Systemy Operacyjne (2)

Marcin Gogolewski
marcing@wmi.amu.edu.pl

Uniwersytet im. Adama Mickiewicza w Poznaniu

Poznań, 3 listopada 2018

Pojęcie procesu

Proces

Działający program

Uwaga

Może istnieć jednocześnie wiele procesów (niezależnych) wykonujących ten sam program!

Model procesów sekwencyjnych

Założenia (minimalne)

- jeden procesor
- zbiór procesów

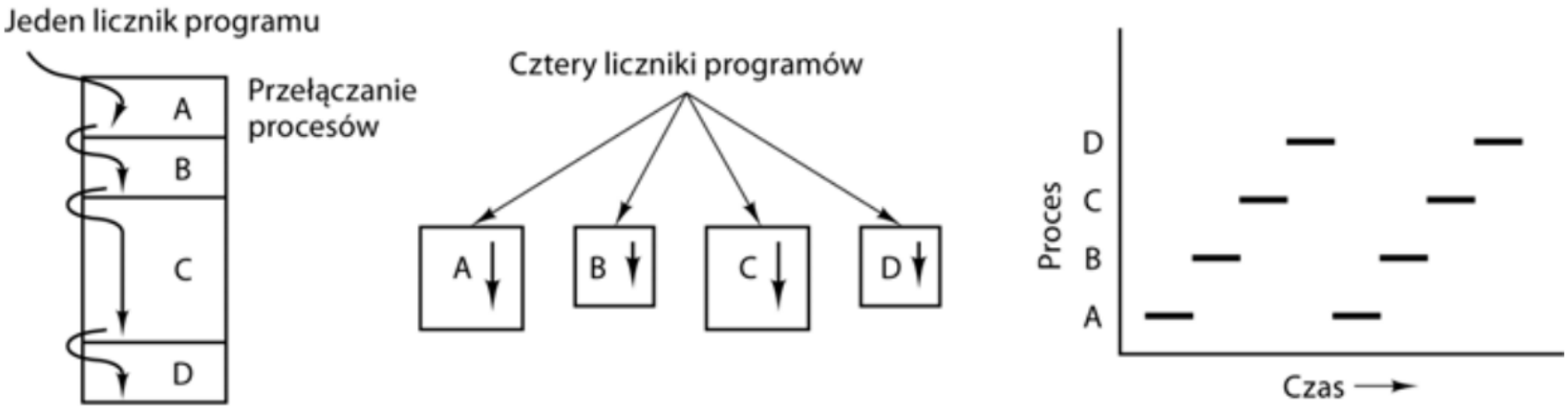
Proces w systemie

Każdy proces w systemie posiada (co najmniej):

- własną kopię stanu rejestrów
- licznik (wskazujący na następną instrukcję do wykonania)
- zmienne (tzn. stos i pozostałe dane w przydzielonej mu pamięci)

Jak działa procesor?

- w danym momencie wykonuje **jeden** program (choć być może kilka instrukcji)
- posiada licznik wskazujący na następną instrukcję



Przełączanie procesów

- zapisanie stanu bieżącego procesu
- załadowanie stanu kolejnego procesu przeznaczonego do wykonania

System operacyjny co jakiś czas (zwykle po wystąpieniu przerwania zegarowego po ustalonym czasie) przejmuje kontrolę nad procesorem, wykonuje konieczne obliczenia i przydziela procesor.

W praktyce wygodniej myśleć o procesach działających jednocześnie (współcześnie jest to często prawdą, bo procesory mają więcej niż jeden rdzeń).

Nie da się przewidzieć rzeczywistego czasu wykonania!

Kiedy tworzone są procesy

- podczas startu systemu
- przez działający proces (np. wywołanie `fork()`)
- przez użytkownika (tzn. za pomocą powłoki)
- uruchomienie zadania wsadowego (np. przez `cron`)

Czasami cały system (poza niewielką częścią odpowiedzialną za niskopoziomową obsługę procesora) składa się z procesów!

Kończenie procesów

- „normalne” (tzn. dobrowolne, zgodne z programem)
- w wyniku błędu (dobrowolnie, np. brak pliku)
- błąd krytyczny (przymusowo, np. dzielenie przez zero)
- przez inny proces (przymusowo, np. `kill -9`)

Hierarchia procesów (pstree)

```
|-lightdm---sh---ssh-agent
|          |     |-xfce4-session---agent---2*[{agent}]
|          |                        |-applet.py
|          |                        |-blueman-applet---3*[{bluer
|          |                        |-guake---bash
|          |                        |        |-bash---geany---2*
|          |                        |        |       |-pstree
|          |                        |        |-bash---mc---bash
|          |                        |        |-gsettings
|          |                        |        |-3*[{guake}]
|          |                        |-kerneloops-appl
|          |                        |-light-locker---3*[{light-l
|          |                        |-polkit-gnome-au---2*[{polk
|          |                        |-update-notifier---3*[{upda
|          |                        |-xfce4-notes---2*[{xfce4-no
```

Start systemu (Linux)

- po załadowaniu jądra uruchamiany `init`
- `init` jest „przodkiem” wszystkich procesów w systemie
- w przypadku utraty rodzica obowiązki przejmuje `init`
- `init` ma PID $==$ 1

Stany procesu

Proces może być w jednym z 3 stanów, z 4 możliwymi przejściami.

1. czeka na dane
2. skończył się przydział procesora
3. otrzymał przydział procesora
4. dane wejściowe gotowe

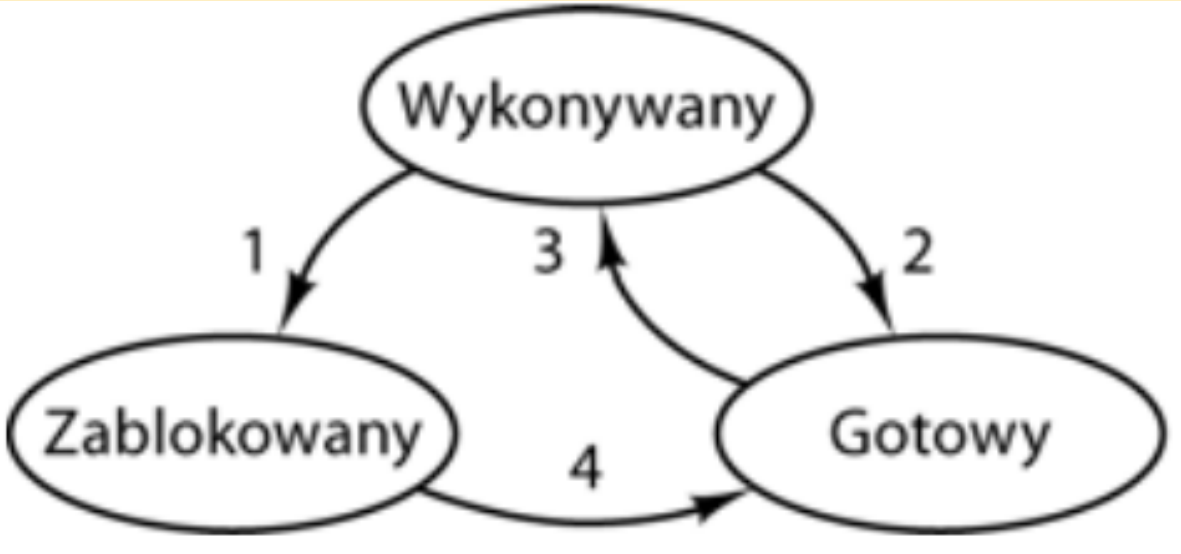


Tabela procesów – zawartość

Zarządzanie procesami	Zarządzanie pamięcią	Zarządzanie plikami
Rejestry	Wskaźnik do informacji segmentu tekstu	Katalog główny
Licznik programu	Wskaźnik do informacji segmentu danych	Katalog roboczy
Słowo stanu programu	Wskaźnik do informacji segmentu stosu	Deskryptory plików
Wskaźnik stosu		Identyfikator użytkownika
Stan procesu		Identyfikator grupy
Priorytet		
Parametry szeregowania		
Identyfikator procesu		
Proces-rodzic		
Grupa procesów		
Sygnały		
Czas rozpoczęcia procesu		
Wykorzystany czas CPU		
Czas CPU procesów-dzieci		
Godzina następnego alarmu		

Obsługa przerwań

- Sprzęt odkłada na stos licznik programu itp.
- Sprzęt ładuje nowy licznik programu z wektora przerwań
- Procedura w języku asemblera zapisuje rejestry
- Procedura w języku asemblera ustawia nowy stos
- Uruchamia się procedura obsługi przerwania w C (np. buforuje dane wejściowe)
- Program szeregujący decyduje o tym, który proces ma być uruchomiony w następnej kolejności
- Procedura w języku C zwraca sterowanie do kodu w asemblerze
- Procedura w języku asemblera uruchamia nowy bieżący proces

Wątki

Wątek

Rodzaj procesu działającego wewnątrz innego procesu, mającego:

- wspólna pamięć (kod)
- zmienne globalne
- otwarte pliki

ale mającego

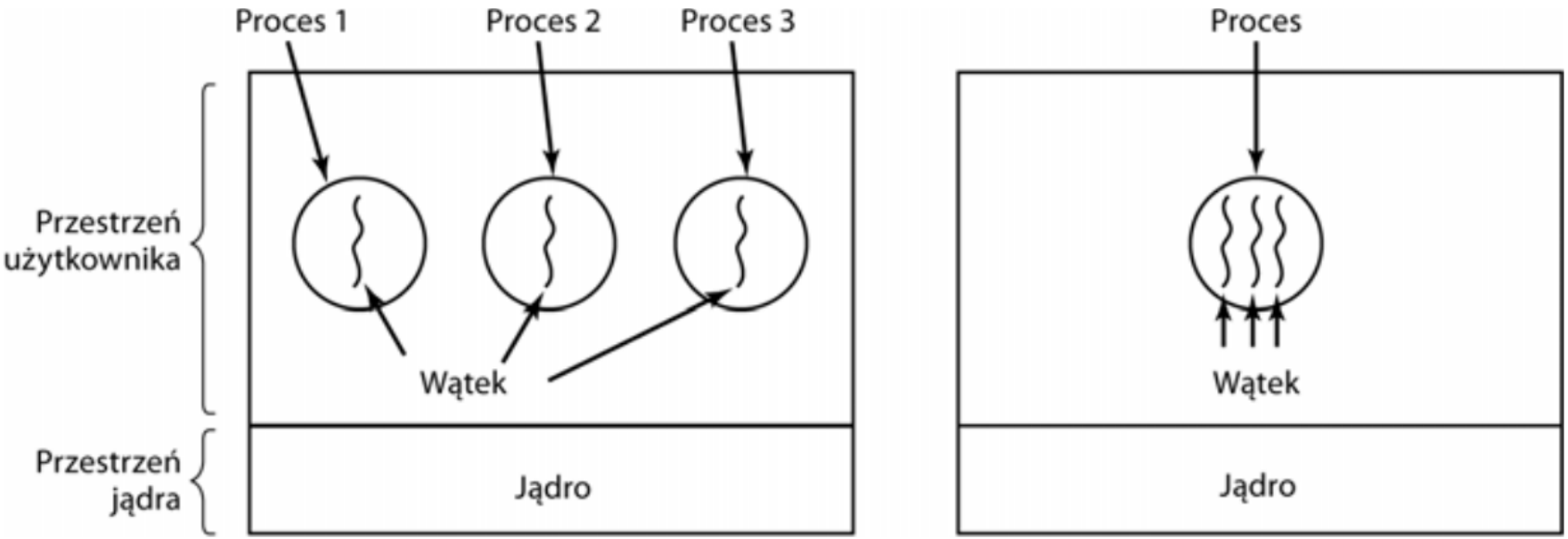
- własny stos (niezależna ścieżka wykonania)
- własny licznik rozkazów

W niektórych systemach stworzenie wątku jest wielokrotnie (nawet 100 krotnie) tańsze od stworzenia nowego procesu.

Zastosowania wątków

- serwery z dużą liczbą użytkowników
- interaktywne programy (np. edytory WYSiWYG)
- jedno z rozwiązań blokujących wywołań systemowych (np. `read`)

Model wątków



Model wątków

Komponenty procesu	Komponenty wątku
Przestrzeń adresowa	Licznik programu
Zmienne globalne	Rejestry
Otwarte pliki	Stos
Procesy-dzieci	Stan
Zaległe alarmy	
Sygnały i procedury obsługi sygnałów	
Informacje dotyczące statystyk	

Interface Pthread (POSIX threads)

Wywołanie obsługi wątku	Opis
`Pthread_create`	Utworzenie nowego wątku
`Pthread_exit`	Zakończenie wątku wywołującego
`Pthread_join`	Oczekiwanie na zakończenie specyficznego wątku
`Pthread_yield`	Zwolnienie procesora w celu umożliwienia działania innemu wątkowi
`Pthread_attr_init`	Utworzenie i zainicjowanie struktury atrybutów wątku
`Pthread_attr_destroy`	Usunięcie struktury atrybutów wątku

Przykład Pthreads

```
void *printhw(void *tid){
  printf("Witaj, Świecie. Pozdrowienia od wątku %d\n, tid);
  pthread exit(NULL);
}
int main(int argc, char *argv[]){
  pthread_t threads[NUMBER_OF_THREADS];
  int status, i;
  for(i=0; i < 10; i++) {
    printf("Tu program główny. Tworzenie wątku %d\n, i);
    status=pthread_create(&threads[i], NULL, printhw, (void*)i);
    if (status != 0) {
       printf("Oops. Funkcja pthread_create zwróciła %d\n, statu
       exit(-1);
    }
  }
  exit(NULL);
}
```

Wyścig (race condition)

Wyścig

Dwa (lub więcej) procesy starają się jednocześnie o dostęp do jednego zasobu.

Uwaga

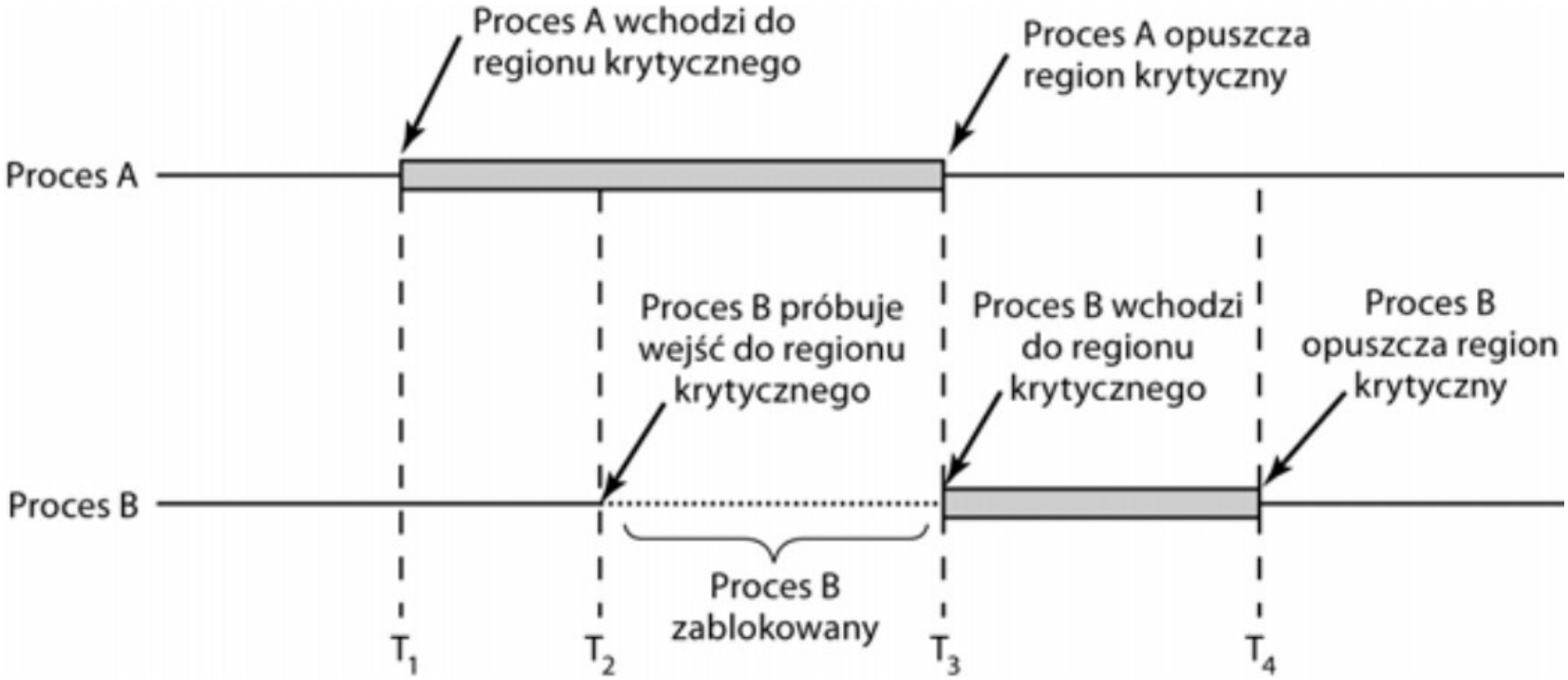
Problem może nie wystąpić podczas testowania (bo ma np. niskie prawdopodobieństwo lub muszą zajść specyficzne warunki), co bardzo komplikuje analizę.

Wyścig – metody unikania

Sekcje krytyczne – założenia

1. Maksymalnie jeden proces może przebywać w **s.k.**
2. Nie można przyjmować żadnych założeń o liczbie, czy szybkości działania procesów
3. Proces działający wewnątrz swojej **s.k.** nie może blokować innych procesów
4. Żaden proces nie powinien oczekiwać w nieskończoność na dostęp do swojej **s.k.**

Sekcje krytyczne



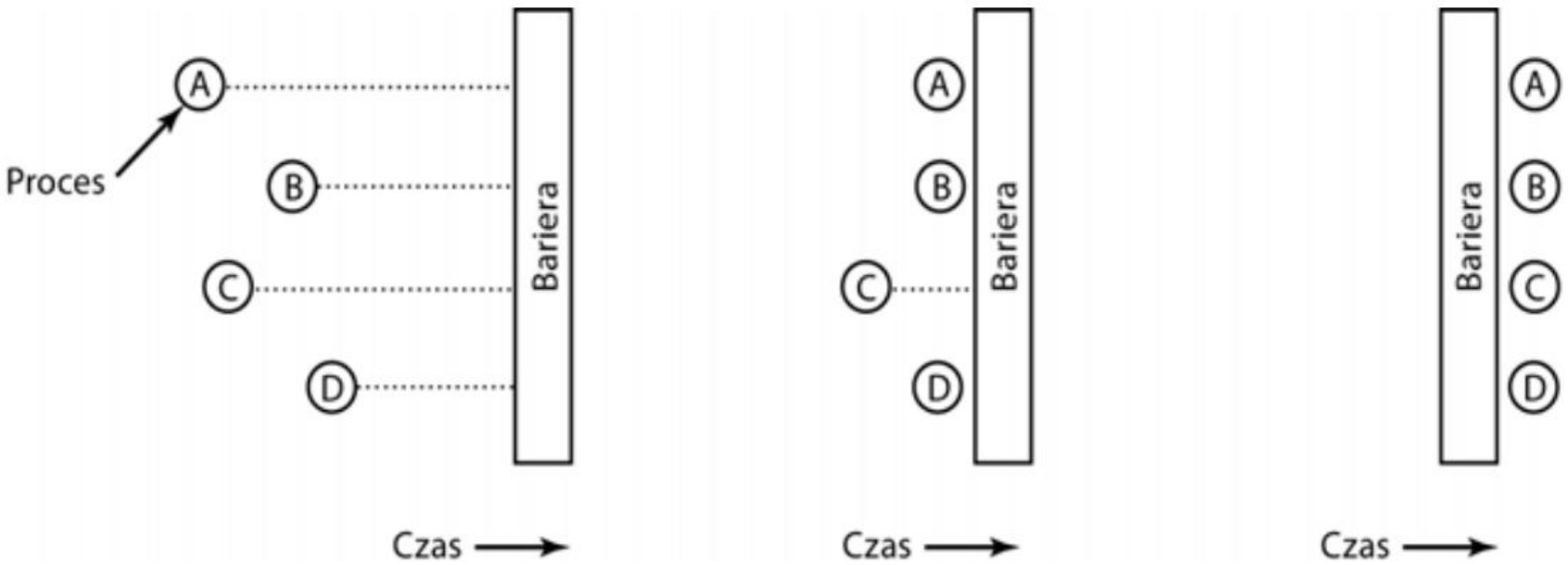
Ale jak to osiągnąć?

Metody wzajemnego wykluczania

- z aktywnym oczekiwaniem
  - wyłączenie przerwań
  - blokowanie zmiennych
  - ścisła naprzemienność
  - algorytm Petersona, instrukcje TSL
- bez aktywnego oczekiwania
  - sleep/wakeup
  - semafory (i muteksy)
  - futeksty
  - monitory

przekazywanie komunikatów (niezawodność?), bariery

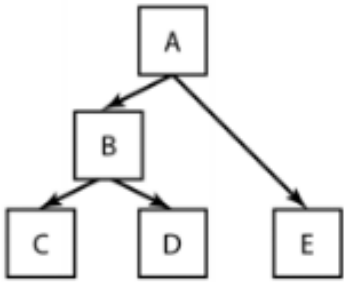
Bariery



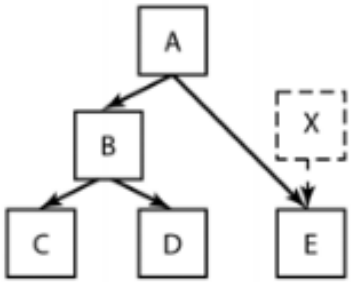
A może dałoby się zrezygnować z blokad?

Metody nieblokujące (np. typu RCU)

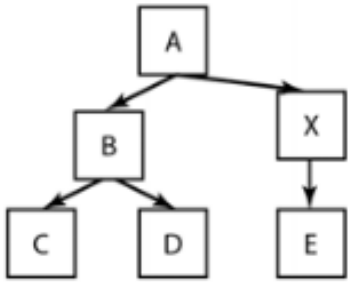
Dodanie węzła:



(a) Drzewo w postaci początkowej

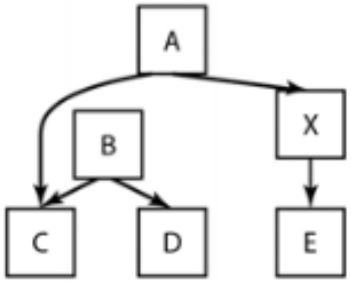


(b) Zainicjowanie węzła X o podłączenie węzła E do węzła X. Operacja nie ma wpływu na czytelników węzłów A i E

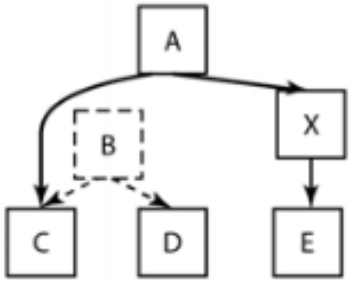


(c) Po całkowitym zainicjowaniu węzła X podłączamy X do A. Czytelnicy, którzy aktualnie są w węźle E, muszą odczytać starą wersję, natomiast czytelnicy w węźle A pobiorą nową wersję drzewa

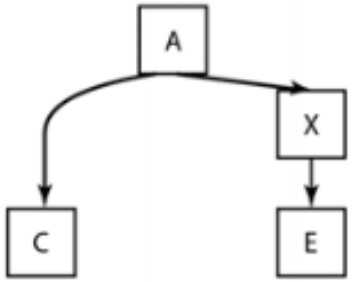
Usuwanie węzłów:



(d) Rozdzielenie B od A. Zwróćmy uwagę, że w B nadal mogą być czytelnicy. Wszystkie procesy-czytelnicy w B będą widziały starą wersję drzewa, natomiast wszystkie procesy-czytelnicy w A będą widziały nową wersję

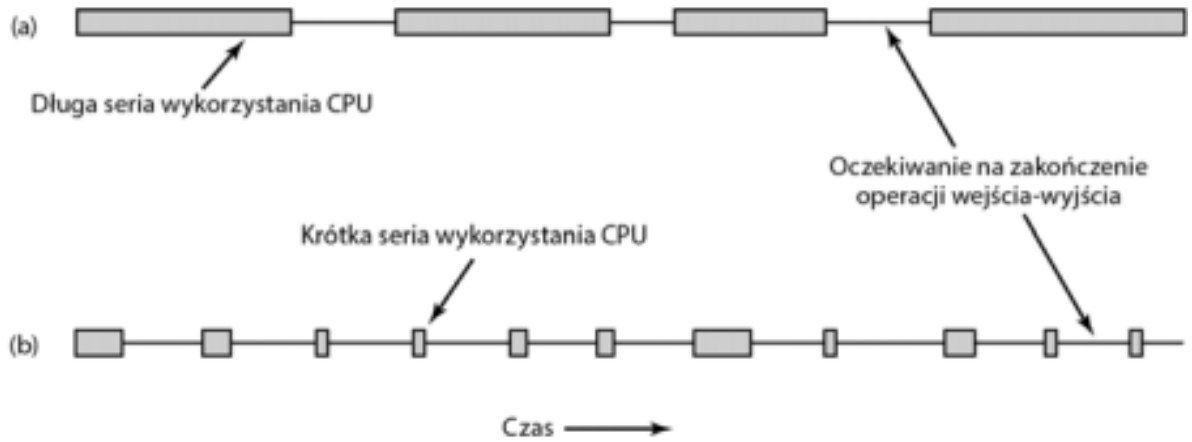


(e) Oczekiwanie do uzyskania pewności, że wszystkie procesy-czytelnicy opuściły węzły B i C. Do tych węzłów nie będzie już dostępu



(f) Teraz można bezpiecznie usunąć węzły B i D

Problem szeregowania



- Kiedy?
- W jaki sposób?

Algorytmy szeregowania

Wszystkie systemy

Sprawiedliwość — przydzielanie każdemu procesowi odpowiedniego czasu procesora

Wymuszanie strategii — sprawdzanie, czy jest przestrzegana zamierzona strategia.

Równowaga — dbanie o to, by wszystkie części systemu były zajęte.

Systemy wsadowe

Przepustowość — maksymalizacja liczby wykonywanych zadań na godzinę.

Czas cyklu przetwarzania — minimalizacja czasu pomiędzy rozpoczęciem pracy procesu, a jej zakończeniem.

Wykorzystanie procesora — dbanie o ciągłą zajętość procesora.

Systemy interaktywne

Czas odpowiedzi — szybka odpowiedź na żądania.

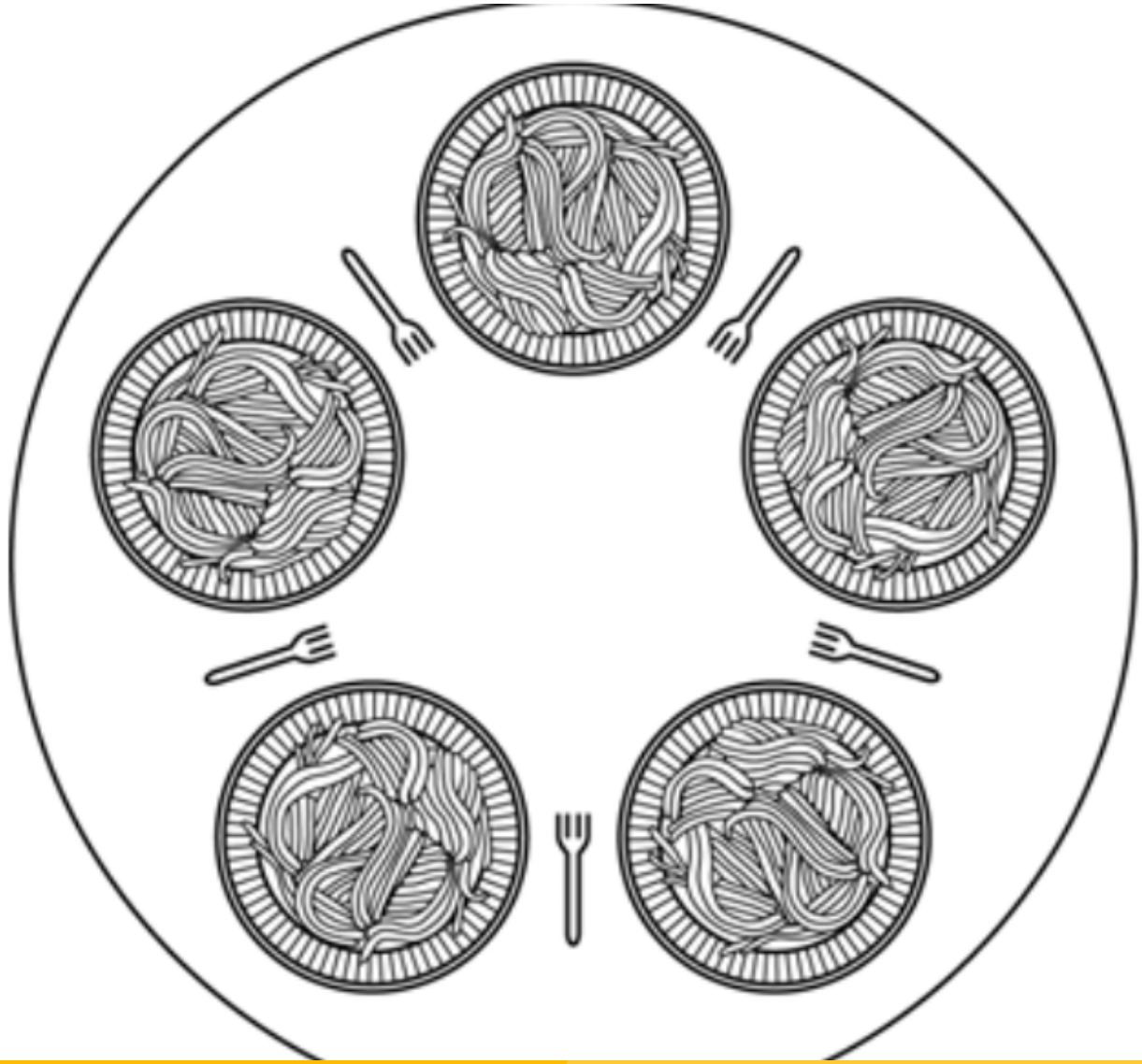
Proporcjonalność — spełnianie oczekiwań użytkowników.

Systemy czasu rzeczywistego

Dotrzymywanie terminów — unikanie utraty danych.

Przewidywalność — unikanie degradacji jakości w systemach multimedialnych.

Klasyczny problem – ucztujący filozofowie



Klasyczny problem – ucztujący filozofowie

Błędne rozwiązanie

```
#define N 5
void philosopher(int i){
  while (TRUE) {
    think( );
    take_fork(i);
    take_fork((i+1) % N);
    eat( );
    put_fork(i);
    put_fork((i+1) % N);
  }
}
```

Inne problemy synchronizacji

Problem czytelników i pisarzy

Zasady dostępu do struktury (np. bazy danych) są następujące: W danym momencie dowolna liczba czytelników może czytać, jeżeli żaden pisarz nie modyfikuje struktury. Modyfikacja jest możliwa wyłącznie przez jednego pisarza jednocześnie.

Problem

Nie zagłodzić pisarzy i czytelników.