Паралелно Програмиране

Проблеми при паралелните алгоритми.
Задача за „Вечерящите философи“.
Мъртва хватка, жива хватка, трудна скалируемост,
глад за ресурси, съперничество и др.
Producer-Consumer.

доц. д-р Александър Пенев

Задача за Вечерящите Философи

Вечерящите философи (Concurrency, Паралелност)

- Пет мълчаливи философи седят на кръгла маса с чинии със спагети;
- Вилици се поставят между всяка двойка съседни философи;
- Всеки философ трябва последователно да мисли и да се храни;

Вечерящите философи (Concurrency, Паралелност)

- Философът обаче може да яде спагети само когато имат и лява, и дясна вилица;
- Всяка вилица може да се държи само от един философ и така философът може да използва вилицата само ако не се използва от друг философ;

Вечерящите философи (Concurrency, Паралелност)

- След като отделен философ приключи с храненето, те трябва да сложат и двете вилици, така че вилиците да станат достъпни за другите;
- Може да вземе вилицата само отдясно или тази отляво, когато станат достъпни и те не могат да започнат да ядат, преди да получат и двете вилици;

Проблем

- Проблемът е как да се проектира дисциплина на поведение (паралелен алгоритъм), така че никой философ да не гладува; т.е. всеки може завинаги да продължи да редува хранене и мислене (никой философ не може да знае кога другите могат да искат да ядат или да мислят).

Вечерящите философи

- Взаимното изключване (mutual exclusion) е основната идея на проблема;
- Провалите, които тези философи могат да изпитат, са аналогични на трудностите, които възникват при реалното компютърно програмиране, когато много програми се нуждаят от изключителен достъп до споделени ресурси;

Вечерящите философи

- Първоначалните проблеми на Dijkstra са свързани с външни устройства като касетни устройства. Въпреки това, трудностите, обяснени от проблема с философите на трапезарията, възникват много по-често, когато множество процеси имат достъп до набори от данни, които се актуализират;
- Системи като ядрото на ОС, използват хиляди заключвания и синхринизации, които изскват стриктно следване на протоколите за да се избегнат проблеми като мъртви хватки и др.

Възможни Решения

- Въвеждане на йерархия на ресурсите – между ресурсите се въвежда частична наредба (номерираме ги), след което когато даден философ взема ресурс (от вилиците номерирани от 1 до 5), то той трябва да го прави в строго нарастващ ред по реда на ресурсите. Макар и решение то не винаги води до ефективни системи;
- Арбитриране – за да се гарантира, че философите ще вземат винаги две вилици, се въвежда арбитър (сервитьор), който разрешава ползването на ресурсите само на един философ в даден момент. Връщането става във всеки момент;
- Chandy-Misra;

Алгоритъм на Декер

Алгоритъм на Декер (Dekker)

- Алгоритъмът на Декер е първото известно решение на проблема с взаимното изключване в паралелното програмиране;
- Той позволява на две нишки да споделят ресурс за еднократна употреба без конфликт, като се използва само споделена памет за комуникация;
- При реализация в модерните компютри трябва да се използва бариера (memory barrier), да се внимава за оптимизациите;
- При езиците предлагащи “volatile” е модификатора да бъде използван пред променливите;

Алгоритъм на Декер (Dekker)

```
vars: wants_to_enter : array of 2 booleans ← {false, false}
      turn : integer ← 0   // or 1
```

```
P0: wants_to_enter[0] ← true
while wants_to_enter[1] {
  if turn ≠ 0 {
    wants_to_enter[0] ← false
    while turn ≠ 0 { // wait }
    wants_to_enter[0] ← true
  }
}
// critical section ...
turn ← 1
wants_to_enter[0] ← false
```

```
P1: wants_to_enter[1] ← true
while wants_to_enter[0] {
  if turn ≠ 1 {
    wants_to_enter[1] ← false
    while turn ≠ 1 { // wait }
    wants_to_enter[1] ← true
  }
}
// critical section ...
turn ← 0
wants_to_enter[1] ← false
```

Задача за Пушачите

Задача за Пушачите

The smoker

(The X-Files)

Thin Man

(Charlie's Angels)

Catherine Tramell

(Basic Instinct)

Задача за Пушачите

- Да приемем, че една цигара изисква три съставки за приготвяне и пушене: тютюн, хартия и кибрит;
- Около масата има трима пушачи, всеки от които има безкрайна доставка на една от трите съставки – един пушач има безкраен запас тютюн, друг има хартия, а третият има кибрит;
- Има и агент за непушачи, който дава възможност на пушачите да си правят цигари, като произволно (недетерминирано) избира два от консумативите, които да поставят на масата;
- Пушачът, който има третата съставка, трябва да вземе двете съставки от масата, като ги използва (заедно със собствената си съставка), за да си направи цигара, която пуши известно време.

Задача за Пушачите

Реализация (псевдокод)

```
def tobacco_smoker():
    repeat:
        paper.wait()
        matches.wait()
        smoke()
        tobacco_smoker_done.signal()
```

Другите две реализации за пушачите са аналогични.

Проблем и Решение

- Когато агентът постави примерно тютюн и хартия на маста, може притежаващия тютюн пушач да вземе хартията, което води до мъртва хватка;
- Решението е да се използват три “семафора” за да описват поставените на маста съставки, както и три “семафора” за всеки пушач, който информира агента, че съответния пушат е приключил с пушенето;

Задача за Спящия Бръснар

Задача за Спящия бръснар

Задача за Спящия бръснар

- Нека да имаме бръснарница с един бръснар и един стол за подстригване;
- Нека имаме чакалня с N на брой стола;
- Когато няма клиенти бръснарят сяда на стола и спи;
- Когато дойде клиент той вижда дали има друг клиент на стола:
  - Ако да или сяда в чакалнята или си тръгва (ако няма места);
  - Ако не – влиза, събъжда бръснаря и сяда на стола;
- Когато бръснарят свърши с подстригването, той изпраща клиента и гледа дали има клиенти в чакалнята:
  - Ако няма – сяда на стола и заспива;
  - Ако има – кани произволен от тях;

Особености и Проблеми при Паралелни Програми

Опасности и проблеми

- Мъртва хватка (Deadlock)
- Жива хватка (Livelock)
- Конвоиране (Convoying)
- Съперничество (Contention)
- Глад за ресурси (Starvation)
- Допълнително време (Overhead)
- Забавяне (Parallel slowdown)

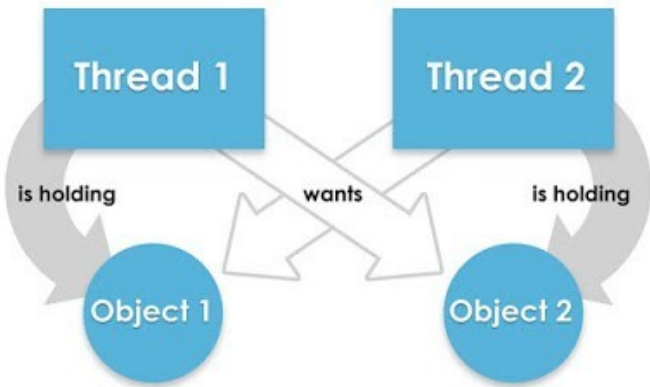
- Безпроблемни “Embarrassingly parallel”
- Детерминирани и недетерминирани алгоритми;

Мъртва хватка (Deadlock)

Мъртва хватка (Deadlock)

Мъртва хватка е проблем в паралелното програмиране, при който два или повече процеса взаимно се изчакват за достъп до общи ресурси, като всички процеси попадат в изчакващо състояние

Ситуацията прилича на тази, в която две коли се срещат на тесен път и нито едната може да мине, нито другата.



Решения при Мъртва хватка (Deadlock)

- Действия при откриване на мъртва хватка – спиране на процесите, спиране един по един до преодоляване на проблема;
- Алгоритъм на “Банкера”;
- Resource Preemption;
- Оптимистичен и писимистичен сценарии;

Жива хватка (Livelock)

Жива хватка (Livelock)

Жива хватка е проблем в паралелното програмиране, при който два или повече процеса взаимно се изчакват за достъп до общи ресурси, като за разлика от мъртвата хватка те не са в изчакващо състояние, а активно се опитват да получат ресурсите без да извършват никаква друга полезна работа.

Ситуацията прилича на тази, в която двама души се срещат лице в лице на коридора и всеки отстъпва път на другия, но после и двамата тръгват напред и пак се сблъскват и т.н.

Решения при Жива хватка (Livelock)

- По аналогия с мъртвата хватка за избягване на проблема;
- Живата хватка е дори по-опасна и трудна за откриване, защото процесите не “спят”, а извършват някаква “работа” – активно чакане, което отнема допълнителни ресурси и не извършва полезна работа;

Конвоиране (Convoying)

Конвоиране (Convoying)

Конвоиране е проблем в паралелното програмиране, при който за разлика от мъртвата хватка процесите не се блокират и работят, но непрекъснато се изчакват един друг, като въпреки че вършат и полезна работа, то тя е много малко, което води до драстично падане на производителността. Например може да се получи, че процесите чакат един от тях, а той изчаква нов квант време за да продължи работата си.

Решения при Конвоиране (Convoying)

- Анализ на работата на процесите;
- По-прецизна синхронизация;

Съперничество (Contention)

Съперничество (Contention)

Съперничество е проблем в паралелното програмиране, при който възниква “спор” за достъп до споделен ресурс, като памет с произволен достъп, дисково съхранение, кеш памет, вътрешни шини или външни мрежови устройства и др. Ресурс, за който има непрекъснати спорове, може да бъде определен като прекалено желан (необходим).

Разрешаване то на тези проблеми е една от основните функции на ОС.

Глад за ресурси (Starvation)

Глад за ресурси (Starvation)

Глад за ресурси е проблем в паралелното програмиране, при който имаме проблеми с общността на заключването:
Ред вместо таблица, клетка вместо ред, ...

Решения при Глад за ресурси (Starvation)

- Прецизиране на заключването;
- Анализ на алгоритъма и промени в него;

Допълнително време (Overhead)

Допълнително време (Overhead)

Допълнително време е проблем в паралелното програмиране, при който понякога времето за изпълнение на организационните дейности по паралелизация, критични секции и други е прекалено много в сравнение с реално извършваните действия на алгоритъма.

Такова време винаги съществува, но проблема е когато то е прекалено голям процент от общото време на алгоритъма.

Решения при Допълнително време (Overhead)

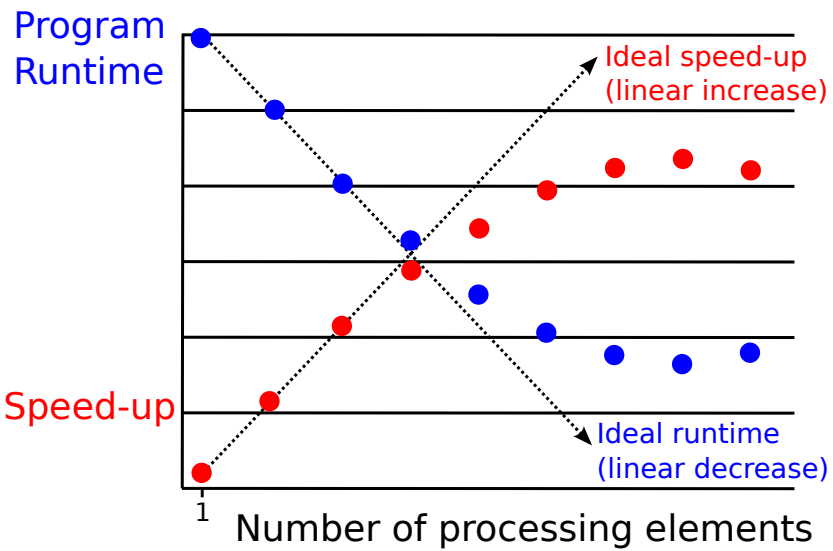
- Намаляне на броя организационни дейности;
- Увеличаване на обемите от данни обработвани от един процес;
- И др.

Забавяне (Parallel slowdown)

Забавяне (Parallel slowdown)

Паралелното забавяне е феномен в паралелните изчисления, при което паралелизирането на алгоритъм над определена точка кара програмата да работи по-бавно (отнема повече време, за да се изпълни до завършване).

Това обикновено се дължи на тесни места или проблеми в комуникацията.



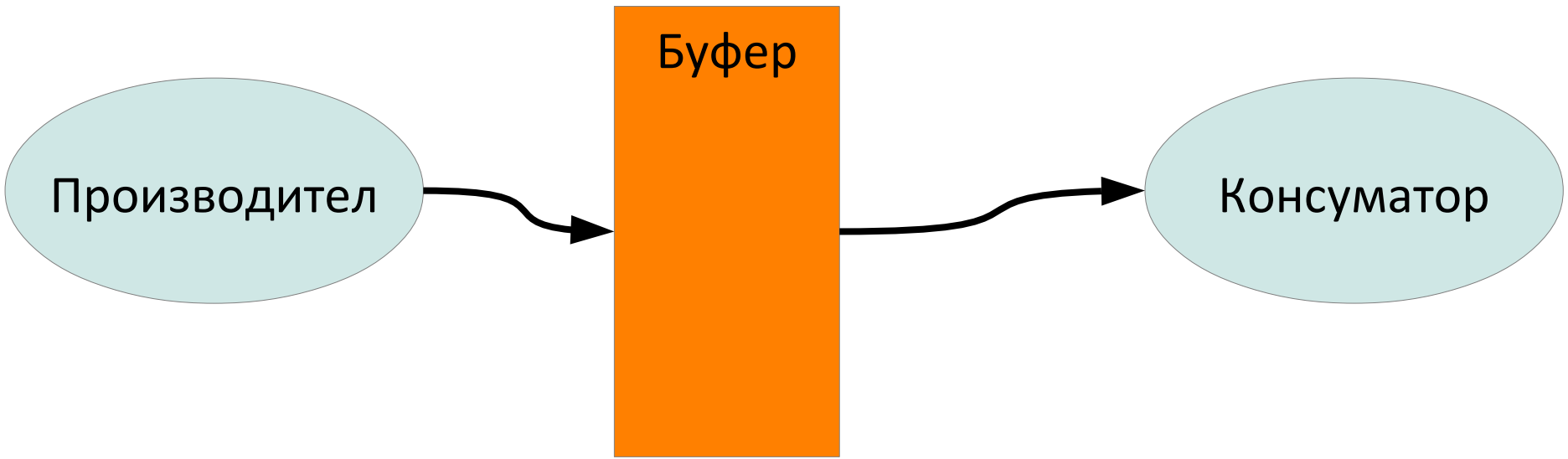
Решения при Забавяне (Parallel slowdown)

- Откриване на тесните места;
- Анализ на комуникацията между процесите;
- Отстраняване на проблема;

- Някои алгоритми като класа “неприлично паралелни” нямат този проблем, защото нямат изобщо или почти нямат комуникация;

Producer-Consumer

Производител-Консуматор



Наивно”решение” – може да доведе до мъртва хватка

```
int itemCount = 0;
procedure producer() {
  while (true) {
    item = produceItem();
    if (itemCount == BUFFER_SIZE) { sleep(); }
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1) { wakeup(consumer); }
  }
}
procedure consumer() {
  while (true) {
    if (itemCount == 0) { sleep(); }
    item = removeItemFromBuffer();
    itemCount = itemCount - 1;
    if (itemCount == BUFFER_SIZE - 1) { wakeup(producer); }
    consumeItem(item);
  }
}
```

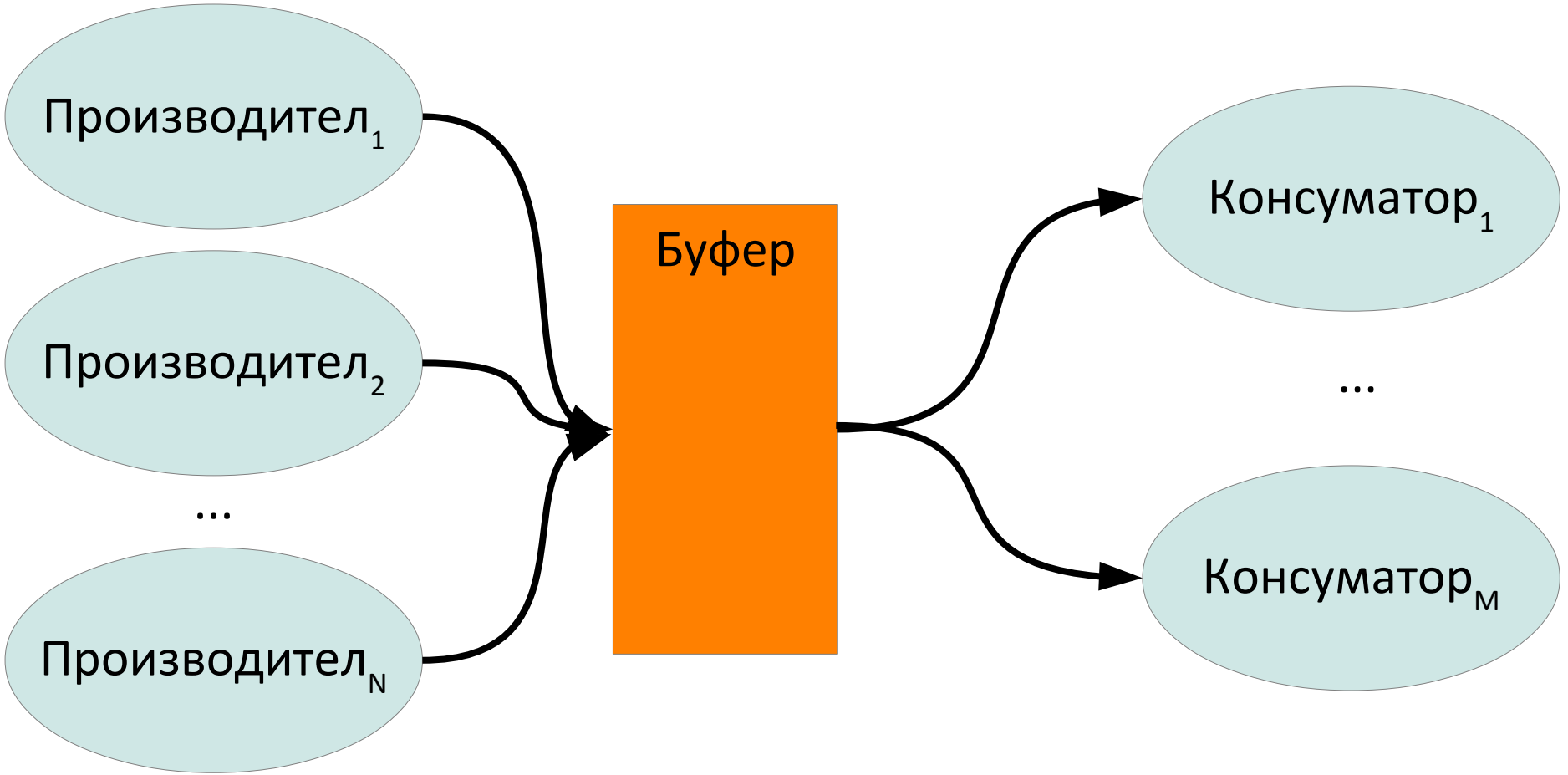
Решение със Семафори

```
semaphore fillCount = 0; // items produced
semaphore emptyCount = BUFFER_SIZE; // remaining space
procedure producer() {
  while (true) {
    item = produceItem();
    down(emptyCount);
    putItemIntoBuffer(item); // Problem when producers are more than one
                             // Solution – use mutex m: down(m); put; up(m);
    up(fillCount);
  }
}
procedure consumer() {
  while (true) {
    down(fillCount);
    item = removeItemFromBuffer(); // Producers > 1: down(m); rem; up(m);
    up(emptyCount);
    consumeItem(item);
  }
}
```

Решение без семафори

```
volatile unsigned int produceCount = 0, consumeCount = 0;
TokenType sharedBuffer[BUFFER_SIZE];
void producer(void) {
  while (1) {
    while (produceCount - consumeCount == BUFFER_SIZE) {
      schedulerYield(); /* sharedBuffer is full */
    }
    /* Write to sharedBuffer _before_ incrementing produceCount */
    sharedBuffer[produceCount % BUFFER_SIZE] = produceToken();
    /* Memory barrier required here to ensure update of the sharedBuffer is
    visible to other threads before the update of produceCount */
    ++produceCount;
  }
}
void consumer(void) {
  while (1) {
    while (produceCount - consumeCount == 0) {
      schedulerYield(); /* sharedBuffer is empty */
    }
    consumeToken(&sharedBuffer[consumeCount % BUFFER_SIZE]);
    ++consumeCount;
  }
}
```

Производител-Консуматор



Други Решения на Проблемите на Паралелните Алгоритми

Други Решения

- Използване на свободни от заключване структури;
- Използване на високо паралелни алгоритми;
- Транзакционална памет (Transactional memory);
- Read-Copy-Update (RCU);
- …;