1. Планирование системы реального времени.

В системах реального времени **существенную роль играет время**. Чаще всего одно или несколько внешних физических устройств генерируют входные сигналы, и **компьютер должен адекватно на них реагировать в течение заданного промежутка времени**.

Системы реального времени делятся на =

1. **жесткие системы** реального времени, что означает наличие жестких сроков для каждой задачи ***(в них обязательно надо укладываться),***
2. и **гибкие системы** реального времени, в которых нарушения временного графика нежелательны, но допустимы.

В обоих случаях реализуется **раз­деление программы на несколько процессов, каждый из которых предсказуем**. Эти **процессы чаще всего бывают короткими и завершают свою работу в течение секунды**. Когда появляется внешний сигнал, именно планировщик должен обеспечить соблюдение графика.

Внешние события, на которые система должна реагировать, можно разделить на=

1. **периодические** (возникающие через регулярные интервалы времени)
2. и н**епериодические** (возникающие непредсказуемо).

**Возможно наличие нескольких пери­одических потоков событий**, которые система должна обрабатывать. В зависимости от времени, затрачиваемого на обработку каждого из событий, **может оказаться, что система не в состоянии своевременно обработать все события.**

1. Потоки. Способы реализации потоков.

**Поток** (thread) — это, сущность операционной системы, процесс выполнения на процессоре набора инструкций, точнее говоря программного кода. Общее назначение потоков — параллельное выполнение на процессоре двух или более различных задач.

### **1:1 (потоки выполнения на уровне ядра)**

Потоки выполнения, созданные пользователем в модели 1-1, соответствуют диспетчируемым сущностям ядра. Это простейший возможный вариант реализации потоковости.

**N:1 (потоки выполнения уровня пользователя)**

В модели N:1 предполагается, что все потоки выполнения уровня пользователя отображаются на единую планируемую сущность уровня ядра, и ядро ничего не знает о составе прикладных потоков выполнения. При таком подходе переключение контекста может быть сделано очень быстро, и, кроме того, он может быть реализован даже на простых ядрах, которые не поддерживают многопоточность.

### **M:N (смешанная потоковость)**

В модели M:N некоторое число M прикладных потоков выполнения отображаются на некоторое число N сущностей ядра или «виртуальных процессоров». Модель является компромиссной между моделью уровня ядра («1:1») и моделью уровня пользователя («N:1»). Вообще говоря, «M:N» потоковость системы являются более сложной для реализации, чем ядро или пользовательские потоки выполнения, поскольку изменение кода как для ядра, так и для пользовательского пространства не требуется. В M:N реализации библиотека потоков отвечает за планирование пользовательских потоков выполнения на имеющихся планируемых сущностях.

1. Прерывания. Типы прерываний.

**Прерывание** (hardware interrupt) – это событие, генерируемое внешним (по отношению к процессору ) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода-вывода (например, закончено чтение данных с диска в основную память).

**Внешние прерывания** возникают вне микропроцессора. Запросы прерываний поступают по специальным входным линиям микросхемы микропроцессора. Внешние прерывания инициируются аппаратурой. Они могут быть вызваны сигналом микросхемы таймера, сигналом от принтера, нажатием клавиши на клавиатуре и множеством других причин. Внешние прерывания не координируются с работой программного обеспечения. Различают *немаскируемые* и *маскируемые* внешние прерывания.

**Немаскируемые прерывания.** Микропроцессор извещается о возникновении немаскируемого внешнего прерывания с помощью сигнала, поступающего по его линии NMI (Non Mask Interrupt). Получив сигнал, микропроцессор после выполнения очередной инструкции производит переход по вектору 2, постоянно назначенному для немаскируемых прерываний.

**Маскируемые прерывания.** Сигнал о запросе маскируемого прерывания поступает в микропроцессор по линии INT. Выполнение текущей инструкции завершается обычным образом и в стеке сохраняется адрес следующей инструкции.

Маскируемые прерывания могут быть запрещены средствами самого микропроцессора – для этого в регистре флагов должен быть сброшен флаг прерываний IF. В таком случае микропроцессор игнорирует сигнал на линии INT и продолжает выполнять инструкции в обычном порядке. Однако когда этот флаг установлен и поступает запрос маскируемого прерывания, микропроцессор завершает выполнение текущей инструкции, после чего передаёт управление подпрограмме обработки прерывания.

**Программные прерывания**представляют обычные подпрограммы, которые вызываются прикладными программами для обработки нажатий клавиш на клавиатуре, событий от таймера, вывода сообщений на экран и других действий. Однако эти подпрограммы относятся не к прикладной программе пользователя, а к операционной системе.

**Прерывания по исключению**разделяются на несколько видов – исключения по ошибке деления, по точке остановка, по переполнению и по трассировке.

1. Механизм обработки прерываний.

*Прерывания* приостанавливают выполнение программы для выполнения специальных системных действий. Необходимость прерываний обусловлено двумя причинами: преднамеренный запрос на выполнение операций ввода-вывода на различные внешние устройства и обработка непредвиденных программных ошибок.

Обработка прерываний микропроцессором производится в три этапа:

* прерывание выполнения текущей программы;
* переход к выполнению подпрограммы обработки прерывания;
* возврат управления прерванной программе.

1. Системные вызовы.

**Системный вызов** является механизмом, который обеспечивает интерфейс между процессом и операционной системой. Это программный метод, при котором компьютерная программа запрашивает сервис у ядра ОС.

Типы системных вызовов

Вот пять типов системных вызовов, используемых в ОС:

* Контроль процесса
* Управление файлами
* Управление устройством
* Информационное обслуживание
* Связи

1. Межпроцессорное взаимодействие. Средства межпроцессорного взаимодействия.

**Межпроцессное взаимодействие** ([англ.](https://ru.m.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ***i****nter-****p****rocess****c****ommunication*, **IPC**) — обмен данными между [потоками](https://ru.m.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F) одного или разных [процессов](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)). Реализуется посредством механизмов, предоставляемых [ядром ОС](https://ru.m.wikipedia.org/wiki/%D0%AF%D0%B4%D1%80%D0%BE_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%BE%D0%B9_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D1%8B) или [процессом](https://ru.m.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), использующим механизмы [ОС](https://ru.m.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) и реализующим новые возможности IPC

|  |  |
| --- | --- |
| **Метод** | **Реализуется** [**ОС**](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) **или** [**процессом**](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) |
| [Файл](https://ru.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Сигнал](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D1%8B_(UNIX)) | Большинство [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0); в некоторых [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), например, в [Windows](https://ru.wikipedia.org/wiki/Windows), сигналы доступны только в библиотеках, реализующих [стандартную библиотеку языка Си](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B0%D0%BD%D0%B4%D0%B0%D1%80%D1%82%D0%BD%D0%B0%D1%8F_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%A1%D0%B8), и не могут использоваться для IPC. |
| [Сокет](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) | Большинство [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Канал](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B2%D0%B5%D0%B9%D0%B5%D1%80_(UNIX)) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Именованный канал](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Неименованный канал](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D0%B8%D0%BC%D0%B5%D0%BD%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%B0%D0%BD%D0%B0%D0%BB) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Семафор](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D1%84%D0%BE%D1%80_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Разделяемая память](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D0%B4%D0%B5%D0%BB%D1%8F%D0%B5%D0%BC%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Обмен сообщениями](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D1%8F%D0%BC%D0%B8)  (без разделения) | Используется в парадигме [MPI](https://ru.wikipedia.org/wiki/Message_Passing_Interface), [Java](https://ru.wikipedia.org/wiki/Java) [RMI](https://ru.wikipedia.org/wiki/RMI), [CORBA](https://ru.wikipedia.org/wiki/CORBA) и других. |
| [Проецируемый в память файл](https://ru.wikipedia.org/wiki/Mmap) (mmap) | Все [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), совместимые со стандартом [POSIX](https://ru.wikipedia.org/wiki/POSIX). При использовании [временного файла](https://ru.wikipedia.org/wiki/%D0%92%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%84%D0%B0%D0%B9%D0%BB) возможно возникновение [гонки](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5_%D0%B3%D0%BE%D0%BD%D0%BA%D0%B8). [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) [Windows](https://ru.wikipedia.org/wiki/Windows) также предоставляет этот механизм, но посредством [API](https://ru.wikipedia.org/wiki/API), отличающегося от [API](https://ru.wikipedia.org/wiki/API), описанного в стандарте [POSIX](https://ru.wikipedia.org/wiki/POSIX). |
| [Очередь сообщений](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81%D0%BE%D0%BE%D0%B1%D1%89%D0%B5%D0%BD%D0%B8%D0%B9) (Message queue) | Большинство [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |
| [Почтовый ящик](https://ru.wikipedia.org/wiki/Mailslot) | Некоторые [ОС](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0). |

1. Логическая организация взаимодействия.

Различают два способа адресации: *прямую* и *непрямую*. В случае *прямой адресации взаимодействующие процессы* непосредственно общаются друг с другом, при каждой операции обмена данными явно указывая имя или номер процесса, которому информация предназначена или от которого она должна быть получена. Если и процесс, от которого данные исходят, и процесс, принимающий данные, указывают имена своих партнеров по взаимодействию, то такая схема адресации называется ***симметричной прямой адресацией***. **Ни один другой процесс не может вмешаться в процедуру симметричного прямого общения двух процессов, перехватить посланные или подменить ожидаемые данные**

1. Сигнальные средства связи.

* Сигнальные. Передается минимальное количество информации – один бит, "да" или "нет". Используются, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Все зависит от того, знает ли он, что означает полученный сигнал, надо ли на него реагировать и каким образом. Неправильная реакция на сигнал или его игнорирование могут привести к трагическим последствиям. Вспомним профессора Плейшнера из кинофильма "Семнадцать мгновений весны". Сигнал тревоги – цветочный горшок на подоконнике – был ему передан, но профессор проигнорировал его. И к чему это привело?
* Канальные. "Общение" процессов происходит через линии связи, предоставленные операционной системой, и напоминает общение людей по телефону, с помощью записок, писем или объявлений. Объем передаваемой информации в единицу времени ограничен пропускной способностью линий связи. С увеличением количества информации возрастает и возможность влияния на поведение другого процесса.

Взаимодействуя друг с другом процессы обмениваются информацией. По объему передаваемой информации и степени возможного воздействия на поведение другого процесса средства обмена делятся на три вида.

* Сигнальные. Данный вид обмена передает минимальное количество информации – один бит, "да" или "нет". Используется, как правило, для извещения процесса о наступлении какого-либо события. Степень воздействия на поведение процесса, получившего информацию, минимальна. Процесс, получивший сигнал должен знать, что означает полученный сигнал и как на него реагировать. Неправильная реакция на сигнал или его игнорирование могут привести к непредсказуемым последствиям.
* Канальные. Операционная система предоставляет линии связи, через которые происходит обмен информации между процессами. Объем передаваемой информации в единицу времени ограничен пропускной способностью линии связи.

1. Канальные средства связи. Cмотри вопрос 8

1. Алгоритмы синхронизации. Взаимоисключение. Условие Бернстайна.

Наличие аппаратной поддержки *взаимоисключений* позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались 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 позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как *атомарные операции*. Давайте обсудим, как концепции таких команд могут использоваться для реализации *взаимоисключений*.

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.

1. Критическая секция. Требования к алгоритмам синхронизации.

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 момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам, участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.

Требования

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка 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 процессов, которые исполняются в соответствующих критических секциях. Это условие получило название 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 процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress) .
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический 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 через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting) .

1. Запрет прерываний. Переменная замок.

**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 прерыванию, 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 умыслу 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 равным 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;

remainder section}

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 P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

1. Строгое чередование. Флаги готовности.

**Строгое чередование**

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую 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 очереди: 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 процесс 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 процесса 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 находится 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 критический 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, который 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.)

1. Алгоритм Патерсона.

**Алгоритм Петерсона** — алгоритм [параллельного программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) для [взаимного исключения](https://ru.wikipedia.org/wiki/%D0%92%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%BE%D0%B5_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) потоков исполнения кода, разработанный **Гарри Петерсоном** в 1981 г. Хотя изначально был сформулирован для 2-поточного случая, алгоритм может быть обобщён для произвольного количества [потоков](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F). Алгоритм условно называется программным, так как не основан на использовании специальных команд [процессора](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) для запрета [прерываний](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), блокировки шины памяти и т. д., используются только общие переменные памяти и цикл для ожидания входа в [критическую секцию](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D1%81%D0%B5%D0%BA%D1%86%D0%B8%D1%8F) исполняемого кода.

1. Команда Test-and-Set. Команда Swap.

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

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

int Test\_and\_Set (int \*target){

int tmp = \*target;

\*target = 1;

return tmp;

}

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 ожидания* 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 проиллюстрировать 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 дополнительную 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

}

1. Механизмы синхронизации. Семафоры. Концепция семафоров. **Семафор** представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать).

**Итоги**

1.С помощью семафоров можно решить любую классическую задачу синхронизации, но по сути, семафоры - это разделяемые глобальные переменные (что является признаком плохой структуры программы)

2.Семафоры используются как для решения задачи взаимного исключения, так и для координации действий

3.Отсутствует связь между семафором и данными, доступом к которым он управляет

4.Нет контроля использования семафоров со стороны компилятора или операционной системы, соответственно - нет гарантий, что семафоры будут использованы правильно

1. Решение проблемы производителя и потребителя.

Проблема производителя-потребителя (англ .: Producer-consumer problem), также известная как проблема ограниченного буфера (англ.: Bounded-buffer problem), является классическим случаем проблем многопоточной синхронизации. Эта проблема описывает, что происходит, когда два потока, совместно использующие буфер фиксированного размера - так называемые «производитель» и «потребитель», фактически работают. Основная роль производителя - сгенерировать определенный объем данных в буфер, а затем повторить процесс. В то же время потребители используют эти данные в буфере. Ключ к этой проблеме - гарантировать, что производители не будут добавлять данные, когда буфер заполнен, а потребители не будут потреблять данные, когда буфер пуст.

1. Мониторы, мьютексы, сообщения.

**Мьютексы** Мьютекс – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят). Операции над мьютексами

•acquire(mutex) – уменьшить (занять) мьютекс

•release(mutex) – увеличить (освободить) мьютекс

•tryacquire(mutex) – часто реализуемая неблокирующая операция, выполняющая попытку уменьшить (занять) мьютекс

Мьютексы в конкретных реализациях могут иметь дополнительные свойства

•Запоминание владельца – освободить мьютекс может только поток, захвативший его

•Рекурсивность – поток может многократно захватить мьютекс (вызывать aquire()); для освобождения мьютекса поток должен соответствующее число раз вызвать release()

•Наследование приоритета – поток, захвативший мьютекс, временно наследует максимальный из приоритет потоков, ждущих освобождения данного мьютекса

**Мониторы**

Мониторы – высокоуровневый механизм взаимодействия и синхронизации процессов, обеспечивающий доступ к неразделяемым ресурсам. Предложены Хоаром (Hoare) в 1974 г. Пер Бринч Хансен первым, описал и реализовал мониторы, основывая их на идеях Хоара.

Монитор – это конструкция языка программирования, поддерживающая управляемый доступ к разделяемым данным. Монитор инкапсулирует:

•разделяемые критические данные;

•функции, использующие разделяемые данные;

•синхронизацию выполнения параллельных потоков, вызывающих указанные функции.

Доступ к переменным монитора, реализуется **только посредством** вызова предоставленных функций.

•Только **один поток** может находиться в мониторе в любой момент времени, если второй поток пытается вызвать метод монитора, он переходит в состояние ожидания до выхода из монитора первого потока.

•Код синхронизации добавляется **компилятором**.

Языки программирования, поддерживающие мониторы:

•Ада

•C# (и другие языки, использующие .NET Framework)

•Concurrent Pascal

•D

•Java (с помощью ключевого слова synchronized)

**Сообщения** Сообщения –способ межпроцессного и межпоточного взаимодействия, позволяющий потокам подавать сигналы друг другу и обмениваться данными. Используется для организации взаимодействия потоков, выполняющихся на различных узлах сети.

и

Типы доставки сообщений **Асинхронный** поток, посылающий сообщение, инициирует процесс доставки сообщения, после чего продолжает свою работу **Синхронный** поток, пославший сообщение, дожидается подтверждения его получения принимающим потоком