Problemy synchronizacji

Systemy Operacyjne 2

Piotr Zierhoffer

5 grudnia 2011

Po co synchronizować?

Co się może stać, jeżeli współbieżne wykonanie procesów współdzielących dane nie jest synchronizowane?

- częściowe odczyty i zapisy
- wyścig
- zakleszczenia przy dostępie do zasobów blokowanych
- jednoczesny dostęp do **sekcji krytycznej**

Problemy synchronizacji

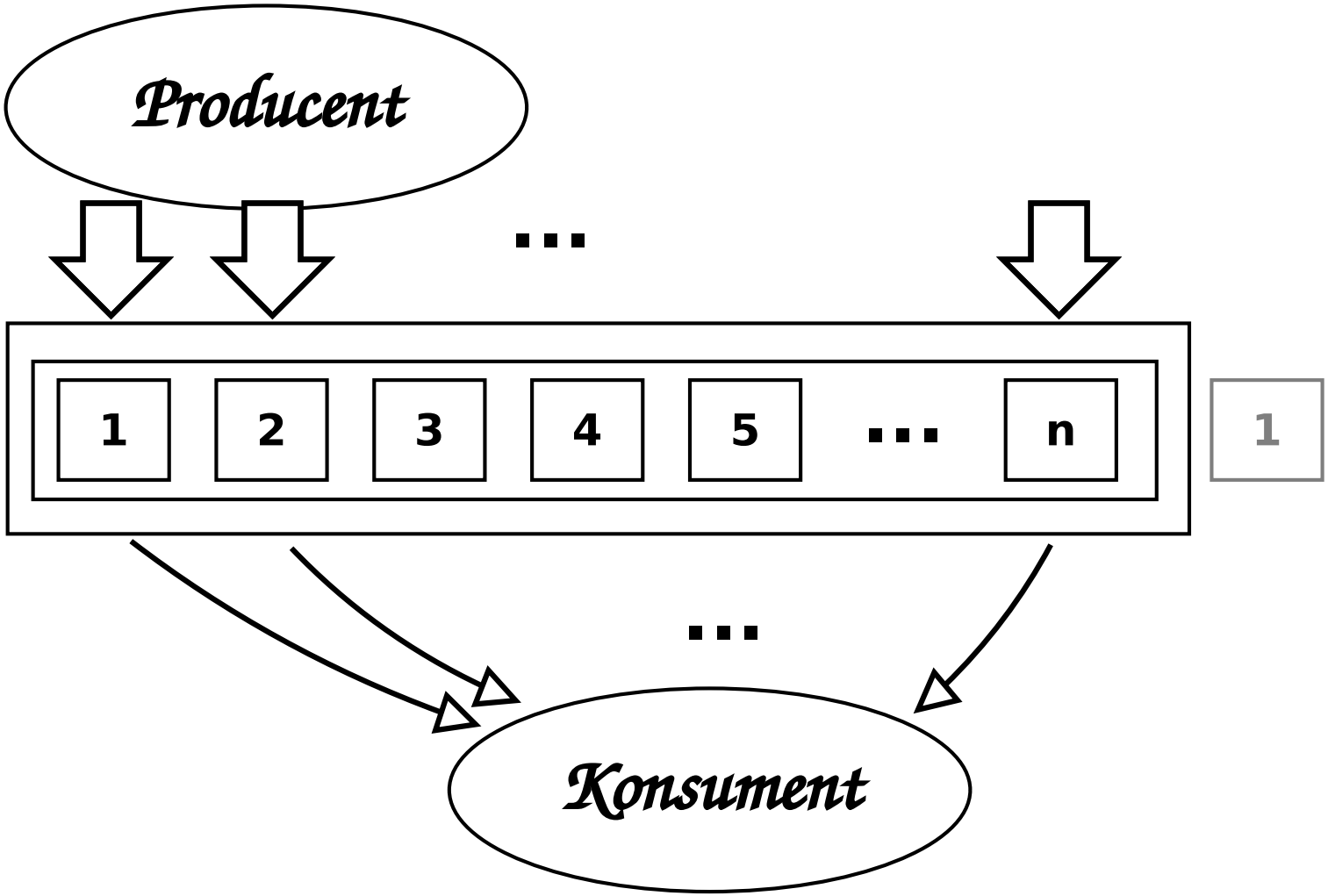
Klasyczne problemy synchronizacji:

- problem producenta i konsumenta
  - najpierw trzeba wyprodukować, żeby skonsumować
- problem czytelników i pisarzy
  - pisarz pracuje samotnie, czytelnicy czytają wspólnie
- problem jedzących filozofów
  - za mało widelców, żeby wszyscy jedli jednocześnie

Producent – konsument

- Ograniczony bufor cykliczny o rozmiarze `n`.
- Proces producenta wypełnia bufor
  - nie może przekroczyć rozmiaru bufora.
- Proces konsumenta pobiera elementy z bufora
  - nie może pobierać, jeżeli brak elementów.
- Podstawowy przypadek: jeden producent, jeden konsument

Producent – konsument



Producent – konsument — zmienne

Przyjmijmy następujące zmienne współdzielone. `COUNT` oznaczać będzie wielkość bufora `buffer`. Wykorzystamy dwa semafory, `empty` i full, oraz dwie zmienne wskazujące na komórki bufora dla producenta i konsumenta, odpowiednio: `write` i read.

```
COUNT : const Integer
buffer : Produkt[COUNT]
empty : Semaphor := 0
full : Semaphor := COUNT
read, write: Integer := 0
```

Producent – konsument, proces producenta

```
while ... do
    elem := Produce()
    P(empty, 1)
    buffer[write] := elem
    write := (write + 1) mod COUNT
    V(full, 1)
end
```

Producent tworzy element, następnie opuszcza semafor symbolizujący ilość wolnych miejsc w buforze.
Wypełnia odpowiednią komórkę bufora, przesuwa wskaźnik na następną komórkę oraz podnosi semafor informując konsumenta, że został utworzony nowy produkt.

Producent – konsument, proces konsumenta

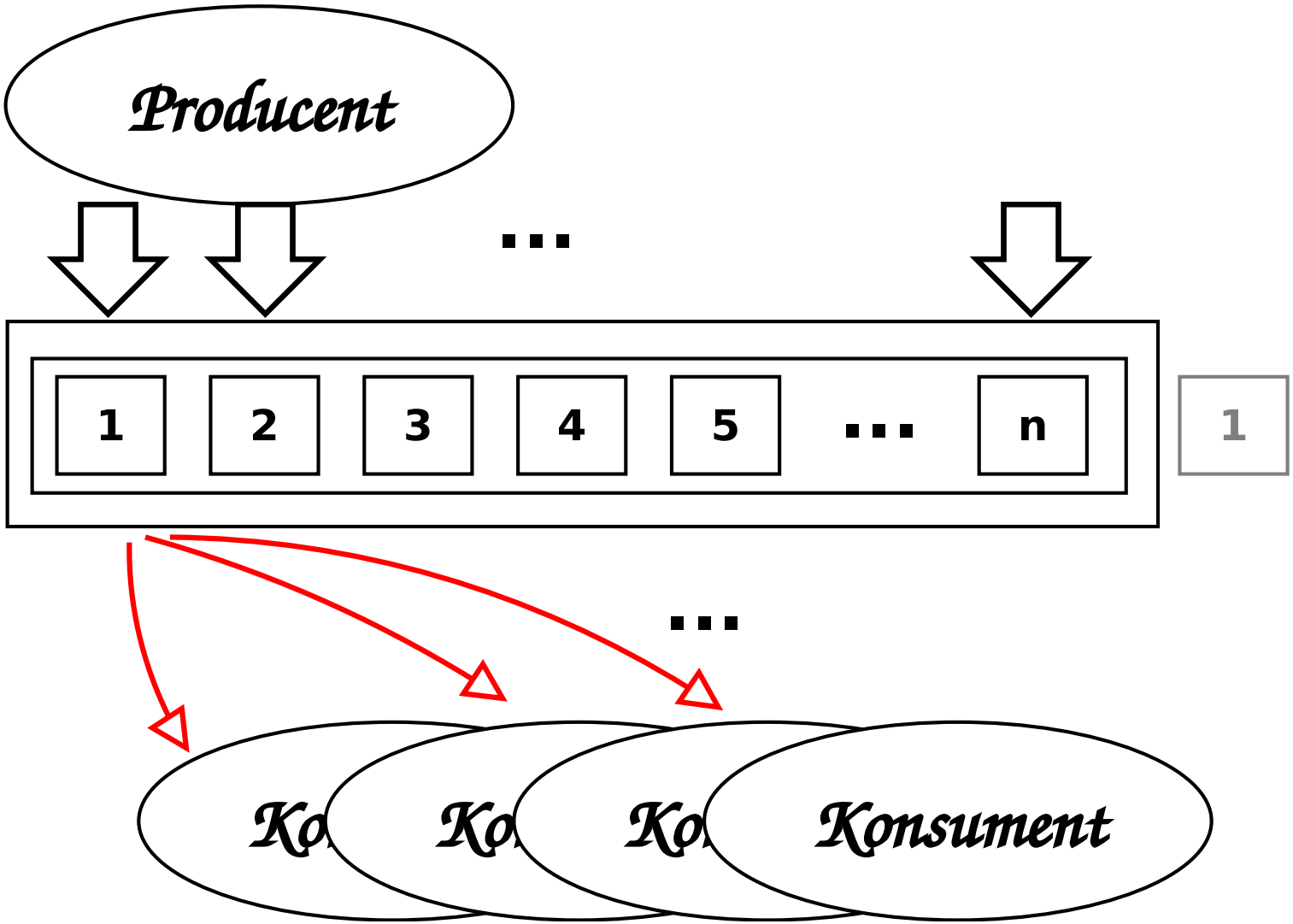
```
while ... do
   P(full, 1)
   elem := buffer[read]
   read := (read + 1) mod COUNT
   V(empty, 1)
   Consume(elem)
end
```

Konsument stara się opuścić semafor symbolizujący ilość utworzonych produktów. Przejście dalej oznacza, że co najmniej jeden produkt jest dostępny — konsument pobiera go w kolejnej operacji, po czym przesuwa wskaźnik na następną pozycję.
W kolejnym kroku podnosi semafor informując producenta, że zwolniło się pole w buforze. Na końcu wykonuje operacje na pobranym elemencie.

Producent – konsument

Czy algorytm ten skaluje się na wielu producentów i konsumentów?

Producent – konsument, wielu konsumentów



Producent – konsument, wiele procesów

Problem wyścigu:

- przedstawione rozwiązanie zabezpiecza działania producentów względem konsumentów i na odwrót
- brak synchronizacji między konsumentami/producentami
- jednocześnie w sekcji krytycznej znajdować się może wiele procesów
- możliwość odczytania/nadpisania jednej komórki wielokrotnie

Rozwiązanie:

- dodatkowa synchronizacja

```
S: BinarySemaphore := 1
```

Producent – konsument, wiele procesów

```
while ... do
   elem := Produce()
   P(empty, 1)
      P(S)
      buffer[write] := elem
      write := (write + 1) mod COUNT
      V(S)
   V(full, 1)
end
```

```
while ... do
   P(full, 1)
      P(S)
      elem := buffer[read]
      read := (read + 1) mod COUNT
      V(S)
   V(empty, 1)
   Consume(elem)
end
```

Producent – konsument

Podsumowanie:

- synchronizacja musi uwzględniać wszystkie aspekty współdzielenia
- warto najpierw rozwiązać problem prostszy a potem go uogólnić
- kolejność zakładania semaforów jest **krytycznie** ważna
- problem producent – konsument dla pojedynczych instancji procesów można rozwiązać bez wykorzystania semaforów
  - $\rightrightarrows$ en.wikipedia.org/...

Problem czytelników i pisarzy

- n czytelników, m pisarzy
- w bibliotece pisarze piszą (!) książki, czytelnicy je czytają
- pisarz wymaga całej biblioteki dla siebie
- czytelnicy mogą czytać wspólnie

Problem czytelników i pisarzy, zmienne

Niech jako `num_read` będzie zdefiniowana ilość aktualnie czytających czytelników. Dostęp do biblioteki synchronizowany będzie za pomocą semafora `sem_bib`, dodatkowo zdefiniujemy semafor `sem_aux` w celu synchronizowania dostępu do zmiennej `num_read`. Oba semafory są binarne.

```
num_read: Integer := 0
sem_bib: Semaphore := 1
sem_aux: Semaphore := 1
```

Problem czytelników i pisarzy, proces pisarza

```
while ... do
   P(sem_bib)
   WRITE()
   V(sem_bib)
end
```

Pisarz ma dostęp do biblioteki wtedy, kiedy nikogo w niej nie ma, tzn. semafor sem_bib jest podniesiony.

Problem czytelników i pisarzy, proces czytelnika

```
while ... do
   P(sem_aux)
   num_read := num_read + 1
   if num_read = 1 then  P(sem_bib)
   V(sem_aux)
   READ()
   P(sem_aux)
   num_read := num_read - 1
   if num_read = 0 then  V(sem_bib)
   V(sem_aux)
end
```

Przed wejściem do sekcji krytycznej pierwszy czytelnik musi się upewnić, że nie ma w niej żadnego pisarza. Opuszcza więc semafor sem_bib i oczekuje na pustą bibliotekę. Dzięki semaforowi sem_aux nie wyprzedzi go żaden inny czytelnik.
Jeżeli z sekcji wychodzi ostatni czytelnik, może on podnieść semafor sem_bib. Dopóki w bibliotece jest chociaż jeden czytelnik, inni czytelnicy nie muszą operować na tym semaforze.

Problem czytelników i pisarzy, problemy

Czy algorytm daje równy dostęp do zasobów wszystkim uczestnikom komunikacji?

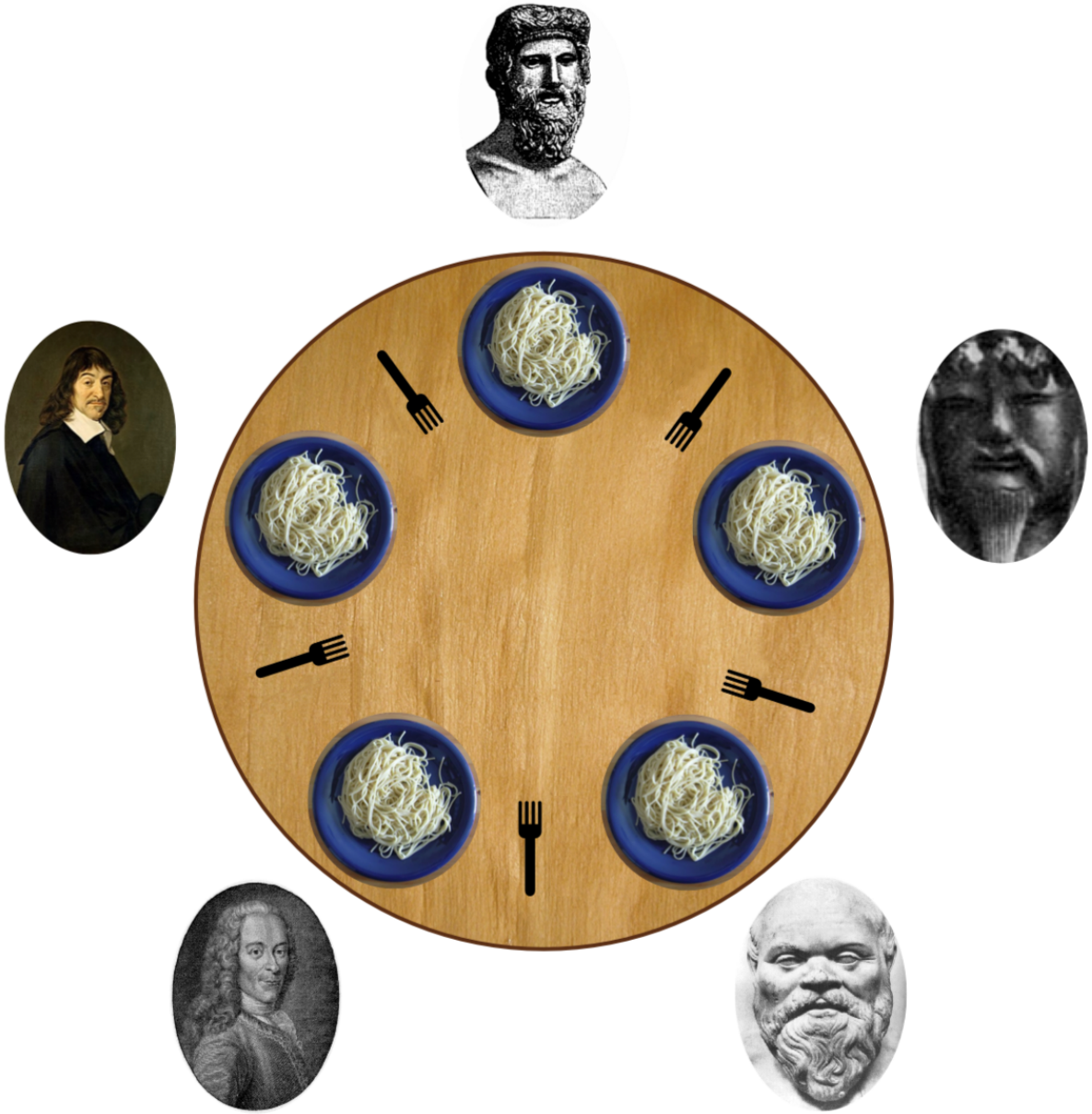
- przedstawiony algorytm faworyzuje czytelników!
- może dojść do **zagłodzenia** pisarzy:
  - istnieje możliwość, że pisarz **nigdy** nie uzyska dostępu do sekcji krytycznej
- istnieją alternatywne wersje algorytmu ($\rightrightarrows$ en.wikipedia.org):
  - faworyzujące pisarzy — *żaden pisarz nie będzie oczekiwał na dostęp do zasobów dłużej, niż to konieczne*
  - sprawiedliwe — oparte na kolejce FIFO, *żaden czytelnik ani pisarz stojący w kolejce nie będzie wyprzedzony przez inny proces*
    - semafory POSIX nie wystarczą do implementacji
    - mniejsze zrównoleglenie czytelników

Problem ucztujących filozofów

Pięciu filozofów siedzi przy okrągłym stole:

- filozof myśli, aż zgłodnieje
- każdy filozof ma przed sobą talerz makaronu
- filozof do jedzenia potrzebuje 2 widelców
- filozof nie odda widelca, dopóki się nie naje
- pomiędzy sąsiadami jest jeden widelec
- filozofowie ze sobą nie rozmawiają
- jak zabezpieczyć filozofów przed zagłodzeniem i zakleszczeniem?

Problem ucztujących filozofów



Problem ucztujących filozofów

Przyjmijmy tablicę semaforów binarnych `fork[N]`, gdzie `N` to liczba dostępnych widelców (a zarazem filozofów). Dla każdego z filozofów `i` oznaczać będzie jego numer.

```
N : const Integer
fork : BinarySemaphore[N] := 1
i : local Integer

while ... do
    Think()
    P(fork[i])
    P(fork[(i + 1) mod N]
    Eat()
    V(fork[i])
    V(fork[(i + 1) mod N]
end
```

Problem ucztujących filozofów, cd.

Czy takie rozwiązanie jest bezpieczne?

- spełniony jest warunek bezpieczeństwa
  - jedzenie po podniesieniu obu widelców
- wzajemne wykluczanie przy dostępie do widelców
- **możliwość zakleszczenia!**
  - gdy wszyscy biesiadnicy chwycą lewy widelec
- możliwe rozwiązania?
  - 4 talerze / kelner
  - hierarchia zasobów
  - $\rightrightarrows$ en.wikipedia.org

Problem ucztujących filozofów, 4 talerze

Dodajmy semafor uogólniony sem_plate symbolizujący ilość talerzy dostępnych na stole. Talerzy jest o 1 mniej niż filozofów.

```
sem_plate : Semaphore := COUNT - 1

while ... do
    Think()
    P(sem_plate)
    P(fork[i])
    P(fork[(i + 1) mod N]
    Eat()
    V(fork[i])
    V(fork[(i + 1) mod N]
    V(sem_plate)
end
```

Problem ucztujących filozofów, hierarchia

Rozwiązanie zaproponowane przez Dijkstrę:

- niech każdy widelec ma numer od 1 do N
- filozof najpierw musi podnieść widelec o niższym numerze, potem wyższym
- odkładanie widelców następuje w odwrotnej kolejności
- filozofowie od 1 do 4 biorą najpierw lewy widelec, potem prawy
- filozof 5 najpierw bierze prawy widelec, potem lewy
- rozwiązanie w rzeczywistych systemach słabo skalowalne
  - jeżeli okazuje się, że potrzebuję zasobu o niskim identyfikatorze, muszę zwrócić wszystkie zasoby o identyfikatorach wyższych