Основы операционных систем 1 модуль.

Андрей Тищенко

2024/2025

Лекция 2 сентября.

1 Структура вычислительной системы.

Вычислительная система состоит нескольких частей:

- Пользователь. (алгоритмы и алгоритмические языки)
- Прикладные программы. (ЦГ, matlab и т.п.)
- Системные программы. (системное программирование)
- Операционная система. (основы операционных систем)
- Техническое обеспечение. (архитектура ЭВМ и языки ассемблера)

Прекрасная притча про слепцов и слона.

1.1 Что такое операционная система?

Различные точки зрения (что такое ОС?):

- Распорядитель ресурсов.
- Защитник пользователей и программ.
- Виртуальная машина (фокусник, виртуальная память).
- Кот в мешке (попросил загрузить ОС, значит всё загруженное и есть ОС).
- Постоянно функционирующее ядро.

Проще сказать не что такое операционная система, а для чего она нужна и чем занимается. Современные ОС это продукт эволюции вычислительных систем, поэтому стоит эту эволюцию рассмотреть.

1.1.1 Эволюция вычислительных систем

Удобство, стоимость и производительность — самые главные факторы отбора в эволюции операционных система.
Условные этапы развития вычислительных систем:

1-й период. (1945–1955гг.) Научно-исследовательнская работа в области вычислительной техники.

- Ламповые машины.
- Нет разделения персонала.
- Вход программы коммутацией или перфокартами.
- Одновременное выполнение толко одной операции.
- Появление прообразов первых компиляторов.
- Нет операционных систем.

2-й период. (1955-начало 60-х гг.) Начало использования вычислительных машин в научных и коммерческих целях.

- Транзисторные машины
- Происходит разделение персонала (появление малочисленной касты программистов)
- Бурное развитие алгоритмических языков (напиши 10 000 строк на асме и отладь, это стало причиной появления первых языков программирования)
- Ввод заадния колодой перфокарт
- Вывод результатов на печать
- Пакеты заданий и системы пакетной обработки.

3-й период. (начало 60-х годов–1980 гг.) Конец этого периода указан точно. Тактовая частота значительно повысилась из-за появления микросхем.

- Машины на интегральных схемах.
- Использование спулинга spooling. (появление процессоров ввода/вывода, которые включаются вместо центрального процессора)
- Планирование заданий (из-за появления магнитных дисков вместо ленты).
- Мультипрограммные пакетные системы. (в память загружается несколько программ, пока происходит ввод/вывод на в одной программе ЦП передаётся другой программе)

- Системы разделения времени (time-sharing).
- Интерактивная отладка программ.
- Виртуальная память.
- Появление семейств ЭВМ.

Мультипрограммирование и эволюция вычислительных систем.

Software	**Hardware**
Планирование заданий	Защита памяти
Управление памятью	Сохранение контекста
Сохранение контекста	Механизм прерывания
Планирование использования процессора	Привилегированные команды
Системные вызовы	
Средства коммуникации	
Средства синхронизации	

Интерактивная отладка стала возможна благодаря изобретению терминала, но возникла новая проблема: нужно хранить информацию разных пользователей так, чтобы они не могли портить данные других пользователей, также потребовался единый стандарт хранения данных в компьютере.

Хотелось разрешить как можно большему числу пользователей одновременно пользоваться вычислительной машиной. Для этого нужно было экономить оперативную память. С целью решения этой проблемы программы подгружаются в машину по частям (только нужные в данный момент куски). На этой концепции основана виртуальная память.

Семейства ЭВМ были нужны для того, чтобы маленькие кампании могли покупать маломощные компьютеры, писать свой код на них, по мере роста кампании покупать более мощные машины того же семейства и исполнять такой же код без переписывания и перекомпиляции.

4-й период (1980–2005 гг.)

- Машины на больших интегральных схемах (БИС).
- Персональные ЭВМ.
- Дружественное программное обеспечение.
- Сетевые и распределённые операционные системы.

Поскольку теперь любой человек мог стать владельцем персонального ЭВМ, было необходимо создать дружелюбный интерфейс для предоставления человеку без специального образования пользоваться персональной машиной.

Из-за увелечения количества компьютеров увеличилась важность компьютерных сетей. Пользователи, работающие на сетевой ОС должны были понимать как доставать и обрабатывать файлы из сервера.

Владелец распределённой операционной системы не должен понимать, где находится его файл, обращение к файлу на локальной машине и к файлу на сервере, управляемом распределённой ОС, должно выглядеть для пользователя одинаково.

Широкое использование ЭВМ в быту, образовании и на производстве.

5-й период (2005-?? гг.)

- Машины на многоядерных процессорах.
- Мобильные компьютеры.
- Высокопроизводительные вычислительные системы.
- Облачные технологии.
- Виртуализация выполнения программ.

Можно виртуализовать программы и операционные системы.

Период Глобальной компьютеризации.

1.2 Основные функции ОС

- Планирование заданий и использования процессора. (кто и после кого будет выполняться)
- Обеспечение программ средствами коммуникации и синхронизации.
- Управление памятью.
- Управление файловой системой.
- Управление вводом-выводом.
- Обеспечение безопасности.

Операционные системы существуют потому что на данный момент их существование — это разумный способ использования вычислительных систем.

2 Архитектурные особенности построения ОС

Внутреннее строение ОС (разновидности)

Монолитное ядро.

- Каждая процедура (функция) может вызвать каждую.
- Все процедуры работают в привилегированном режиме.
- Ядро совпадает со всей операционной системой.
- Пользовательские программы взаимодействуют с ядром через системные вызовы.

Многоуровневые (Layered) системы.

- Процедура уровня K может вызывать только процедуры уровня $K-1$
- Все или почти все уровни работают в привилегированном режиме
- Ядро совпадает или почти совпадает со всей операционной системой
- Пользовательские программы взаимодействуют с ОС через интерфейс пользователя

Самый низкий уровень — Hardware. Самый высокий — интерфейс пользователя. В самой первой такой машине выглядело так:

1. Интерфейс пользователя.
2. Управление ввода вывода.
3. Связь с консолью.
4. Управление памятью.
5. Планирование заданий и программ.

Работает медленнее, чем система монолитного ядра, но отладка и модификация сильно упрощается.

Микроядерная (microkernel) архитектура.

Функции микроядра:

- Взаимодействие между программами.
- Планирование использования процессора.
- Первичная обработка прерываний и процессов ввода вывода
- Управление памятью.

Устройство системы:

- Микроядро составляет лишь малую часть ОС.
- В привилегированном режиме работает только микроядро.
- Взаимодействие частей ОС между собой и с программами пользователей путем передачи сообщений через микроядро.

Все части ОС, кроме микроядра, могут быть заменены без прерывания работы ОС. Микроядро сразу же сможет обратиться к заменённой части системы.

Микроядро в таком концепте работает весьма медленно, поэтому на практике для повышения быстродействия функционал микроядра расширили.

Виртуальные машины

Каждому пользователю предоставляется своя копия виртуального hardware

Новая микроядерная архитектура (экзоядерная архитектура).

- Взаимодествие между программами
- Выделение и выспобождение физических ресурсов
- Контроль прав доступа

Вся остальная функциональность выделяется на поддержание абстракций, которые разделены на библиотеки. Считается спорным подходом и на практике не используется (только в учебных ОС).

Смешанные системы.

Монолитное ядро - необходимость перекомпиляции при каждом изменении, сложность отладки, высокая скорость работы.
Многоуровневые системы - необходимость перекомпиляции при изменениях, отлаживается только изменённый уровень, меньшая скорость работы.
Микроядро - простота отладки, возможность замены компонент, без перекомпиляции и остановки системы, очень медленные.
Все три подхода используются в совеременных ОС.

3 Понятие процесса. Операции над процессами.

Уточнение терминологии

Термин *программа* - не может использоваться для описания происходящего внутри ОС.
Термин *задание* - не может использоваться для описания происходящего внутри ОС.
Изначально эти термины были придуманы для статических объектов, а нам нужно описывать динамические объекты.
Введём термин *процесс* для описания динамических объектов.
Теперь под *процессом* будем понимать совокупность:

- набора исполняющихся команд
- ассоциированных с ним ресурсов
- текущего момента его выполнения

Вся эта совокупность находится под управлением операционной системы.
Важно: Процесс $\neq$ программа, которая исполняется.

- для исполнения одной программы может организовываться несколько процессов
- В рамках одного процесса может исполняться несколько программ
- В рамках процесса может исполняться код, отсутствующий в программе.

Состояния процесса

1. Исполняется
2. Не исполняется

Новые процессы не исполняются. Переход из состояний происходит в случае выбора его для исполнения или приостановки. После исполнения процесс выходит из ОС.

Примитивная система, которая не является верной, так как процесс может "не исполняться разными способами".

1. Готовность
2. Исполнение

3. Ожидание

Новые процессы попадают в состояние готовности. Выбор готового процесса для исполнения прерывает в состояние исполнения. В случае прерывания исполняемого процесса он переходит в состояние готовности. Если для исполнения нужно дождаться какого-то события, то он переходит в состояние ожидания, после исполнения ожидаемого, он переходит в состояние готовности. Однако эту схему тоже стоит дополнить:

1. Рождение
2. Готовность
3. Исполнение
4. Ожидание
5. Закончил исполнение

Новые процессы попадают в состояние рождения, после прохождения допуска к планированию процесс попадает в состояние готовности. При завершении работы процесса он попадает в состояние "закончил исполнение". **Важно:** переход между состояниями процесса осуществляет ОС.

Лекция 16 сентября

Операции над процессами:

- Создание процесса - завершение процесса
- Запуск процесса - приостановка процесса
- блокировка процесса - разблокирование процессса
- изменение приоритета

Чтобы ОС могла взаимодействовать с процессами, она должна иметь какое-то представление об этом объекте. Эта информация хранится в PCB (process control block).
Содержимое PCB

- Состояние процесса
- Програмнный счётчик (следующая команда в процессе)
- содержимое регистров

- данные для планирования использования процессора и управления памятью
- учётная информация
- сведения об устройствах ввода-вывода, связанных с процессом (возможно файлы, с которыми взаимодействует).

Регистровый контекст: программный счётчик, содержимое регистров.
Всё остальное входит в системный контекст.
Всё, что лежит в PCB лежит в ядре ОС, код и данные программы называются пользовательским контекстом, входят в адресное пространство процесса.
Создание процесса:

- Присвоение идентификационного номера
- Порождение нового PCB с состоянием процесса "рождение"
- Выделение ресурсов (из ресурсов родителя и ОС)
- Занесение в адресное пространство кода и установка значения программного счётчика. Может создаться дубликат родителя, владеющий всей информацией родителя до момента желания создать дочерний процесс. Информация процесса может быть получена из файла (пользовательский контекст убирается, вместо него через системный вызов открывается исполняемый файл и подгружается как пользовательский контекст).
- Окончание заполнения PCB
- Изменение состояния процесса на "готовность"

Завершение процесса: Это состояние необходимо для того, чтобы родитель мог узнать, по какой причине ребёнок завершил процесс.

- Изменение состояния процесса на "закончил исполнение"
- Освобождение ресурсов (остаётся только PCB процесса, в нём остаётся учётная информация, родитель)
- Очистка соответствующих элементов в PCB
- Сохранение в PCB информации о причинах завершения.

После окончания родительского процесса некоторые ОС убивают всех его детей, а некоторые (например Windows и Unix) позволяют детям жить.
Но в таком случае возникает вопрос с новым родителем. Например у процесса 251 был ребёнок 1000, после окончания 251 1000 продолжил работать, ОС создала новый процесс 251, который не должен являться родителем 1000. Поэтому осиротевшие процесса усыновляются процессами, которые существуют, пока ОС не прекратит работу.
Если процессы используют какой-то ресурс (например файл), ему присваивается счётчик, равный количеству процессов, использующих этот ресурс. Освобождение происходит если один из процессов явно это попросил или если счётчик стал равен нулю.
Запуск процесса:

- Изменение состояния процесса на "исполнение"
- Обеспечение наличия в оперативной памяти информации, необходимой для его выполнения
- Восстановление значений регистров
- передача управления по адресу, на который указывает программный счётчик.

PCB меняется только при переключении состояния процесса. Во время работы процесса ничего в PCB не заносится (так как это было бы непроизводительно).
Приостановка процесса:

- Автоматическое сохранение программного счётчик и части регистров (работа hardware)
- Передача управления по специальному адресу (hardware)
- Сохранение динамической части... UNFINISHED

Блокирование процесса

- Сохранение контекста процесса в PCB.
- Обработка системного вызова (разбираемся, чего мы ждём)
- Перевод процесса в состояние "ожидание"

Разблокирование процесса:

- Уточнение тоо, какое именно событие произошло

- Проверка наличия процесса, ожидающего этого события
- Перевод ожидающего процесса в состояние "готовность"
- Обработка произошедшего события

4 Кооперация процессов и основные аспекты ее логической организации

Основные причины объединения усилий:

- Повешение скорости решения задач.
- Совместное использование данных.
- Модульная конструкция какой-либо системы.
- Для удобства работы пользователя (например, отладчик).

Кооперативные или взаимодейтсвующие процессы - это процессы, которые влияют на поведение друг друга путем обмена информацией.
Категории средств взаимодействия:

- Сигнальные (переданная информация зачастую ограничивается битами).
- Канальные (создание логического канала связи. Данные из одного процесса могут быть получены при желании другого, тогда произойдёт передача запрошенных данных)
- Разделяемая память (доступна обоим процессам).

Как устанавливаестя связь

- Нужна или не нужна инициализация?
- Способы адресации
  1. Прямая адресация
     - симметричная
     - асимметричная
  2. Непрямая или косвенная адресация

Информационная валентность процессов и средств связи

- Сколько процессов может быть ассоциировано с конкретным средством связи?
- Сколько идентичных средств связи может быть задействовано между двумя процессами?
- Направленность связи
  - Симплексная связь (только в одну сторону)
  - Полудуплексная связь (двусторонняя, но только по очереди, как рация)
  - Дуплексная связь (двусторонняя)

Лекция 23 сентября.

Основные аспекты логической организации передачи информации

Особенности канальных средств связи

Буферизация

- Буфера нет (нулевая ёмкость). Процесс-передатчик всегда обязан ждать приёма
- Буфер конечной ёмкости. Процесс-передатчик обязан ждать освобождения места в буфере перед продолжением работы.
- Буфер неограниченной ёмкости (нереализуемо!). Процесс-передатчик никогда не ждёт.

Модели передачи данных

- Потоковая модель. Операции приёма/передачи не интересуются содержимым данных и их происхождением. Данные не структурируются.
- Модель сообщений. На передаваемые данные накладывается определённая структура.

Потоковая модель - pipe

Источники вкладывают информацию порциями любого размера и могут считывать эти порции в любом размере. Через pipe могут общаться только процессы, имеющие общего предка, создавшего pipe. Это происходит

потому что о положении начала и конца pipe знает только его создатель (и дети). Без средств синхронизации это сторого односторонняя связь.
Потоквая модель - FIFO

Представляет собой pipe с помеченным входом и выходом, что позволяет неродственным процессам общаться через него.
Модель сообщений

Сообщения в pipe имеют чёткие границы, прочитать сообщение можно только целиком. У принимающего процесса пропадает возможность получать данные в произвольном размере.

Надёжность средств связи

Средство связи считается надёжным, если:

- Нет потери информации
- Нет повреждения информации
- Нет нарушения порядка поступления информации
- Не появляется лишняя информация

Современные машины считаются надёжными.

Как завершается связь

- Нужны ли специальные действия для прекращения использования средства связи?
- Как влияет прекращение использования средства связи одним процессом на поведение других участников взаимодействия?

Нити исполнения (threads)

Для реализации многопоточности нужно из основного процесса создать ещё один процесс. Переключить контекст, запросить доступ к общей памяти (или другому средству связи). Такие накладные расходы нужны для реализации простейшей многопоточности. На практике из-за накладных расходов это работате ещё медленнее. Поэтому придумали thread'ы.
На удивление хорошо поведение процесса можно сравнить с игрушечной железной дорогой. Шпалы принимаются за инструкции, развилки — условные переходы. Станции — данные со стека или input/output.

Сам поезд можно принимать за регистры и данные в стеке.
Если поставить две такие дороги рядом, то можно получить аналог второго процесса.
А если поставить на те же рельсы второй поезд, можно получить аналог thread. Так как поезда разные, у них разные данные на регистрах и в стеке, всё остальное у них общее, так же должна присутствовать информация, позволяющая различать поезда между собой.
Процесс: системный контекст, регистровый контекст, код, данные вне стека, стек.
Нить исполнения: системный контекст нити, регистровый контекст, стек.
При появлении процесса нить всего одна (называется главной нитью или master нитью). Нити также могут находиться в состоянии готовности, ожидания, исполнения и т.д.
Нить находится в состоянии

- Готовности, если нет ни одной нити в состоянии исполнения и есть хотя бы одна в состоянии готовности
- Ожидания, если нет ни одной нити в состоянии готовности и исполнениия.
- Исполнения, если хотя бы одна из нитей находится в состоянии исполнения.
- Закончил исполнения, если все его нити находятся в состоянии закончил исполнение.

По-прежнему нужно тратить ресурсы на создание новой нити (нить создаётся легче, чем процесс). Между нитями нет нужды создавать дополнительные средства связи, так как внестековое пространство у них и так общее. Переключение контекста между нитями происходит быстрее, чем между процессами. Везде выиграли.

Алгоритмы синхронизации

Активности и атомарные операции

Активность — последовательное выполнение ряда действий, направленных на достижение определённой цели.
Активность: приготовление бутерброда.

- Отрезать ломтик хлеба
- Отрезать ломтик колбасы

- Намазать хлеб маслом
- Положить колбасу на хлеб

Каждый элемент активности является атомарным (неделимым). Во время операции никуда отвлекаться нельзя, между ними - можно.
Пусть активность P: a b c, Q: d e f. (какие-то операции)
При последовательном выполнении PQ: a b c d e f.
Псевдопараллельное выполнение
(режим разделения времени): a d b e c f. В таком случае атомарные операции одного процесса расположены в правильном порядке, поэтому на итоговый результат не влияет.

Детерминированность выбора

P: $x = 2,\ y = x - 1$, Q: $x = 3,\ y = x + 1$
При последовательном выполнении PQ: (x, y) = (3, 4).
Однако перестановка $x = 2, x = 3, y = x - 1, y = x + 1 \Rightarrow$ (x, y) = (3, 3), приводит к недетерминированности.

Условия Бернстайна

Пусть P: 1. $x = u + V$, 2. $y = x \cdot w$. Входные данные: $R_1 = \{u,\ v\}$, $R_2 = \{x,\ w\}$. Выходные данные: $W_1 = \{x\}$, $W_2 = \{y\}$.
Вход для активности $R(P) = \{u,\ v,\ x,\ w\}$. Выход для активности $W(P) = \{x,\ y\}$
Тогда достаточные условия детерминированности Бернстайна:

1. $W(P) \cap W(Q) = \varnothing$
2. $W(P) \cap R(Q) = \varnothing$
3. $R(P) \cap W(Q) = \varnothing$

Если все условия выполнены, то набор активностей $\{P,\ Q\}$ является детерминированным.
В недетерминированных наборах всегда встречается race condition (состояние гонки).
Избежать недетерминированного поведения при неважности очередности доступа можно с помощью *взаимоисключения* (mutual exclusion).

Критическая секция

Пример про трёх студентов и пиво в пятницу. Каждый из студентов приходит в комнату, видит, что пива нет и уходит в магазин покупать. В итоге каждый из них купил пива на троих и получилась славная пьянка. Поэтмоу операцию уйти за пивом, купить на всех и вернуться с пивом стоит заменить атомарной операцией. Тогда первый студент выполнит эту операцию и двум другим ничего делать не придётся.
Структура кооперативного процесса

```
while (some condition) {
entry section
    critical section
exit section
    remainder section
}
```

Требования к программным алгоритмам

1. Программный алгоритм должен быть программным (проблемы решаются в коде, а не на уровне hardware)
2. Нет предположений об относительных скоростях выполнения и числе процессоров
3. Выполняется условие взаимоисключения (mutual exclusion) для критических участков.
4. Выполняется условие прогресса (progress).
   - Решения принимают только те, кто готов войти в критическую секцию
   - Решение должно приниматься за конечное время
5. Выполняется условие ограниченного ожидания (bound waiting). Для каждого процесса можно сказать, какое максимальное (конечное) количество процессов он может пропустить, после чего гарантированно пройдёт в критическую секцию.

Лекция 30 сентября

Подробнее о пункте 3. Мы хотим заставить участки процесса выполняться атомарно по отношению процессам, входящим в набор взаимодействующих процессов. Никакие два процесса из набора не могут оказаться в критическом участке в состоянии исполнения или готовности одновременно.

История придумывания алгоритма синхронизации

Запрет прерываний

Существует команда Client Disable Interrupt, она говорит процессору отключить прерывания (кроме критических ошибок (nonmaskable interrupt) и прерываний, созданных пользователем). Также есть команда, возвращающая процессору возможность видеть прерывания. Если перед критической секцией поставить запрет на прерывания, а после этой секции включить прерывания, то процессы внутри будут выполняться атомарно. Минусы: бесконечный цикл внутри критической секции может быть прерван только выключением вычислительной машины. Поэтому такой метод используется только при программировании на уровне ядра.

Переменная “замок”

Оборачивать код в нечто похожее на

```
shared int lock = 0;
while (lock == 1);
lock = 1;
critical section
lock = 0;
```

Минусы: оба процесса хотят войти в критическую секцию, первый проходит проверку на lock, заходит внутрь, но сделать lock = 1 он не успевает, так как у него забирают управление. Второй процесс входит в секцию, проверяет lock, заходит в критическую секцию, ставит lock = 1, после чего у него забирают управление. В результате мы имеем два процесса, один из которых исполняется в критической секции, а другой - в готовности в этой же критической секции.

Строгое чередование

Пусть у нас два процесса. Оборачиваем критическую секцию следующим образом:

```
shared int turn = 1;
```

$Process_0$

```
while (condition) {
    while (turn != 0);
```

```
        critical section
    turn = 1;
    remainder section
}
```

$Process_1$

```
while (condition) {
    while (turn != 1);
        critical section
    turn = 0;
    remainder section
}
```

Процессы не могут одновременно находиться в критической секции, так как тогда переменная turn должна быть 0 и 1 одновременно, что невозможно. Однако условие прогресса нарушается. Пусть у $Process_0$ долгая remainder section, а у $Process_1$ обе секции короткие. Тогда первый процесс может прокрутиться внутри цикла и встать в очередь на вход в критическую секцию. Тем временем нулевой процесс всё ещё в remainder section и не может поставить флаг для первого процесса.

Флаги готовности

Пусть у нас два процесса. Оборачиваем критическую секцию следующим образом:

```
    shared int ready[2] = {1, 0};
```

$Process_0$

```
while (condition) {
    ready[0] = 1;
    while (ready[1]);
        critical section
    ready[0] = 0;
    remainder section
}
```

$Process_1$

```
while (condition) {
    ready[1] = 1;
```

```
    while (ready[0]);
        critical section
    ready[1] = 0;
    remainder section
}
```

Здесь нарушается вторая часть условия прогресса, так как оба процесса могут одновременно быть готовыми ко входу в критическую секцию, тогда они оба будут крутить бесконечный цикл по очереди.

Алгоритм Петерсона

Совмещаем флаг и переменную готовности.

```
shared int ready[2] = {1, 0};
shared int turn;
```

$Process_0$

```
while (condition) {
    ready[0] = 1;
    turn = 1;
    while (ready[1] && turn == 1);
        critical section
    ready[0] = 0;
    remainder section
}
```

$Process_1$

```
while (condition) {\
ready[1] = 1;
turn = 0;
while (ready[0] && turn == 0);
    critical section
ready[1] = 0;
remainder section
}
```

В этом случае для двух процессов все условия выполняются.

Bakery algorithm

Если бы алгоритм придумали в России, то назвали бы алгоритм регистратуры в поликлинике (так как пекарни на западе выпекают заказы, а не предлагают готовый ассортимент).

Основные идеи

1. Процессы можно сравнивать по именам.
2. Перед входом в критическую секцию процессы получают талончик с номером.
3. Первым в критическую секцию входит процесс с наименьшим номером на талоне.
4. Выписывание талона - неатомарная операция. Могут быть талоны с одинаковыми номерами
5. При совпадении номеров на талоне первым входит процесс с меньшим значением имени процесса.

Последние два алгоритма прекрасно работали до 2005 года. В один роковой день появились многоядерные процессоры. Теперь разные ядра могли писать в одну и ту же память, в результате чего одно ядро могло записать в память одно число, после чего считать оттуда же другое, ведь с этой памятью поработало другое ядро.
Для решения проблемы было принято решение ослабить консистентность памяти.

Аппаратная поддержка

Команда Test-And-Set

Выполняет нечто похожее на это:

```
int Test_And_Set (int* a) {
    int tmp = *a;
    *a = 1;
    return tmp;
}
```

Используется для сокращение прологов и эпилогов:

```
shared int lock = 0
while (condition) {
    while(Test_And_Set(lock));
        critical section
    lock = 0;
        remainder section
}
```

Однако это нарушает условие ограниченного ожидания (исправление ошибки оставлено лектором как несложное упражнение).

Команда Swap

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

Можно использовать так:

```
shared int lock = 0
int key = 0;
while (condition) {
    key = 1;
    do Swap(&lock, &key);
    while(key);
    critical section
    lock = 0
    remainder section
}
```

Однако это нарушает условие ограниченного ожидания (исправление ошибки оставлено лектором как несложное упражнение).

Механизмы синхронизации

Недостатки программных алгоритмов

Непроизводительная трата процессорного времени в циклах пролога

Посмотрим

```
while (condition) {
entry section
    critical section
exit section
    remainder section
}
```

Хотелось бы сократить убрать цикл активного ожидания (busy wait) в пассивное ожидание. Это не всегда оправдано, но зачастую является таковым. То есть просим процесс не крутиться в цикле, а перейти в состояние ожидания (обратное переключение с синхронизацией могут занимать больше времени, чем просто прокрутка по циклу).

Задачи с разными приоритетами

Задача с высоким приоритетом с более важным видом ждёт очереди на вход в критическую секцию.

Семафоры Дейкстры

S — семафор. Целая разделяемая переменная с неотрицательными значениями. При создании может быть инициализирована любым неотрицательным значением.

Допустимые атомарные операции

- P(S): пока S == 0 процесс блокируется;
  Также выполняется S = S - 1
- V(S): S = S + 1

Проблема Producer-Consumer

Producer:

```
while(1) {
    produce_item();
    put_item();
}
```

Consumer:

```
while(1) {
    get_item();
    consume_item();
}
```

Синхронизируем их работу с помощью семафоров.

```
Semaphore mut_ex = 1;
Semaphore full = 0;
Semaphore empty = N;
```

Producer:

```
while(1) {
    produce_item();
    P(empty);
    P(mut_ex);
    put_item();
    V(mut_ex);
    V(full);
}
```

Consumer:

```
while(1) {
    P(full);
    P(mut_ex);
    get_item();
    V(mut_ex);
    V(empty);
    consume_item();
}
```

Интересный факт: если поменять местами P(full) и P(mut_ex) у Consumer, то можно словить deadlock. Эта ошибка происходит не всегда, найти её сложно, поэтому с семафорами нужно работать осторожно.

Мониторы Хора

Структура

```
Monitor monitor_name {
    params
    void m_1(...) {...}
    void m_2(...) {...}
    ...
    void m_n(...) {...}
    variables initialization
}
```

Условные переменные (condition variables)

Condition C;

- C.wait
  Процесс, выполнивший операцию wait над условной переменной, всегда блокируется
- C.signal
  Выполнение операции signal приводит к разблокированию только одного процесса, ожидающего этого (если он не существует). Процесс, выполнивший операцию signal, немедленно покидает монитор.

Попытаемся решить задачу Producer-Consumer через мониторы Хора:

```
Monitor Prod_Cons {
    Condition full, empty;
    int count;
    void put() {
        if (count == N) full.wait;
        put_item(); count++;
        if (count == 1) empty.signal;
    }
    void get() {
        if (count == 0) empty.wait;
        get_item(); count--;
        if (count == N - 1) full.signal;
    }
    Prod_Cons(): count(0) {}
}
```

Producer:

```
while(1) {
    produce_item();
    Prod_Cons.put();
}
```

Consumer:

```
while(1) {
    Prod_Cons.get();
    consume_item();
}
```