**03.12.2020 ИС2К Операционные системы**

**«Основы операционных систем» К.А. Коньков** <https://yadi.sk/i/SojomTcdoQ2v_g>

**Тема:** Лекция 5: **Алгоритмы синхронизации**

(Учебник «Основы операционных систем» К.А.Коньков стр. 49-58)

**Ответить на вопросы: (кратко 3-4 предложения максимум)**

1. **Неделимые операции (определение)**
2. **Interleaving, race condition и взаимоисключения (определение)**
3. **Критическая секция (определение)**
4. **Программные алгоритмы организации взаимодействия процессов(перечислить)**

**Ответы присылать** [kuzn117@yandex.ru](mailto:kuzn117@yandex.ru) **можно в ВК -**  **id480169637**

**Пишите тему письма и файл именуйте: Фамилия-ОС-0312**

**Для тех у кого проблемы с Яндекс диском сегодняшнюю тему**

**(**Лекция 5:

Алгоритмы синхронизации)

**выкладываю ниже**

**5.Лекция: Алгоритмы синхронизации**

Для корректного взаимодействия процессов недостаточно одних организационных усилий операционной  
системы. Необходимы определенные внутренние изменения в поведении процессов. В настоящей лекции  
рассматриваются вопросы, связанные с такими изменениями, приводятся программные алгоритмы кор-  
ректной организации взаимодействия процессов.

В предыдущей лекции мы говорили о внешних проблемах кооперации, связанных с организацией взаи-  
модействия процессов со стороны операционной системы. Предположим, что надежная связь процессов  
организована, и они умеют обмениваться информацией. Нужно ли нам предпринимать еще какие-либо  
действия для организации правильного решения задачи взаимодействующими процессами? Нужно ли  
изменять их внутреннее поведение? Разъяснению этих вопросов и посвящена данная лекция.

**Interleaving, race condition и взаимоисключения**

Давайте временно отвлечемся от операционных систем, процессов и нитей исполнения и поговорим о  
некоторых "активностях". Под активностями мы будем понимать последовательное выполнение ряда  
действий, направленных на достижение определенной цели. Активности могут иметь место в программ-  
ном и техническом обеспечении, в обычной деятельности людей и животных. Мы будем разбивать ак-  
тивности на некоторые неделимые, или атомарные, операции. Например, активность "приготовление бу-  
терброда" можно разбить на следующие атомарные операции:

1. Отрезать ломтик хлеба.

2. Отрезать ломтик колбасы.

3. Намазать ломтик хлеба маслом.

4. Положить ломтик колбасы на подготовленный ломтик хлеба.

Неделимые операции могут иметь внутренние невидимые действия (взять батон хлеба в левую руку,  
взять нож в правую руку, произвести отрезание). Мы же называем их неделимыми потому, что считаем  
выполняемыми за раз, без прерывания деятельности.

Пусть имеется две активности

P: a b c

Q: d e f

где a, b, c, d, e, f - атомарные операции. При последовательном выполнении активностей мы получаем  
такую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени?  
Активности могут расслоиться на неделимые операции с различным чередованием, то есть может про-  
изойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередо-

|  |  |
| --- | --- |
| вания: |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | |  |  |  |
| а | b | c | d | e | f |
| a | b | d | c | e | f |
| a | b | d | e | c | f |
| a | b | d | e | f | c |
| a | d | b | c | e | f |
| d | e | f | a | b | c |

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохра-  
нением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух актив-

**Основы операционных систем**

**50**

ностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения  
может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеет-  
ся две активности P и Q, состоящие из двух атомарных операций каждая:

P: x=2 Q: x=3

y=x-1 y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и у являются для  
активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, у): (3, 4), (2,  
1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) детерминирован, ес-  
ли всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает  
одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример неде-  
терминированного набора программ. Понятно, что детерминированный набор активностей можно безбо-  
язненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение  
нежелательно.

Можно ли до получения результатов определить, является ли набор активностей детерминированным  
или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к про-  
граммам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы  
входных и выходных переменных - это наборы переменных, которые атомарная операция считывает и  
записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов  
входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных про-  
граммы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых  
действий. Например, для программы

P: x=u+v

y=x\*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в  
W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

• пересечение W(P) и W(Q) пусто,

• пересечение W(P) с R(Q) пусто,

• пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может  
быть, и нет.

Случай двух активностей естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически не-  
взаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали ак-  
тивности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ог-  
раничить число возможных чередований атомарных операций, исключив некоторые чередования с по-  
мощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ  
программ к некоторым данным.

Про недетерминированный набор программ (и активностей вообще) говорят, что он имеет race condition  
(состояние гонки , состояние состязания). В приведенном выше примере процессы состязаются за вычис-  
ление значений переменных x и у.

**Основы операционных систем**

**51**

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда  
нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право  
доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех  
других процессов возможность одновременного общения с этими ресурсами, если это может привести к  
недетерминированному поведению набора процессов. Такой прием называется взаимоисключением  
(mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных  
результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения  
программ.

**Критическая секция**

Важным понятием при изучении способов синхронизации процессов является понятие критической сек-  
ции (critical section) программы. Критическая секция - это часть программы, исполнение которой может  
привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект  
гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый мо-  
мент времени только один процесс мог находиться в своей критической секции, связанной с этим ресур-  
сом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций  
программ. Реализация взаимоисключения для критических секций программ с практической точки зре-  
ния означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая  
секция начинает выполняться как атомарная операция. Давайте рассмотрим следующий пример, в кото-  
ром псевдопараллельные взаимодействующие процессы представлены действиями различных студентов  
(таблица 5.1):

Здесь критический участок для каждого процесса - от операции "Обнаруживает, что хлеба нет" до опе-  
рации "Возвращается в комнату" включительно. В результате отсутствия взаимоисключения мы из си-  
туации "Нет хлеба" попадаем в ситуацию "Слишком много хлеба". Если бы этот критический участок  
выполнялся как атомарная операция - "Достает два батона хлеба", то проблема образования излишков  
была бы снята.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.1. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  | |
| 17-07 | Обнаруживает, что хлеба нет |  | |
| 17-09 | Уходит в магазин |  | |
| 17-11 |  | Приходит в комнату |  |
| 17-13 |  | Обнаруживает, что хлеба нет |  |
| 17-15 |  | Уходит в магазин |  |
| 17-17 |  |  | Приходит в комнату |
| 17-19 |  |  | Обнаруживает, что хлеба нет |
| 17-21 |  |  | Уходит в магазин |
| 17-23 | Приходит в магазин |  | |
| 17-25 | Покупает 2 батона на всех |  | |
| 17-27 | Уходит из магазина |  | |
| 17-29 |  | Приходит в магазин |  |
| 17-31 |  | Покупает 2 батона на всех |  |
| 17-33 |  | Уходит из магазина |  |
| 17-35 |  |  | Приходит в магазин |
| 17-37 |  |  | Покупает 2 батона на всех |
| 17-39 |  |  | Уходит из магазина |
| 17-41 | Возвращается в комнату |  | |
| 17-43 |  |  | |

**Основы операционных систем**

**52**

|  |  |  |
| --- | --- | --- |
| 17-45  17-47 | Возвращается в комнату |  |
| 17-49 |  | |
| 17-51 |  | |
| 17-53 |  | Возвращается в комнату |

Сделать процесс добывания хлеба атомарной операцией можно было бы следующим образом: перед на-  
чалом этого процесса закрыть дверь изнутри на засов и уходить добывать хлеб через окно, а по оконча-  
нии процесса вернуться в комнату через окно и отодвинуть засов. Тогда пока один студент добывает  
хлеб, все остальные находятся в состоянии ожидания под дверью (таблица 5.2).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 5.2. | | | |
| **Время** | **Студент 1** | **Студент 2** | **Студент 3** |
| 17-05 | Приходит в комнату |  |  |
| 17-07 | Достает два батона хлеба |  |  |
| 17-43 | | Приходит в комнату |  |
| 17-47 | |  | Приходит в комнату |

Итак, для решения задачи необходимо, чтобы в том случае, когда процесс находится в своем критиче-  
ском участке, другие процессы не могли войти в свои критические участки. Мы видим, что критический  
участок должен сопровождаться прологом (entry section) - "закрыть дверь изнутри на засов" - и эпилогом  
(exit section) - "отодвинуть засов", которые не имеют отношения к активности одиночного процесса. Во  
время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический  
участок, а во время выполнения эпилога - сообщить другим процессам, что он покинул критическую  
секцию.

В общем случае структура процесса, участвующего во взаимодействии, может быть представлена сле-  
дующим образом:

while (some condition) {  
entry section

critical section  
exit section

remainder section

}

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Оставшаяся часть этой лекции посвящена различным способам программной организации пролога и эпи-  
лога критического участка в случае, когда очередность доступа к критическому участку не имеет значе-  
ния.

**Программные алгоритмы организации взаимодействия процессов  
Требования, предъявляемые к алгоритмам**

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения  
race condition, но не является достаточной для правильной и эффективной параллельной работы коопера-  
тивных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программ-  
ного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут  
проходить их в произвольном порядке.

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей спе-  
циальных команд взаимоисключения. При этом предполагается, что основные инструкции языка

**Основы операционных систем**

**53**

программирования (такие примитивные инструкции, как load, store, test) являются атомарными  
операциями.

2. Не должно существовать никаких предположений об относительных скоростях выполняющихся  
процессов или числе процессоров, на которых они исполняются.

3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других про-  
цессов, которые исполняются в соответствующих критических секциях. Это условие получило на-  
звание условия взаимоисключения (mutual exclusion).

4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не  
могут препятствовать другим процессам входить в их собственные критические участки. Если нет  
процессов в критических секциях и имеются процессы, желающие войти в них, то только те про-  
цессы, которые не исполняются в remainder section, должны принимать решение о том, какой про-  
цесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно дол-  
го. Это условие получило название условия прогресса (progress).

5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой кри-  
тический участок. От того момента, когда процесс запросил разрешение на вход в критическую  
секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти че-  
рез свои критические участки лишь ограниченное число раз. Это условие получило название ус-  
ловия ограниченного ожидания (bound waiting).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа  
организации пролога и эпилога для критической секции.

**Запрет прерываний**

Наиболее простым решением поставленной задачи является следующая организация пролога и эпилога:

while (some condition) {

запретить все прерывания  
critical section  
разрешить все прерывания  
remainder section

}

Поскольку выход процесса из состояния **исполнение** без его завершения осуществляется по прерыва-  
нию, внутри критической секции никто не может вмешаться в его работу. Однако такое решение может  
иметь далеко идущие последствия, поскольку позволяет процессу пользователя разрешать и запрещать  
прерывания во всей вычислительной системе. Допустим, что пользователь случайно или по злому умыс-  
лу запретил прерывания в системе и зациклил или завершил свой процесс. Без перезагрузки системы в  
такой ситуации не обойтись.

Тем не менее запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим  
секциям внутри самой операционной системы, например при обновлении содержимого PCB.

**Переменная-замок**

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое  
предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением рав-  
ным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка  
равно 0, одновременно изменяя ее значение на 1 - закрывая замок. При выходе из критической секции  
процесс сбрасывает ее значение в 0 - замок открывается (как в случае с покупкой хлеба студентами в  
разделе "Критическая секция").

shared int lock = 0;

/\* shared означает, что \*/

/\* переменная является разделяемой \*/

while (some condition) {  
while(lock); lock = 1;

critical section  
lock = 0;

}

**Основы операционных систем**

remainder section

**54**

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию  
взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс Ро  
протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до  
присвоения переменной lock значения 1, планировщик передал управление процессу Р1. Он тоже изучает  
содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два  
процесса, одновременно выполняющих свои критические секции.

**Строгое чередование**

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать об-  
щую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка  
для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это  
выглядит так:

shared int turn = 0;

while (some condition) {  
while(turn != i);

critical section  
turn = 1-i;

remainder section

}

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по оче-  
реди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение  
turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если про-  
цесс P1 находится в **remainder section**.

**Флаги готовности**

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг  
друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процес-  
са имеют разделяемый массив флагов готовности входа процессов в критический участок

shared int ready[2] = {0, 0};

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение  
равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не  
входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или нахо-  
дится в ней.

while (some condition) {  
ready[i] = 1;  
while(ready[1-i]);

critical section  
ready[i] = 0;

remainder section

}

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в крити-  
ческий участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает  
условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После  
выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, кото-  
рый также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг  
друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой  
(deadlock). (Подробнее о тупиковых ситуациях рассказывается в лекции 7.)

**Алгоритм Петерсона**

**Основы операционных систем**

**55**

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмот-  
ренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон  
(Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готов-  
ности и к переменной очередности.

shared int ready[2] = {0, 0};

shared int turn;  
while (some condition) {  
ready[i] = 1;  
turn =1-i;

while(ready[1-i] && turn == 1-i);

critical section  
ready[i] = 0;

remainder section

}

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить крити-  
ческий участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба  
процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и пред-  
ложат выполняться друг другу. При этом одно из предложений всегда следует после другого. Тем самым  
работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Давайте докажем, что все пять наших требований к алгоритму действительно удовлетворяются.

Удовлетворение требований 1 и 2 очевидно.

Докажем выполнение условия взаимоисключения методом от противного. Пусть оба процесса одновре-  
менно оказались внутри своих критических секций. Заметим, что процесс Pi может войти в критическую  
секцию, только если ready[1-i] == 0 или turn == i. Заметим также, что если оба процесса выполняют свои  
критические секции одновременно, то значения флагов готовности для обоих процессов совпадают и  
равны 1. Могли ли оба процесса войти в критические секции из состояния, когда они оба одновременно  
находились в процессе выполнения цикла while? Нет, так как в этом случае переменная turn должна была  
бы одновременно иметь значения 0 и 1 (когда оба процесса выполняют цикл, значения переменных из-  
мениться не могут). Пусть процесс P0 первым вошел в критический участок, тогда процесс P1 должен  
был выполнить перед вхождением в цикл while по крайней мере один предваряющий оператор (turn = 0;).  
Однако после этого он не может выйти из цикла до окончания критического участка процесса P0, так как  
при входе в цикл ready[0] == 1 и turn == 0, и эти значения не могут измениться до тех пор, пока процесс  
P0 не покинет свой критический участок. Мы пришли к противоречию. Следовательно, имеет место  
взаимоисключение.

Докажем выполнение условия прогресса. Возьмем, без ограничения общности, процесс P0. Заметим, что  
он не может войти в свою критическую секцию только при совместном выполнении условий ready[1] ==

1 и turn == 1. Если процесс P1 не готов к выполнению критического участка, то ready[1] == 0, и процесс  
P0 может осуществить вход. Если процесс P1 готов к выполнению критического участка, то ready[1] == 1  
и переменная turn имеет значение 0 либо 1, позволяя процессу P0 либо процессу P1 начать выполнение  
критической секции. Если процесс P1 завершил выполнение критического участка, то он сбросит свой  
флаг готовности ready[1] == 0, разрешая процессу P0 приступить к выполнению критической работы. Та-  
ким образом, условие прогресса выполняется.

Отсюда же вытекает выполнение условия ограниченного ожидания. Так как в процессе ожидания разре-  
шения на вход процесс P0 не изменяет значения переменных, он сможет начать исполнение своего кри-  
тического участка после не более чем одного прохода по критической секции процесса P1.

**Алгоритм булочной (Bakery algorithm)**

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов.  
Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который  
получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее  
назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибы-

**Основы операционных систем**

**56**

вающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим  
номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычис-  
ления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с  
разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым об-  
служивается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом по-  
рядке). Разделяемые структуры данных для алгоритма - это два массива

shared enum {false, true} choosing[n];  
shared int number[n];

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем сле-  
дующие обозначения

(a,b) < (c,d), если a < c

или если a == c и b < d

max(a0, ax, ...., an) - это число k такое, что  
k >= ai для всех i = 0, ...,n

Структура процесса P; для алгоритма булочной приведена ниже

while (some condition) {  
choosing[i] = true;  
number[i] = max(number[0], ...,

number[n-1]) + 1;  
choosing[i] = false;  
for(j = 0; j < n; j++){  
while(choosing[j]);

while(number[j] != 0 && (number[j],j) <

(number[i],i));

}

critical section  
number[i] = 0;

remainder section

}

Доказательство того, что этот алгоритм удовлетворяет условиям 1 - 5, выполните самостоятельно в каче-  
стве упражнения.

**Аппаратная поддержка взаимоисключений**

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эф-  
фективность точно так же, как это происходит и в других областях программирования. Мы уже обраща-  
лись к общепринятому hardware для решения задачи реализации взаимоисключений, когда говорили об  
использовании механизма запрета/разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые по-  
зволяют проверить и изменить значение машинного слова или поменять местами значения двух машин-  
ных слов в памяти, выполняя эти действия как атомарные операции. Давайте обсудим, как концепции  
таких команд могут использоваться для реализации взаимоисключений.

**Команда Test-and-Set (проверить и присвоить 1)**

О выполнении команды Test-and-Set, осуществляющей проверку значения логической переменной с од-  
новременной установкой ее значения в 1, можно думать как о выполнении функции

int Test and Set (int \*target){  
int tmp = \*target;

\*target = 1;  
return tmp;

}

**Основы операционных систем**

**57**

С использованием этой атомарной команды мы можем модифицировать наш алгоритм для переменной-  
замка, так чтобы он обеспечивал взаимоисключения

shared int lock = 0;

while (some condition) {

while(Test and Set(&lock));

critical section  
lock = 0;

remainder section

}

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожида-  
ния для алгоритмов. Подумайте, как его следует изменить для соблюдения всех условий.

**Команда Swap (обменять значения)**

Выполнение команды Swap, обменивающей два значения, находящихся в памяти, можно проиллюстри-  
ровать следующей функцией:

void Swap (int \*a, int \*b){  
int tmp = \*a;

\*a = \*b;

\*b = tmp;

}

Применяя атомарную команду Swap, мы можем реализовать предыдущий алгоритм, введя дополнитель-  
ную логическую переменную key, локальную для каждого процесса:

shared int lock = 0;  
int key;

while (some condition) {  
key = 1;

do Swap(&lock,&key);  
while (key);

critical section  
lock = 0;

remainder section

}

**Заключение**

Последовательное выполнение некоторых действий, направленных на достижение определенной цели,  
называется активностью. Активности состоят из атомарных операций, выполняемых неразрывно, как  
единичное целое. При исполнении нескольких активностей в псевдопараллельном режиме атомарные  
операции различных активностей могут перемешиваться между собой с соблюдением порядка следова-  
ния внутри активностей. Это явление получило название interleaving (чередование). Если результаты вы-  
полнения нескольких активностей не зависят от варианта чередования, то такой набор активностей назы-  
вается детерминированным. В противном случае он носит название недетерминированного. Существует  
достаточное условие Бернстайна для определения детерминированности набора активностей, но оно на-  
кладывает очень жесткие ограничения на набор, требуя практически не взаимодействующих активно-  
стей. Про недетерминированный набор активностей говорят, что он имеет race condition (условие гонки,  
состязания). Устранение race condition возможно при ограничении допустимых вариантов чередований  
атомарных операций с помощью синхронизации поведения активностей. Участки активностей, выполне-  
ние которых может привести к race condition, называют критическими участками. Необходимым услови-  
ем для устранения race condition является организация взаимоисключения на критических участках:  
внутри соответствующих критических участков не может одновременно находиться более одной актив-  
ности.

Для эффективных программных алгоритмов устранения race condition помимо условия взаимоисключе-  
ния требуется выполнение следующих условий: алгоритмы не используют специальных команд процес-

**Основы операционных систем**

**58**

сора для организации взаимоисключений, алгоритмы ничего не знают о скоростях выполнения процес-  
сов, алгоритмы удовлетворяют условиям прогресса и ограниченного ожидания. Все эти условия выпол-  
няются в алгоритме Петерсона для двух процессов и алгоритме булочной - для нескольких процессов.

Применение специальных команд процессора, выполняющих ряд действий как атомарную операцию, -  
Test-and-Set, Swap - позволяет существенно упростить алгоритмы синхронизации процессов.