Problemy czytelników i pisarzy oraz 5 ucztujących filozofów

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

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

oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

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)

Rozwiązanie I – uprzywilejowanie czytelników

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *writer_thread_fun(void *arg) {

   printf("Writer %d started...\n",(int) ((int *) arg));
   sleep(1);
   while(1)
   {  //pthread_mutex_lock(&wsem);
      sem_wait(&wsem);
         shared_data++;
      //pthread_mutex_unlock(&wsem);
      sem_post(&wsem);
      sleep(3);
   }
   pthread_exit(0);
}
```

```
void *reader_thread_fun(void *arg) {
   int data_read;
   printf("Reader %d started...\n",(int) ((int *) arg));
   sleep(1);
   while(1)
   {  pthread_mutex_lock(&x);
         readcount++;
         if(readcount==1)  sem_wait(&wsem);
         printf("Readcount=%d\n",readcount);
      pthread_mutex_unlock(&x);
      data_read=shared_data;
      printf("Reader %d  consumed %d.\n",(int) ((int *)
arg),data_read);
      pthread_mutex_lock(&x);
         readcount--;
         printf("Readcount=%d\n",readcount);
         if(readcount==0) //pthread_mutex_unlock(&wsem);
                  sem_post(&wsem);
      pthread_mutex_unlock(&x);
      sleep(1);
   }
   pthread_exit(0);
}
// Dalej: powołanie czytelników i pisarzy + inicjalizacja
// semaforów
```

Uwagi

- Semafor *wsem* jest wykorzystywany do wymuszenia wzajemnego wykluczania.
- Dopóki jeden pisarz ma dostęp do współdzielonego obszaru danych, żaden pisarz ani żaden czytelnik nie może mieć do niego dostępu.
- Proces czytelnika wykorzystuje semafor *wsem*, by wymusić wzajemne wykluczenie.
- Jednakże aby umożliwić dostęp wielu czytelników, wymaga się, by w sytuacji, gdy żaden czytelnik nie odczytuje danych, pierwszy czytelnik, który próbuje odczytać poczekał na semafor *wsem.*
- Kiedy przynajmniej jeden czytelnik odczytuje dane, kolejni czytelnicy nie muszą czekać, zanim uzyskają dostęp.
- Zmienna globalna *readcount* jest stosowana do śledzenia liczby czytelników, a semafor (mutex) x jest wykorzystywany, by upewnić się, że zmienna *readcount* jest poprawnie aktualizowana.
- Rozwiązanie ma wadę: W sytuacji gdy jeden czytelnik rozpoczął uzyskiwanie dostępu do danych, czytelnicy mogą kontrolować dane pod warunkiem, że przynajmniej jeden czytelnik odczytuje dane. Ro z kolei grozi **zagłodzeniem** pisarzy.

Rozwiązanie II – uprzywilejowanie pisarzy

```
void *writer_thread_fun(void *arg) {
   printf("Writer %d started...\n",(int)((int *) arg));
   sleep(2);
   while(1)
   {   pthread_mutex_lock(&y);
          writecount++;
          if(writecount==1) sem_wait(&rsem);
       pthread_mutex_unlock(&y);
       sem_wait(&wsem);
          shared_data++;
       sem_post(&wsem);
       pthread_mutex_lock(&y);
          writecount--;
          if(writecount==0) sem_post(&rsem);
       pthread_mutex_unlock(&y);
       sleep(3);
   }
   pthread_exit(0);
}
```

```
void *reader_thread_fun(void *arg) {
   int data_read;
   printf("Reader %d started...\n",(int)((int *) arg));  sleep(2);
   while(1)
   {  pthread_mutex_lock(&z);
         sem_wait(&rsem);
            pthread_mutex_lock(&x);
               readcount++;
               if(readcount==1) sem_wait(&wsem);
               printf("Readcount=%d\n",readcount);
            pthread_mutex_unlock(&x);
         sem_post(&rsem);
      pthread_mutex_unlock(&z);
      data_read=shared_data;
      printf("Reader %d  consumed %d.\n",(int)((int *)
arg),data_read);
      pthread_mutex_lock(&x);
         readcount--;
         printf("Readcount=%d\n",readcount);
         if(readcount==0) sem_post(&wsem);
      pthread_mutex_unlock(&x);
      //sleep(1);
   }
   pthread_exit(0);}

// Dalej: powołanie czytelników i pisarzy + inicjalizacja
// semaforów
```

Uwagi

- Rozwiązanie gwarantuje, że żadni nowi czytelnicy nie uzyskają dostępu do obszaru danych, jeśli przynajmniej jeden pisarz zadeklarował, że chce zrealizować operację zapisu.
- W przypadku pisarzy zostały dodane następujące semafory oraz zmienne:
  - Semafor *rsem,* który blokuje wszystkich czytelników, jeśli przynajmniej jeden pisarz spróbuje uzyskać dostęp do obszaru danych
  - Zmienna *writecount* kontrolująca ustawienia semafora *rsem*
  - Semafor y, który steruje aktualizacją zmiennej *writecount*
- W przypadku czytelników potrzebny jest dodatkowy semafor. Nie można dopuścić do powstania dużej kolejki na semaforze *rsem*, bowiem w przeciwnym razie pisarze nie będą w stanie wskoczyć do kolejki. Tak więc, tylko jeden czytelnik może się znaleźć w kolejce semafora *rsem*. Wszyscy dodatkowi czytelnicy muszą być skierowani do kolejki semafora z natychmiast przed oczekiwaniem na semafor *rsem*.

Blokady czytelników i pisarzy w POSIX (1)

- Problem czytelników i pisarzy jest na tyle powszechny, że w wielu systemach do programowania współbieżnego zostały zaproponowane specjalne blokady lub semafory wspierające konstruowanie oprogramowania realizującego ten problem
- Przykładem mogą być blokady czytelników i pisarzy POSIX:

Inicjacja blokady

`int pthread_rwlock_init(pthread_rwlock_t * rwlock,`
`pthread_rwlockattr_t * attr`)

- **`rwlock`** Zadeklarowana i zainicjowana zmienna typu **`pthread_rwlock_t`**
- **`attr`** Atrybuty blokady lub NULL gdy maja być domyślne

Zajęcie blokady do odczytu

`int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock`)

- Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu.
- Zajmuje blokadę do odczytu gdy nie została już wcześniej zajęta do odczytu.

Blokady czytelników i pisarzy w POSIX (2)

Zajęcie blokady do zapisu

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

- Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu lub odczytu. Gdy nie jest zajęta to zajmuje blokadę do zapisu.

Zwolnienie blokady

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
```

Funkcja zdejmuje blokadę nałożoną jako ostatnią przez bieżący wątek. Jeżeli istnieją inne blokady założone na obiekt to pozostają. Jeżeli jest to ostatnia blokada i istnieją wątki czekające na jej zwolnienie to jeden z nich zostanie odblokowany. Wybór wątku do zwolnienia zależy to od implementacji.

Blokady czytelników i pisarzy w POSIX (3)

Nieblokujące zajęcie blokady do zapisu

`int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock`)

- Gdy blokada jest wolna następuje jej zajęcie do zapisu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego i zwraca kod błędu.

Nieblokujące zajęcie blokady do odczytu

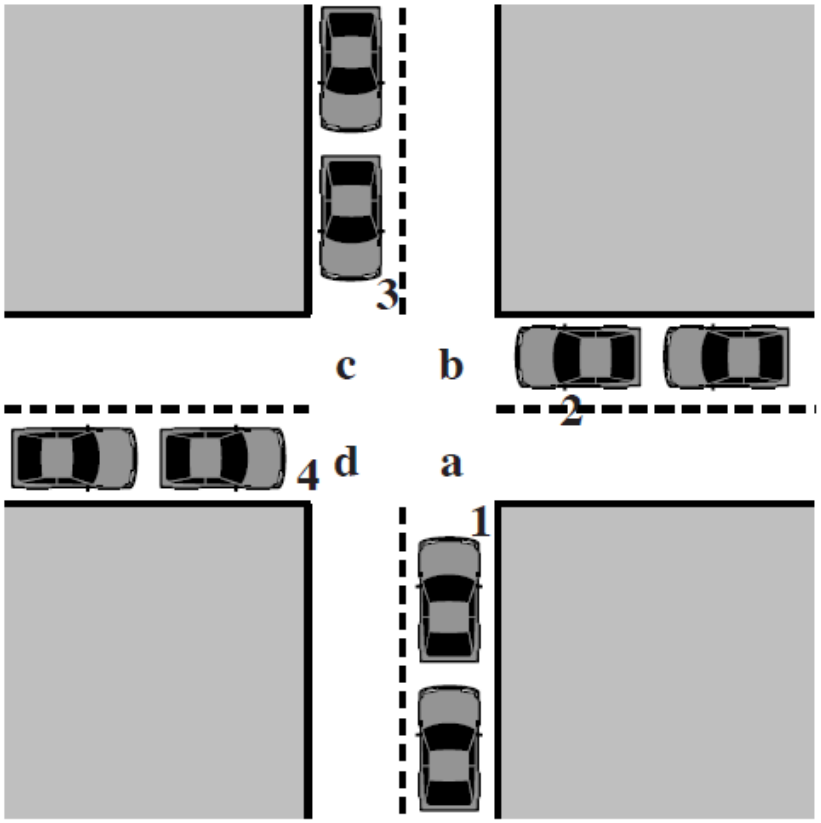
`int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock`)

- Gdy blokada jest wolna lub zajęta do odczytu następuje jej zajęcie do odczytu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego i zwraca kod błędu.

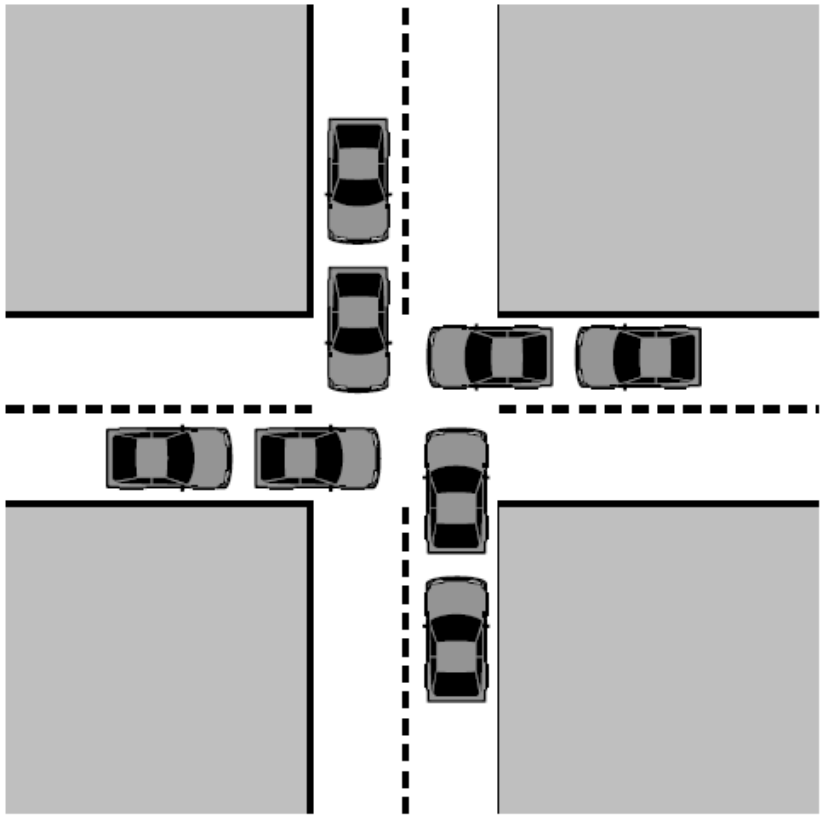
Skasowanie blokady

`int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)`

Impas/Zakleszczenie



Możliwy impas/zakleszczenie



Impas/zakleszczenie

Czym jest impas

- Impas można zdefiniować jako trwałe zablokowanie zestawu procesów, które rywalizują o zasoby lub komunikują się ze sobą nawzajem
- Do impasu dochodzi, każdy każdy proces zestawu jest zablokowany i oczekuje na zdarzenie (zazwyczaj na zwolnienie żądanego zasobu), które może zaistnieć tylko, jeśli zostanie zainicjowane przez inny proces z zestawu procesów.
- Impas jest stanem trwałym, ponieważ żadne ze zdarzeń nigdy nie zachodzi.
- W przeciwieństwie do innych problemów współbieżności, nie istnieje skuteczne rozwiązanie takiej sytuacji.

Przykład impasu/zakleszczenie

a

-

-

Wait(s_Q)

Wait(s_V)

Signal(s_V)

Signal(s_Q)

b

-

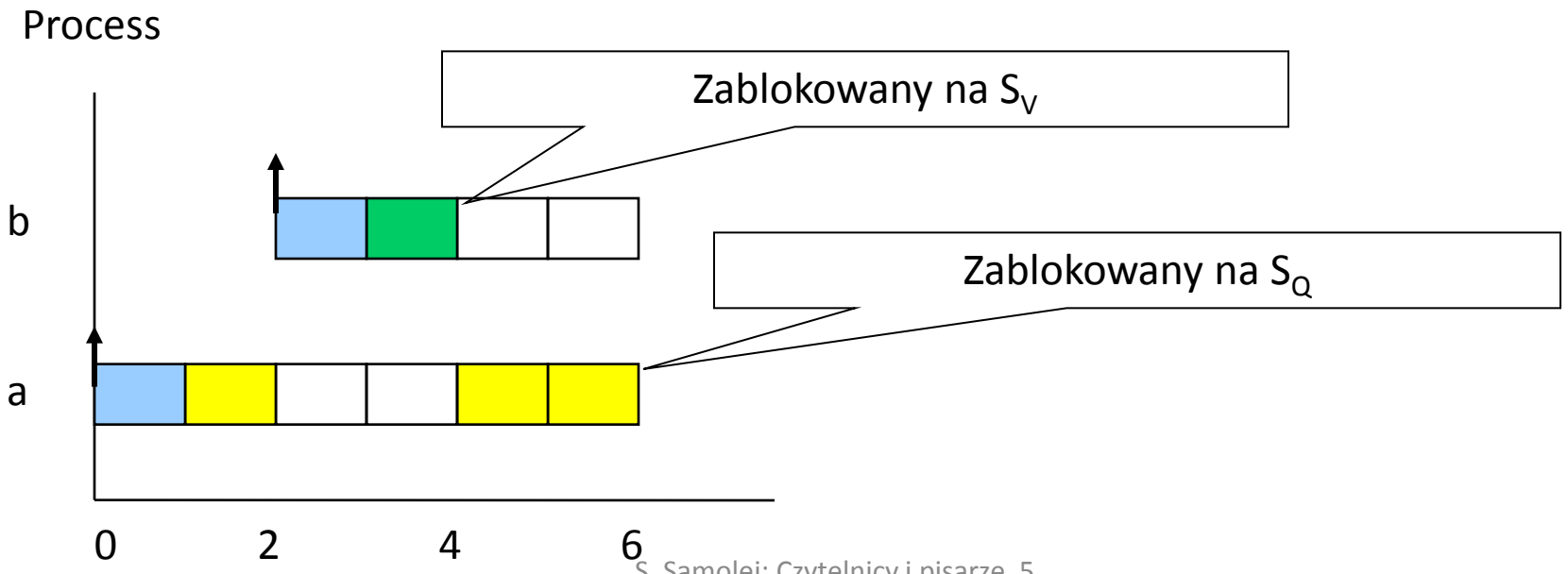
-

Wait(s_V)

Wait(s_Q)

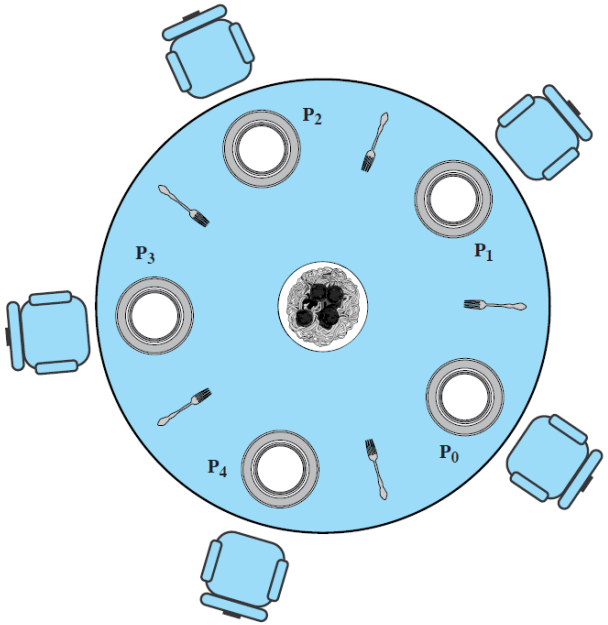
Signal(s_Q)

Signal(s_V)



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.

Rozwiązanie I – nieprawidłowe – możliwość zakleszczenia

```
#define NO_OF_PHIL 5
#define NO_OF_FORKS 5
#define MAX_DEL 4

void *philosopher(void *arg);
pthread_t philosopher_threads_table[NO_OF_PHIL];
sem_t _fork[NO_OF_FORKS];

void think(int i)
{
   int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d THINKS...\n",i);
   sleep(think_time);
}

void eat(int i)
{    int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d EATS...\n",i);
   sleep(think_time);
}

void hungry(int i)
{    printf("PHILOSOPHER %d HUNGRY...\n",i);
}
```

```
void *philosopher(void *arg)
{
   int phil_no;
   phil_no=( int)((int *) arg);

   printf("Philosopher %d started\n",phil_no);
   sleep(0);
   while(1)
   {
      think(phil_no);
      hungry(phil_no);
      sem_wait(&_fork[phil_no]);
         printf("Philosopher %d has one\n",phil_no);
      sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
         printf("Philosopher %d has two\n",phil_no);
      eat(phil_no);
      sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);
      sem_post(&_fork[phil_no]);
   }
}


// Dalej: powołanie czytelników i pisarzy + inicjalizacja
// semaforów + inicjalizacja losowych odcinków czasu
```

Uwagi

- Każdy z filozofów sięga po widelec z lewej strony talerza, a następnie po widelec z prawej strony.
- Kidy dany filozof skończy posiłek, odkłada oba widelce na stół.
- Dodatkowe funkcje tkink, eat i hungry służą do raportowania stanu danego filozofa oraz do symulowania czasu jedzenia i myślenia
- Takie rozwiązanie prowadzi do **impasu**: Stanie się to wtedy, gdy wszyscy filozofowie poczują jednocześnie głód i usiądą wspólnie przy stole, chwycą widelec z lewej strony, po czym sięgną po widelec z prawej strony.

Rozwiązanie II – bez możliwości zakleszczenia, ale kod filozofów się różni

```
#define NO_OF_PHIL 5
#define NO_OF_FORKS 5
#define MAX_DEL 4

void *philosopher(void *arg);
pthread_t philosopher_threads_table[NO_OF_PHIL];
sem_t _fork[NO_OF_FORKS];

void think(int i)
{    int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d THINKS...\n",i);
   sleep(think_time);}

void eat(int i)
{    int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d EATS...\n",i);
   sleep(think_time);}

void hungry(int i)
{    printf("PHILOSOPHER %d HUNGRY...\n",i);}
```

```
void *philosopher(void *arg){
   int phil_no;
   phil_no=( int)((int *) arg);
   printf("Philosopher %d started\n",phil_no);
   sleep(0);
   while(1)
   {if(phil_no%2) // parzyści       {
         think(phil_no);
         hungry(phil_no);
         sem_wait(&_fork[phil_no]);
            printf("Philosopher %d has one\n",phil_no);
         sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
            printf("Philosopher %d has two\n",phil_no);
         eat(phil_no);
         sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);
         sem_post(&_fork[phil_no]);       }
      else //nieparzyści       {
         think(phil_no);
         hungry(phil_no);
         sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
            printf("Philosopher %d has one\n",phil_no);
         sem_wait(&_fork[phil_no]);
            printf("Philosopher %d has two\n",phil_no);
         eat(phil_no);
         sem_post(&_fork[phil_no]);
         sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);}}}…
```

Uwagi

- Filozofów podzielona na „parzystych” i „nieparzystych”
- Parzyści filozofowie zachowują się jak w poprzednim przykładzie, zaś nieparzyści sięgają najpierw po prawy widelec
- W ten sposób uniknięto impasu, ale zachowanie wszystkich filozofów nie jest takie same.
- Innym rozwiązaniem jest wprowadzenie „kelnera”, który nie pozwoli, aby równocześnie przy stole siedziało 5 filozofów lub zastosowanie zmiennych warunkowych decydujących o dostępie do widelców.