009**
- **Zaawansowane programowanie w systemie Linux / Neil Matthew, Richard Stones, Helion, 2002**
- **Systemy czasu rzeczywistego QNX6 Neutrino / Jędrzej Ułasiewicz, Wydawnictwo btc, 2007.**
- Wprowadzenie do obliczeń równoległych / Zbigniew Czech, PWN, 2010
- Programowanie w Linuksie / K. Kuźniar, K. Lal, T. Rak, Helion, 2012
- Podstawy programowania współbieżnego i rozproszonego / Mordechai Ben-Ari, WNT, 2009
- Java 2. Techniki zaawansowane. Wydanie II /Cay Horstmann, Gary Cornell, Helion, 2005
- Systemy operacyjne czasu rzeczywistego / Szymczyk P. Wydawnictwa AGH, Kraków 2003
- Modele i metody inżynierii oprogramowania systemów czasu rzeczywistego/ T. Szmuc, Uczelniane Wydawnictwa Naukowo-Dydaktyczne AGH, Kraków 2001

Programowanie współbieżne

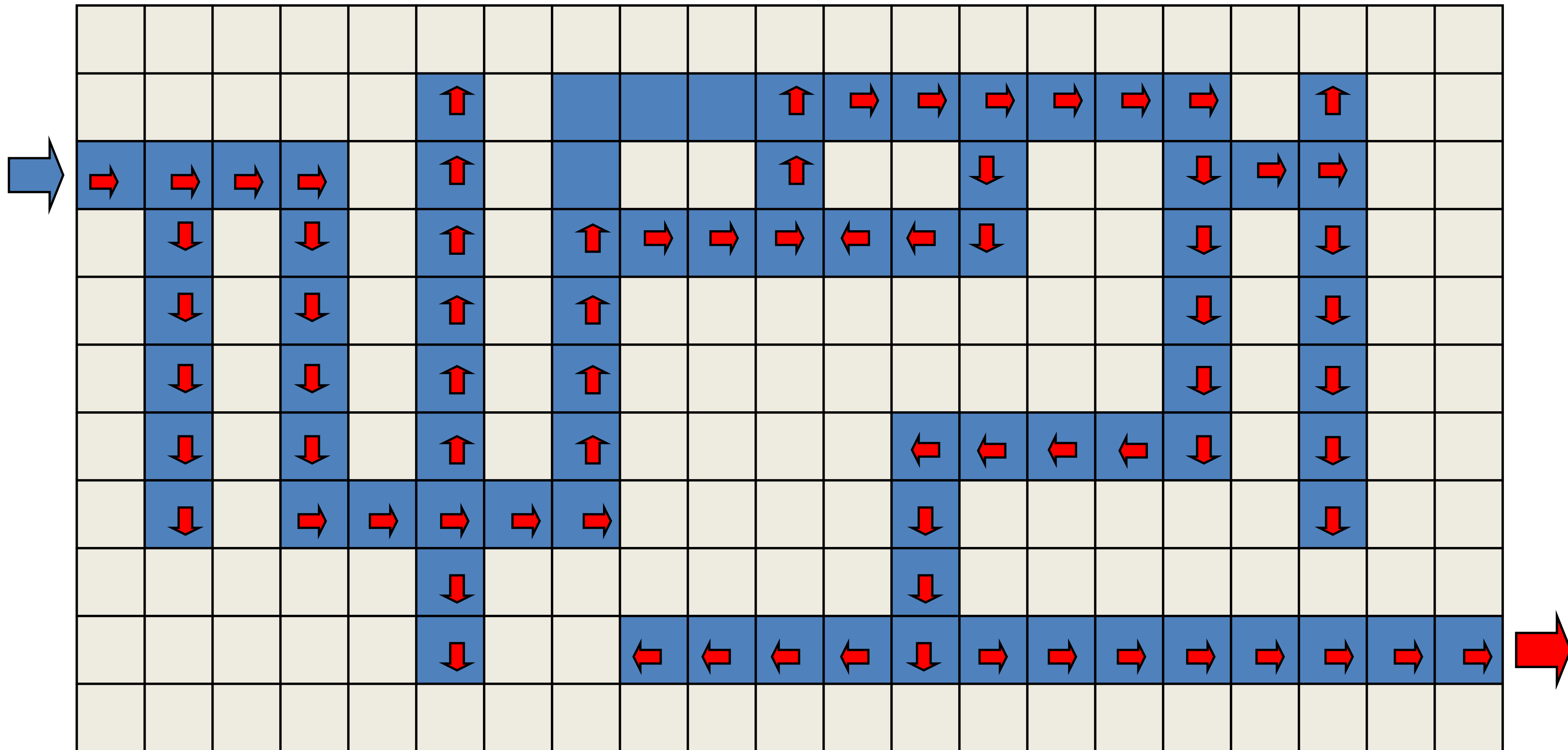
- Programowanie współbieżne dotyczy notacji i technik programowania umożliwiających specyfikację **potencjalnej równoległości** oraz **rozwiązywanie zagadnień synchronizacji i komunikacji**.
- Jak zrealizowana jest możliwość współbieżnego wykonywania programów jest problemem z dziedziny systemów komputerowych i w zasadzie wychodzi poza zakres programowania współbieżnego.
- Paradygmat programowania współbieżnego ustala pewne abstrakcyjne założenia do studiowania współbieżności bez wnikania w szczegóły implementacyjne.

(Ben-Ari 1982)

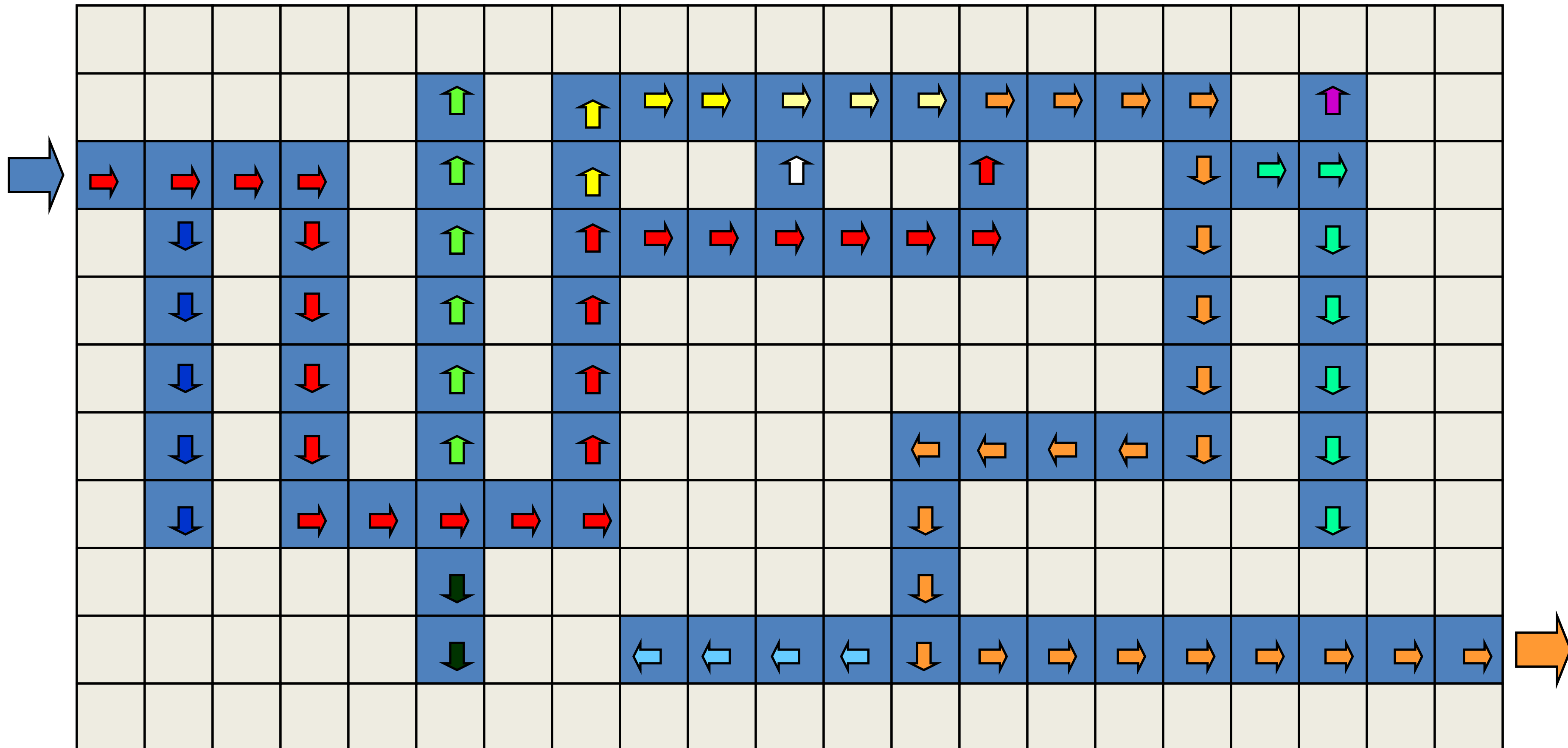
Motywacja

- **Niektóre problemy są z natury współbieżne**, ich rozwiązania dają się łatwo i elegancko wyrazić w postaci niezależnie wykonujących się procedur (system śledzenia celów lotniczych na krążowniku, wprowadzanie kolejnych „jednostek bojowych” lub bohaterów w grach, sortowanie przez scalanie).
- Współczesne systemy operacyjne zezwalają na uruchamianie **wielu procesów jednocześnie**.
- System operacyjny z **podziałem czasu** potrafi wykonywać wiele procesów współbieżnie dzieląc czas procesora między wiele procesów.
- Malejące ceny sprzętu sprzyjają powstawaniu architektur wieloprocesorowych, w których można uzyskać prawdziwą współbieżność, tzn. wykonywać jednocześnie wiele procesów na różnych procesorach.
- Rozpowszechnienie się **sieci komputerowych** stwarza jeszcze inne możliwości współbieżnego wykonywania. Poszczególne obliczenia składające się na duże zadanie mogą być wykonywane na różnych komputerach połączonych siecią.

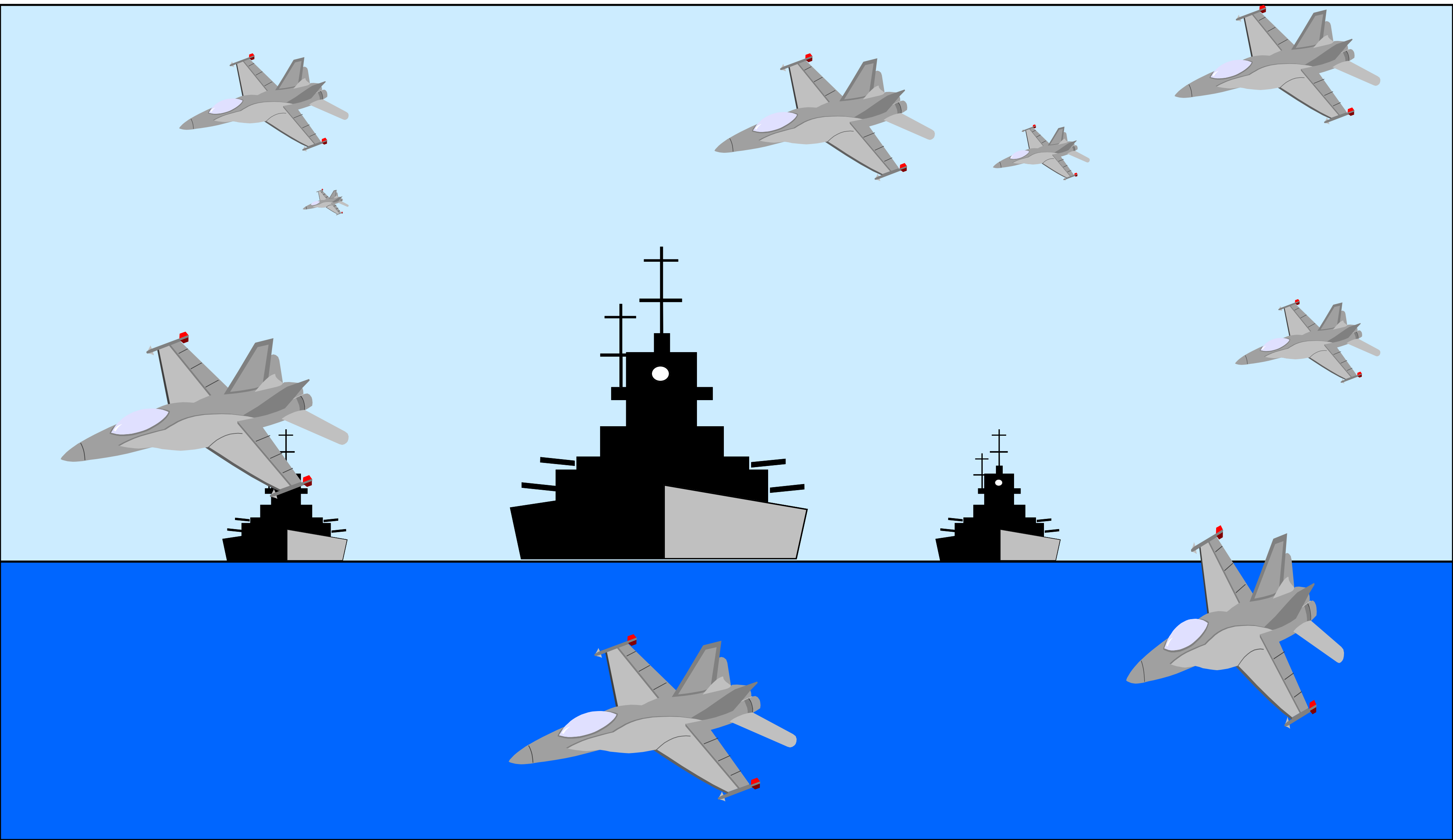
Przykład: Sekwencyjne przeszukiwanie labiryntu



Przykład: Współbieżne przeszukiwanie labiryntu



Jak zaprojektować system śledzenia celów do zestrzelenia?



Przykład: – sortowanie przez scalanie

- **Wersja rekurencyjna:**

```
procedure sortuj (i, j) { sortuje tablicą A[i..j] }
begin
  ...
  m := (i+j) div 2;
  sortuj (i, m);
  sortuj (m+1, j);
  scal (i, m, j);
end
```

- **Wersja współbieżna:**

```
procedure sortuj (i, j) { sortuje tablicą A[i..j] }
begin
  ...
  m := (i+j) div 2;
  cobegin
    sortuj (i, m);
    sortuj (m+1, j);
  coend
  scal (i, m, j);
end
```

Uwagi na tym etapie

- Trwa dyskusja, czy łatwiej dla twórcy oprogramowania jest widzieć je jako zbiór współbieżnych procesów, czy jak zbiór sekwencyjnie wykonywanych operacji.
- Niekiedy współbieżne otoczenie narzuca współbieżne tworzenie oprogramowania, chociaż twórcy aplikacji sekwencyjnych będą dokonywać wszystkich sztuczek, aby ich program został dalej sekwencyjny, bo lepiej go rozumieją (np. twórcy aplikacji wbudowanych zbudują własny pseudo-system operacyjny z zaawansowanym zarządzaniem podprogramami i przerwaniami, a nie zastosują systemu operacyjnego i mechanizmu procesów, bo go nie rozumieją)
- Programowanie współbieżne nie jest panaceum na wszystkie problemy informatyczne:
  – Części algorytmów nie da się zrównoleglić
  – Dowodzenie poprawności programów współbieżnych jest utrudnione
  – …

Ukryta współbieżność

- Użytkownicy współczesnych komputerów osobistych traktują współbieżność jako naturalną własność komputerów
- Twórcy aplikacji biznesowych rzadziej stosują wprost programowanie współbieżne wykorzystując mechanizmy współbieżności osadzone na serwerach: każda sesja z serwerem WWW jest osobnym wątkiem, aplikacje odpytują zduplikowane, fizycznie współbieżnie serwery bazy danych
- Serwery organizuje się jako mniej lub bardziej konstrukcyjnie i programowo powiązane klastry komputerów (łatwość skalowania i zachowania bezpieczeństwa danych)

Równoległość a współbieżność

- *Wykonanie sekwencyjne*. Poszczególne akcje procesu są wykonywane jedna po drugiej. Dokładniej: kolejna akcja rozpoczyna się po całkowitym zakończeniu poprzedniej.
- *Wykonanie równoległe*. Kilka akcji jest wykonywanych w tym samym czasie. Jest to "prawdziwa" współbieżność, możliwa do uzyskania na komputerze z wieloma procesorami.
- *Wykonanie w przeplocie*. Choć jednocześnie odbywa się wykonanie tylko jednej akcji, to jednak jest wiele czynności rozpoczętych i wykonywanych na zmianę krótkimi fragmentami.
- *Wykonanie współbieżne*. Kolejna akcja **rozpoczyna się przed zakończeniem** poprzedniej. Zauważmy, że nie mówimy nic na temat tego, czy akcje te są wykonywane w tym samym czasie czy też w przeplocie. Tak naprawdę wykonanie współbieżne jest abstrakcją równoległości i zawiera w sobie zarówno wykonania równoległe jak i wykonania w przeplocie.

Proces a program

- **Program** to obiekt statyczny --- tekst wykonywanego przez proces kodu.
- **Proces** to obiekt dynamiczny, "żywy", to wykonanie programu w pewnym środowisku lub wstrzymane wykonanie (w oczekiwaniu na jakiś zasób).
- Proces ma przydzielone zasoby (pamięć, urządzenia we/wy) i rywalizuje o dostęp do procesora.
- Procesy wykonują się pod kontrolą systemu operacyjnego.

Stany procesu w systemie operacyjnym

Co nas będzie interesowało?

- Program współbieżny składa się z kilku (co najmniej dwóch) współbieżnych procesów sekwencyjnych, które muszą się ze sobą komunikować lub synchronizować swoje działania.
- Nie interesują nas procesy rozłączne, czyli takie które działają niezależnie od siebie, nie wymagając żadnych działań synchronizacyjnych ani nie wymieniając między sobą danych.
- Zajmiemy się omówieniem mechanizmów udostępnianych przez systemy operacyjne do synchronizacji procesów. Zwrócimy uwagę na pułapki, w jakie może wpaść programista.

Synchronizacja i komunikacja

- Poprawne zachowanie programu współbieżnego zależy od synchronizacji i komunikacji pomiędzy procesami
- **Synchronizacja** to wypełnienie ograniczeń dotyczących kolejności wykonywania pewnych akcji przez procesy (np. pewna akcja wykonywana przez jeden proces może nastąpić tylko wtedy, gdy pewna inna akcja została wykonana w innym procesie)
- **Komunikacja** to przekazywanie informacji od jednego procesu do innego
- Te dwa pojęcia są ze sobą „splecione" ponieważ komunikacja nie może się odbyć bez synchronizacji, a synchronizacja może być potraktowana jako komunikacja bez wymiany danych
- Wymiana danych jest zwykle oparta na **współdzieleniu zmiennych** albo **przekazywaniu wiadomości**

Wątki

- W przypadku budowania aplikacji współbieżnych na bazie pojedynczego komputera jednym z mechanizmów dostarczanym przez system operacyjny jest możliwość tworzenia aplikacji wielowątkowych
- W obrębie jednej aplikacji tworzone są współbieżnie wykonywane zadania, które zwykle współdzielą przydzielony aplikacji obszar pamięci i zasoby
- Wątki potencjalnie mogą efektywnie wykorzystywać wszystkie sprzętowe mechanizmy wspierające działanie aplikacji współbieżnych – wykonywanie na różnych rdzeniach, możliwość efektywnego wykonywania wielu wątków w jednym rdzeniu (np. hyper-threating)

Systemy rozproszone

- System rozproszony jest to zestaw niezależnych komputerów, sprawiający na jego użytkownikach wrażenie jednego, logicznie zwartego systemu
- Charakter systemu rozproszonego mogą mieć systemy wieloprocesorowe, gdzie każdy z procesorów dysponuje własną pamięcią operacyjną
- Komunikacja w aplikacjach dla tego typu systemów opiera się na przekazywaniu komunikatów
- Istnieje oprogramowanie pozwalające widzieć system rozproszony jako pewien wirtualny komputer
- Innym podejściem do tworzenia oprogramowania dla tego typu systemów są mechanizmy lokowania pewnych obliczeń na wskazanych komputerach (zdalne wykonywanie procedur)

Własność bezpieczeństwa (zapewniania)

- Program współbieżny musi mieć własność bezpieczeństwa zwaną również własnością zapewniania. Najkrócej tę własność można wyrazić następująco:

  nigdy nie dojdzie do niepożądanej sytuacji

- Popatrzmy na przykład. Przypuśćmy, że rozpatrujemy ruchliwe skrzyżowanie w środku dużego miasta. Procesy to samochody usiłujące przez to skrzyżowanie przejechać (na wprost). Własność bezpieczeństwa wyraża fakt, że nie dojdzie do kolizji. Możemy ją wyrazić tak:

  nigdy na skrzyżowaniu nie będą jednocześnie samochody jadące w kierunku wschód-zachód i północ-południe.

- W omawianym przykładzie skrzyżowania własność bezpieczeństwa można bardzo łatwo zagwarantować: nie wpuszczać na skrzyżowanie żadnego samochodu! Mamy wtedy bardzo bezpieczne rozwiązanie, ale ... trudno uznać je za poprawne. Z tego powodu poprawny program współbieżny powinien mieć także inną własność...

Własność żywotności

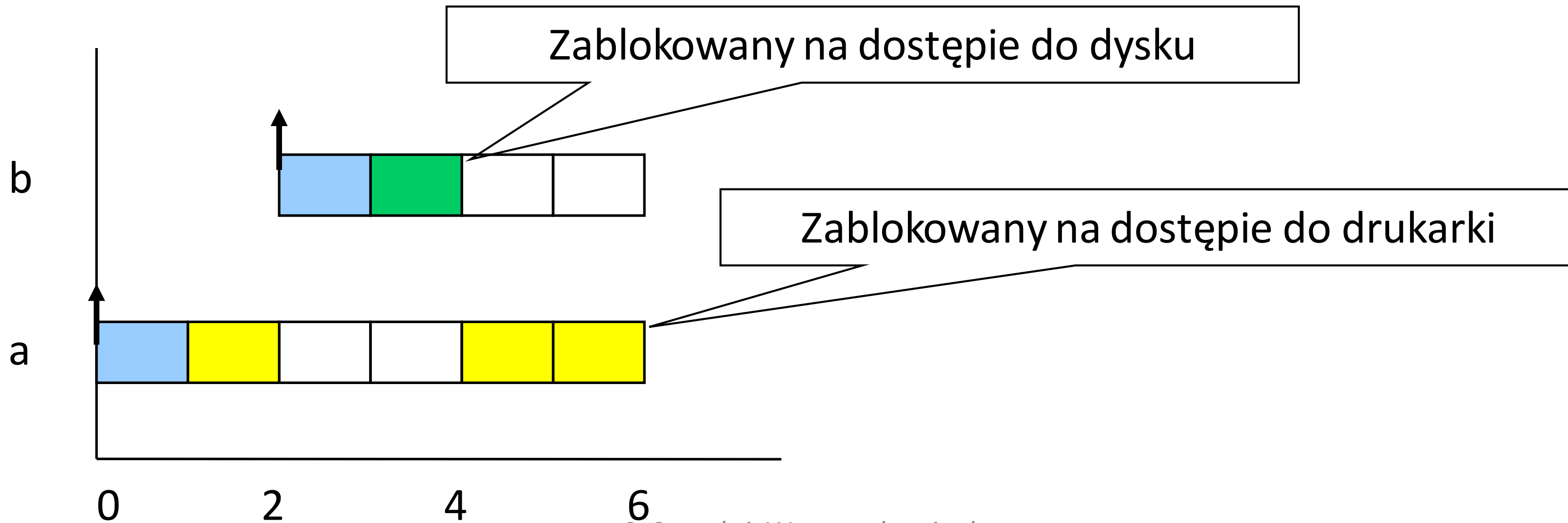
- Własność żywotności można wyrazić tak:

  jeśli proces chce coś zrobić, to w końcu mu się to uda

- W przykładzie skrzyżowania oznacza to, że każdy samochód, który chce przez skrzyżowanie przejechać, w końcu przez to skrzyżowanie przejedzie.

Zakleszczenie

- Jest to globalny brak żywotności. Nic się nie dzieje, system nie pracuje, oczekując na zajście zdarzenia, które nigdy nie zajdzie. Oto klasyczny przykład zakleszczenia. Przypuśćmy, że w systemie wykonują się dwa procesy, które w pewnym momencie muszą odczytać dane z dysku i wydrukować je na drukarce. Proces *P1* prosi system o przydzielenie mu dysku i otrzymuje zgodę. Zanim jednak poprosi o drukarkę, proces *P2* otrzymuje drukarkę. Teraz proces *P1* czeka aż dostanie drukarkę, a proces *P2* czeka na dysk. Żaden nie chce zrezygnować z zasobu, który już otrzymał, więc oczekiwanie będzie trwało w nieskończoność.



Zagłodzenie

- Jest to lokalny brak żywotności.
- Przypuśćmy, że w przedstawionym problemie skrzyżowania, procesy jadące mogą wjechać na skrzyżowanie zawsze wtedy, gdy są na nim inne procesy jadące w tym samym kierunku lub gdy skrzyżowanie jest puste. Załóżmy, że jako pierwszy zjawia się proces jadący na kierunku wschód-zachód. Oczywiście wjeżdża na skrzyżowanie. Jeśli teraz pojawi się proces jadący z północy, to oczywiści musi poczekać. Jednak jeśli zanim ze skrzyżowania zjedzie pierwszy proces pojawi się inny proces jadący ze wschodu, to będzie mógł wjechać na skrzyżowanie. Jeśli sytuacja się będzie ciągle powtarzać, to skrzyżowanie nigdy nie będzie puste --- będą na nim ciągle samochody jadące na kierunku wschód-zachód, co doprowadzi do zagłodzenia procesów drugiej grupy. W systemie coś się ciągle dzieje. Część procesów pracuje normalnie. Brak żywotności dotyka tylko niektóre procesy.

Inne pożądane własności

- Przygotowując program współbieżny nie przyjmujemy żadnym założeń dotyczących czasu. Nie interesuje nas jak długo będzie trwała dana czynność, a jedynie to, czy się skończonym czasie zakończy, czy wystąpi jakaś reakcja na nią itp.
- Wyjątek stanowią systemy czasu rzeczywistego, gdzie na wykonanie obliczeń współbieżnych nakłada się ograniczenia czasowe
- Nie wolno zakładać niepodzielności danych instrukcji
- Czasami narzucające się rozwiązanie problemu synchronizacyjnego jest bezpieczne i żywotne, ale jest nie do przyjęcia z innych powodów. Załóżmy, że w przykładzie ze skrzyżowaniem występują dwa procesy (jeden jadący ciągle z północy), drugi jadący ciągle ze wschodu. Przyjmijmy, że każdy z nich w pętli nieskończonej przejeżdża skrzyżowanie, po czym wraca inną drogą itd. Narzucające się rozwiązanie polega na tym, aby wpuszczać te samochody na skrzyżowanie na zmianę. Jeśli jednak jeden z nich chciałby przejeżdżać przez skrzyżowanie raz na dwa dni, a drugi raz na godzinę, to takie rozwiązanie powodowałoby, że drugi oczekiwałby bezczynnie przez prawie cały czas swojego działania. Taką sytuację nazywamy zbyt ścisłym powiązaniem procesów i będziemy ją również traktować jak błąd.

Problem wzajemnego wykluczania

```
process P;
begin
  while true do
  begin
    własne_sprawy;
    protokół_wstępny;
    sekcja_krytyczna;
    protokół_końcowy;
  end
end;
```

- Procedura **własne_sprawy** to fragment kodu, który nie wymaga żadnych działań synchronizacyjnych. Proces może wykonywać własne sprawy dowolnie długo (nawet nieskończenie długo), może ulec tam awarii.
- **sekcja_krytyczna** to fragment programu, który może być jednocześnie wykonywany przez co najwyżej jeden proces. Zakładamy, że każdy proces, który wchodzi do sekcji krytycznej, w skończonym czasie z niej wyjdzie. Oznacza to w szczególności, że podczas pobytu procesu w sekcji krytycznej nie wystąpi błąd, proces nie zapętli się tam. Zadanie polega na napisaniu protokołu wstępnego i protokołu końcowego tak, aby:

1. W sekcji krytycznej przebywał co najwyżej jeden proces jednocześnie (bezpieczeństwo).
2. Każdy proces, który chce wykonać sekcję krytyczną w skończonym czasie do niej wszedł (żywotność).

Producenci i konsumenci (1)

- W systemie działa *P>0* procesów, które produkują pewne dane oraz *K>0* procesów, które odbierają dane od producentów. Między producentami a konsumentami może znajdować się bufor o pojemności *B*, którego zadaniem jest równoważenie chwilowych różnic w czasie działania procesów. Procesy produkujące dane będziemy nazywać producentami, a procesy odbierające dane --- konsumentami. Zadanie polega na synchronizacji pracy producentów i konsumentów, tak aby:
  1. Konsument oczekiwał na pobranie danych w sytuacji, gdy bufor jest pusty. Gdybyśmy pozwolili konsumentowi odbierać dane z bufora zawsze, to w sytuacji, gdy nikt jeszcze nic w buforze nie zapisał, odebrana wartość byłaby bezsensowna.
  2. Producent umieszczając dane w buforze nie nadpisywał danych już zapisanych, a jeszcze nie odebranych przez żadnego konsumenta. Wymaga to wstrzymania producenta w sytuacji, gdy w buforze nie ma wolnych miejsc.
  3. Jeśli wielu konsumentów oczekuje, aż w buforze pojawią się jakieś dane oraz ciągle są produkowane nowe dane, to każdy oczekujący konsument w końcu coś z bufora pobierze. Nie zdarzy się tak, że pewien konsument czeka w nieskończoność na pobranie danych, jeśli tylko ciągle napływają one do bufora.
  4. Jeśli wielu producentów oczekuje, aż w buforze będzie wolne miejsce, a konsumenci ciągle coś z bufora pobierają, to każdy oczekujący producent będzie mógł coś włożyć do bufora. Nie zdarzy się tak, że pewien producent czeka w nieskończoność, jeśli tylko ciągle z bufora coś jest pobierane.

Producenci i konsumenci (2)

- Rozpatruje się różne warianty tego problemu:
  1. Bufor może być nieskończony.
  2. Bufor cykliczny może mieć ograniczoną pojemność.
  3. Może w ogóle nie być bufora.
  4. Może być wielu producentów lub jeden.
  5. Może być wielu konsumentów lub jeden.
  6. Dane mogą być produkowane i konsumowane po kilka jednostek na raz.
  7. Dane muszą być odczytywane w kolejności ich zapisu lub nie.
- Problem producentów i konsumentów jest abstrakcją wielu sytuacji występujących w systemach komputerowych, na przykład zapis danych do bufora klawiatury przez sterownik klawiatury i ich odczyt przez system operacyjny.

Czytelnicy i pisarze (1)

- W systemie działa C>0 procesów, które odczytują pewne dane oraz P>0 procesów, które zapisują te dane. Procesy zapisujące będziemy nazywać pisarzami, a procesy odczytujące --- czytelnikami, zaś moment, w którym procesy mają dostęp do danych, będziemy nazywać pobytem w czytelni.

```
process Czytelnik;
begin
 repeat
  własne_sprawy;
  protokół_wstępny_czytelnika;
  CZYTANIE;
  protokół_końcowy_czytelnika
 until false
end
```

```
process Pisarz;
begin
 repeat
  własne_sprawy;
  protokół_wstępny_pisarza;
  PISANIE;
  protokół_końcowy_pisarza
 until false
end
```

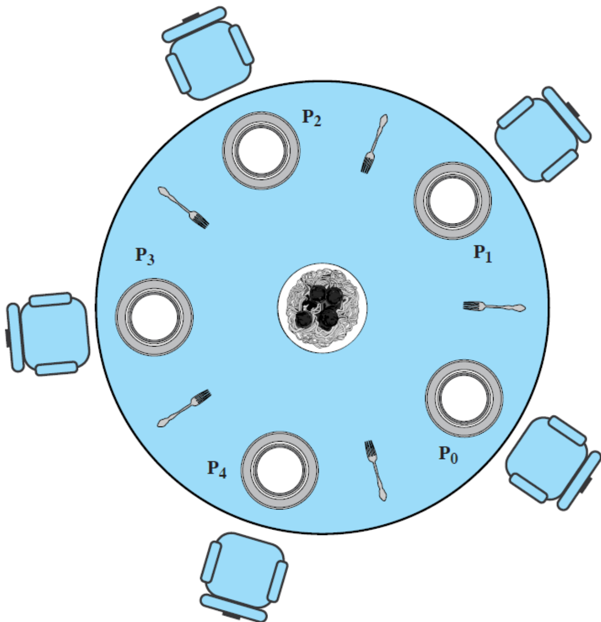
- Zauważmy, że jednocześnie wiele procesów może odczytywać dane. Jednak jeśli ktoś chce te dane zmodyfikować, to rozsądnie jest zablokować dostęp do tych danych dla wszystkich innych procesów na czas zapisu. Zapobiegnie to odczytaniu niespójnych informacji (na przykład danych częściowo tylko zmodyfikowanych).

Czytelnicy i pisarze (2)

- Należy tak napisać protokoły wstępne i końcowe poszczególnych procesów, aby:
  1. Wielu czytelników powinno mieć jednocześnie dostęp do czytelni.
  2. Jeśli w czytelni przebywa pisarz, to nikt inny w tym czasie nie pisze ani nie czyta.
  3. Każdy czytelnik, który chce odczytać dane, w końcu je odczyta.
  4. Każdy pisarz, który chce zmodyfikować dane, w końcu je zapisze.
- Rozpatruje się różne warianty tego problemu:
  1. W czytelni może przebywać dowolnie wielu czytelników.
  2. Czytelnia może mieć ograniczoną pojemność.
  3. Pisarze mogą mieć pierwszeństwo przed czytelnikami (ale wtedy rezygnujemy z żywotności czytelników)
  4. Czytelnicy mogą mieć pierwszeństwo przed pisarzami (ale wtedy rezygnujemy z żywotności pisarzy)

Pięciu filozofów (1)

- Ten problem nie ma praktycznych analogii, jak w przypadku poprzednich klasycznych problemów, ale bardzo dobrze ilustruje problemy występujące przy tworzeniu programów współbieżnych.
- Pięciu filozofów siedzi przy okrągłym stole. Przed każdym stoi talerz. Między talerzami leżą widelce. Pośrodku stołu znajduje się półmisek z rybą. Każdy filozof myśli. Gdy zgłodnieje sięga po widelce znajdujące się po jego prawej i lewej stronie, po czym rozpoczyna posiłek. Gdy się już naje, odkłada widelce i ponownie oddaje się myśleniu.




```
process Filozof (i: 0..4);
begin
  repeat
    myśli;
    protokół_wstępny;
    je;
    protokół_końcowy;
  until false
end;
```

Pięciu filozofów (2)

- Należy tak napisać protokoły wstępne i końcowe, aby:
  1. Jednocześnie tym samym widelcem jadł co najwyżej jeden filozof.
  2. Każdy filozof jadł zawsze dwoma (i zawsze tymi, które leżą przy jego talerzu) widelcami.
  3. Żaden filozof nie umarł z głodu.
- Chcemy ponadto, aby każdy filozof działał w ten sam sposób.