Основы операционных систем

Лекции

Савва Чубий, БПИ233

2024–2025

2024-09-02

2024-09-09

2024-09-16

2024-09-23

2024-09-30

2024-09-02

Обзор. Введение

Карпов Владимир Ефимович

carpson@mail.ru

Оценка:

$$О_{теор}\frac{О_{кр}+О_{экз}}{2}$$

$$О_{итог}\frac{2*О_{практ}*О_{теор}}{О_{практ}+О_{теор}}$$

0 за $О_{теор}$ или $О_{практ}$ обнулит $О_{итог}$

Структура вычислительной системы

- Пользователь (человек или устройство)
- Программное обеспечение
  - Прикладные программы
  - Системные программы
    - Прочие системные программы
  - Операционная система
- Техническое обеспечение

Границы между прикладным и системными программами условны

Определение операционной системы. Основные точки зрения

- Распорядитель ресурсов
- Защитник (разграничитель) пользователей и программ
- Виртуальная машина (создание абстракций для работы с файлами, памятью и тд)
- «Кот в мешке»
- Постоянно функционирующее ядро (программа, которая поддерживает работу других программ)
- ...

Эволюция вычислительных систем

Факторы отбора эволюции выч. систем:
- Удобство пользования
- Стоимость
- Производительность

Развитие hard- и software нужно рассматривать совокупно

Периоды развития:

1-ый период (1945 – 1955)

- Ламповые машины
  - Огромные
  - Часто перегорали

- Нет разделения персонала
- Ввод программы коммутацией или перфокартами
- Одновременное выполнение только одной операции (те только либо программирование, либо расчеты, либо счет перфокарты и тд)
- Появление прообразов первых компиляторов
- Нет операционных систем

Фон Нейман имеет минимальное отношение к принципам Фон Неймана

ENIAC работал в 10-ой ссч

2-ой период (1955 – начало 60-ых)

- Полупроводниковые компьютеры
- Разделение персонала
- Бурное развитии алгоритмических языков
- Ввод задания колодой перфокарт
- Вывод результатов на печать
- Пакеты заданий и системы пакетной обработки (прообраз операционных систем)

Начало использования ЭВМ в коммерческих и исследовательских целях

3-период (начало 60-ых – 1980)

- Микросхемы, машины на интегральных схемах
  - Машины меньше
  - Сигнал идет быстрее, можно повысить тактовую частоту
  - Считыватели и принтеры становятся bottleneck
- Использование спулинга (отдельные процессоры для ввода, вывода и счета)
- Планирование заданий (из-за создания магнитных дисков)
- Мультипрограммные пакетные системы
- Системы разделения времени (time-sharing)

Изменения из-за мультипрограммирования:

Software	Hardware
• Планирование заданий • Управление памятью • Сохранение контекста • Планирование использования процессора • Системные вызовы • Средства коммуникации между программами • Средства синхронизации	• Защита памяти • Сохранение контекста • Механизм прерываний • Привилегированные команды (в тч команды ввода/ вывода)

2024-09-09

Опр. Квант времени — время, пока программа работает подряд (без передачи управления другим программам)

Раньше клавиатура и дисплей стали независимы, потом превратились в терминалы, которые выводили данные во время выполнения программы

Появилась возможность **отладки**

Появляются **файловые системы** (много пользователей могут работать на одном устройстве хранения данных)

Программа обычно считается частями: всё программу хранить в оперативной памяти не обязательно

Появляется концепция **виртуальной памяти**: абстракция, иллюзия большой оперативной памяти

Появилась идея **обратной совместимости**, **полной совместимости**, **линеек устройств** (от слабых компьютеров до мейнфреймов)

Популярные линейки:
- IBM
- PDP

Обратная совместимость имеет преимущества, но и заставляет «тащить» за собой недостатки

В опр момент IBM решили, что баги в системе править не будут, так как возникают новые баги

4-ый период (1980 – 2005)

- 1980 год — развитие больших интегральных схем: весь процессор мог быть на одном кристалле
- Первые персональные ЭВМ
- Дружественное программное обеспечение: программы пишутся для удобства пользователей
- Резкая деградация ОС: пропадает мультипроцессорность, защита памяти и т.д.
- Из-за роста мощности (в 90-е) деградация ОС прекращается
- Переосмысление роли сетей: из оборонки в пользовательские
- Сетевые и распределенные ОС

Сетевая ОС — пользователь явно использует возможности сети

Распределенная ОС — пользователь неявно использует возможности сети, используется абстракция

Период широкого использования ЭВМ в быту, в образовании, на производстве

5-ый период (2005 – ??)

- Появление многоядерных процессоров
- Мобильные компьютеры
- Высокопроизводительные вычислительные системы
- Облачные технологии
- Виртуализация выполнения программ: выполнение программы на любом из компьютеров распределительной сети

Период глобальной компьютеризации

—— Основные функции ОС ——

- Планирование заданий и использование процессора
- Обеспечение программ средствами коммуникации и синхронизации (межпроцессорные коммуникации)
- Управление памятью
- Управление файловой системой
- Управление вводом-выводом

- Обеспечение безопасности

Дальше в курсе будем изучать, как эти функции выполняются по отдельности и совместно

Архитектурные особенности построения ОС

Внутреннее строение ОС

- Монолитное ядро:
  - Каждая процедура может вызывать каждую
  - Все процедуры работают в привилегированном режиме
  - Ядро совпадает со всей операционной системой (вся ОС всегда сидит в оперативной памяти)
  - Точки входа в ядро — системные вызовы
+, -	
– Быстродействие	– Нужно много памяти – Невозможность модификации без полной перекомпиляции

- Многоуровневая (Layered) система:
  - Процедура уровня K может вызывать только процедуры уровня $K - 1$
  - [Почти] все уровни работают в привилигировнном режиме
  - Ядро [почти] совпадает со всей операционной системой
  - Точка входа — верхний уровнеь
+, -	
– Легкая отладка (при удачном проектировании)	– Медленно – Нужно много памяти – Невозможность модификации без полной перекомпиляции

- Микроядерная (microkernel) архитектура:
  - Функции микроядра:
    - взаимодействие между программами
    - планирование испльзования процессора
    - ...
  - Микроядро — лишь малая часть ОС
  - Остальное — отдельные программы-"менеджеры", раб в пользовательском режиме
  - Всё общение через микроядро
+	-
– Только ядро — «особенное» – Легче отлаживать и заменять компоненты	– Ядро перезагружено — bottleneck – Всё очень-очень медленно работает

- Виртуальные машины
  - У каждого пользователя своя копия hardware
  - Пример:
    - Реальное hardware
      - Реальная ОС
        - Виртуальное hardware - Linux - Пользователь 1
        - Виртуальное hardware - Windows 11 - Пользователь 1
        - Виртуальное hardware - MS-DOS - Пользователь 1

+	-
– Удобно	– Медленно из-за многоуровневости

- Экзоядерная (новая микроядерная) архитектура:
  - Функции экзоядра:
    - взаимодействие между программами
    - выделение и высвобождение физических ресурсов
    - контроль прав доступа
  - Весь остальной функционал выкидывается в библиотеки

Подходы **не** используются в чистом виде

Понятие процесса. Операции над процессами

Процесс

Термины «программа» и «задание» были придуманы для статических объектов

Для динамических объектов будем использовать «процесс»

Процесс характеризует совокупность:
- набора исполняющихся команд
- ассоциированных с ним ресурсов
- текущего момента его выполнения (контекст)

Процесс $\neq$ программа, которая исполняется тк:
- одна программа может использовать несколько процессов
- один процесс может использовать несколько программ
- процесс может исполнять код, которого не было в программе

Состояние процесса

Процесс сам состояния не меняет, его переводит ОС, совершая «операцию»

2024-09-16

Набор (пар) операций:
- одноразовые:
  - создание – завершение
- многоразовые:
  - запуск – приостановка
  - блокирование – разблокирование
  - изменение приоритета процесса

Process Control Block и контекст процесса

Process Control Block — структура (или набор структур) с информацией о процессе; хранит:
- Регистровый контекст:
  - Программный счетчик
  - Содержимое регистров
- Системный контекст:
  - Состояние процесса
  - Данные для планирования использования процессора и управления памятью
  - Учетная информация

PCB хранится в адресном пространстве ОС

Код и данные в адресном пространстве — пользовательский контекст

Контекст процесса — совокупность всех трех контекстов

Одноразовые операции

Медленные тк:
- Выполняют много действий
- Выделяют/ освобождают ресурсы
- Меняют число процессов

Первые несколько процессов создаются «хитро»

Всякий новый процесс рождается от другого процесса → процессы образуют генеологическое дерево (или лес)

Создание процесса

1. Присваивание PID
2. Создание PCB с состоянием «рождение»
3. Выделение ресурсов.

   Подходы:
   1. из ресурсов родителя
   2. из ресурсов ОС (более частый вариант)
4. Занесение кода в адресное пространство и установка программного счетчика.

   Подходы:

   1. Клонирование родителя (в UNIX)

      Есть системный вызов, который создает клона

      Есть системный вызов, который полностью заменяет пользовательский контекст
   2. Из файла (в Windows)
5. Окончательное заполнение PCB
6. Изменение состояния на «готовность»

Завершение процесса

Состояние «закончил исполнение» нужно, чтобы возможно было узнать причину завершения

1. Изменение состояния на «закончил исполнение»
2. Освобождение ресурсов
3. Очистка элементов PCB
4. Сохранение в PCB информации о причинах завершения

Процесс выкидывается из системы, когда родитель
- умирает
- интересуется причинами завершение

Если родитель умер раньше ребенка, `PID=1` усыновляет ребенка

Zombie-процесс — процесс в состоянии «закончил исполнение»

Запуск процесса

- Изменение состояние на «исполнение»
- Обеспечение наличия в оперативной памяти необходимой информации
- Восстановление значения регистров
- Передача управления по адресу программного счетчика

Приостановка процесса

- Автоматическое сохранение програмного счетчика и части регистров (работа hardware)
- Передача управления по специальному адресу (работа hardware)
- Сохранение динамическойчасти
- Обработка прерывания
- Перевод процесса в состояние «Готовность»

Блокирование процесса

- Сохранение контекста процесса в PCB
- Обработка системного вызова
- Перевод процесса в состояние «ожидание»

Разблокирование процесса

- Уточнение, какое событие произошло
- Проверка наличия процесса, ожидавшего события
- Перевод ожидающего процесса в состояние «готовность»
- Обработка произошедшего события

Пример цепочки операций

Кооперация процессов

Кооперативные (взаимодействующие) процессы — процессы, которые влияют на поведения друг друга путем обмена информацией

Основные причины кооперации:
- Повышение скорости решения задач (для многоядерной системы)
- Совместное использование данных
- Модульная конструкция какой-то системы
- Для удобства работы пользователя

Взаимодействие между процессами происходят через ОС

Категории средств взаимодействия

- Сигнальные: передача бита
- Канальные: логический канал
- Разделяемая память: общее адресное пространство

Как устанавливается связь

- Нужна ли инициация?
  - Обычно нужна для канальной и разделяемой памяти и не нужна для сигнальной
- Способы адресации:
  - Прямая
    - Симметричная: и отправитель, и получатель указывают ID друг друга

  - Асимметричная: только отправитель указывает ID получателя
- Косвенная: есть вспомогательный объект для передачи

Информационная валентность процессов и средств связи

- Сколько процессов может одновременно ассоциировать с конкретным видом связи? — «It depends».
- Сколько идентичных средств связи может быть задействовано между двумя процессами? — «It depends».
- Направленность связи:
  - Симплексная связь: односторонняя
  - Полудуплексная связь: как в рации
  - Дуплексная связь: двусторонняя

2024-09-23

Особенности канальных средств связи

Буферизация

Обладает ли канал внутренней памятью

Случаи:

- Буфера нет

  Процесс-передатчик блокируется, пока процесс-получатель не считает данные

- Буфер неограниченной емкости (физически не реализуем)

  Процесс-передатчик никогда не ждет

- Буфер конечной емкости

  Самый частый случай

Модель передачи данных

- Потоковая модель
  - Операции приема/ передачи не интересуются содержанием данных и их происхождением
  - Данные не структурируются
  - Нет разделителей между записываемыми блоками
  - Можно считывать любое количество байт
- Модель сообщений
  - На данные накладывается некоторая структура
  - Отдельные сообщения явно разделены
  - Иногда сообщение хранит дополнительные данные, например, имя процесса-отправителя
  - Считывать можно только сообщение целиком

Примеры средств связи

Pipe

- Потоковая модель
- Косвенная адресация

- Читать и писать может любое число процессов
- Однонаправленный

Через pipe могут общаться только процессы-родственники, так как вход и выход в pipe не видны остальной ОС.

FIFO (именованный pipe)

Вход и выход именованы

Теперь общаться могут любые процессы, а не только родственники

—— Надежность средств связи ——

Система считается надежным:
- Нет потери информации
- Нет повреждения информации
- Нет нарушения порядка
- Не появляется лишняя информация

—— Как завершить связь? ——

Специальное действие для завершение нужно, если было нужно специальное средство для завершения

Если один и процессов больше не будет использовать средство связи, то система оповещает другой

——— Нити исполнения (threads) ———

```
Ввести массив A
Ожидания ввода A
Ввести массив B
Ожидания ввода B
Ввести массив C
Ожидания ввода C
A = A + B
C = A + C
Вывсети массив C
Ожидание вывода C
```

Процессор много простаивает — хочется распараллелить: нужен второй процесс:

Процесс 1		**Процесс 2**
Создание процесса 2		
	Переключение контекста	
		Доступ к общей памяти
		Ожидание ввода A и B
	Переключение контекста	
Доступ к общей памяти		
	TODO	

Нужны дополнительные действия:
- Породить процесс 2
- Получить разделяемую память

- Нужны переключения контекста

На одноядерной системе это неэффективно из-за доп расходов

Аналогия с железной дорогой:
- Поезд — процессор с регистрами и данными в стеке
- Стрелки — условные переходы
- Склады — данные вне стрека/ операции ввода-вывода

Если два поезда могут ехать одновременно, то получается мультипроцессорная система

Thread — каждый из поездов

В процессе могут быть несколько thread-ов:
- Общие:
  - Системный контекст (всего процесса)
  - Код
  - Данные вне стека
- Разные:
  - Регистровый контекст
  - Стек
  - Системный контекст thread-а

Для создания нового thread-а используется системный вызов

Между thread-ами есть отношение родитель-ребенок

У thread-ов тоже есть состояния, как и у процессов

Master thread (главная нить) — нить создаваемая при создании процесса

Thread-ы хороши, так как создавать их «дешевле», чем новые процессы

Thread-ы могут создавать либо на уровне библиотек, либо на уровне ядра

Thread 1	**Thread 2**
Создание нити 2	
Ввести массив A	
Ожидание ввода A	
Переключение контекста	
	Ожидание ввода A и B
Переключение контекста	
Ввести массив B	
Ожидание ввода B	
Ввести массив C	
Ожидание ввода C	
Переключение контекста	
	A = A + B
Переключение контекста	
C = A + C	
Вывести массив C	
Ожидание вывода C	

——— Алгоритмы синхронизации ———

Активность — последовательное выполнение ряда действий, направленных на достижение определенной цели

Операции внутри активности считаем атомарными (неделимыми): между операциями «отвлекаться» можно, во время операции — нельзя

Активность P из операций a b c

Активность Q из операций d e f

Последовательное выполнения PQ: a b c d e f

Псевдопараллельное выполнение (режим разделения времени): порядок операций от каждой активности фиксирован, но они могут чередоваться — интерливинг. Например, **a** d e **b c** f

Если активности зависимы, то могут быть проблемы

Недетерминированный набор — при одинаковых начальных результатах возможны разные результаты

Детерминированый набор — при одинаковых начальных данных результат всегда один

—— Условие Бернштейна ——

Достаточные условия детерминированности набор

Входные данные активности (W): объединение множеств входных данных всех активностей
Выходные данные активности (R): объединение множеств выходных данных всех активностей

Условия (для двух активностей P и Q):
- $W(P) \cap W(Q) = \emptyset$
- $W(P) \cap R(Q) = \emptyset$
- $R(P) \cap W(Q) = \emptyset$

—— Как чинить? ——

Нужно запретить «плохие» чередования

В недетерминированных наборах всегда встречается **race condition** (состояние гонки).

Mutual exclusion (взаимоисключение): если процесс захватил ресурс, то больше никто другой его не использует. Используется, когда не важно, кто первый захватил ресурс

Критические секции — участки, которые приводят к появлению race condition

Нужно сделать, чтобы критические секции выполнялись, как атомарные операции. Для этого нужно ввести «пролог» и «эпилог» для критических секций

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

Требования к алгоритмам синхронизации (от Дейкстры?):

- Алгоритм должен быть на уровне software
- Нет предположений об относительных скоростях выполнения и числе ядер

- Выполняется условие взаимоисключения (mutual exclusion) для критических участков
- Выполняется условие прогресса (progress):

  Только процессы, готовые войти в критическую секцию, принимаю решение, кто первый

  Решение не должно приниматься за конечное время
- Выполняется условие ограниченного ожидания (bound waiting):

  Каждый процесс «не пускают» не более заранее выбранного числа раз

2024-09-30

Конкретные алгоритмы (программные)

Запрет прерываний

Существует команда процесса (Clear Interrupt), которая заставляет его игнорировать почти все прерывания, кроме критических.

У процесса не могу отобрать процессор ни при каких условиях.

Существует парная команда, которая обрабатывает все запомненные и новые прерывания.

```
while (some condition) {
    запретить все прерывания
    critical section
    разрешить все прерывания
    remainder section
}
```

Если в программе есть бесконечный цикл, то вся система насмерть зависнет.

Команды доступны только ядру. Прием используется только внутри кода ОС.

Переменная-замок

Общая переменная–замок для всех процессов в наборе

```
shared int lock = 0;
while (some condition) {
    while (lock == 1);
    lock = 1;
    critical section
    lock = 0;
    remainder section
}
```

Если между `while (lock == 1);` и `lock = 1;` перейдет передача управления, то будет беда: оба процесса войдут в критическую секцию.

Строгое чередование

Введем порядок, в котором процессы будут проходить свои критические секции.

```
// для двух процессов
// i-ый процесс
shared int turn = 0;
while (some condition) {
    while (turn != i);
    critical section
```

```
    turn = (i + 1) % 2;
    remainder section
}
```

Если у процессов скорость выполнения сильно разная, то один из процессов может очень долго ждать входа в критическую секцию.

Флаги готовности

```
shared int ready[proc_num] = {0};

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

Если оба процесса скажут, что они готовы, то беда: оба зависнут в цикле.

Алгоритм Петерсона

Совмещение идей очередности и готовности.

```
// для 0-ого процесса
shared int ready[2] = {0};
shared int turn;
while (some condition) {
    ready[0] = 1;
    turn = 1;
    while (ready[1] && turn == 1);
    critical section;
    ready[0] = 0;
    remainder section;
}
```

Все пять требований выполняются.

Для n процессов алгоритм сложнее, но существует.

Bakery algorithm (алгоритм булочной)

«Алгоритм регистратуры в поликлинике»

Основные идеи:
1. Процессы можно сравнивать по именам (id-шникам)
2. Перед входом в критическую секцию процессы получают «талон» с номером. Может случиться, что номера талонов совпали.
3. В критическую секцию входит тот, у кого меньше пара (номер талона, id-шник).

Заключение

Всё работало хорошо до 2005 года, пока не появились многоядерные системы

Раньше использовалась строгая модель консистентности памяти: из ячейки всегда считывается то значение, которое последнее было туда записано.

Модель плоха с точки зрения hardware: кэши разных ядер и оперативу долго синхронизировать.

Модель ослабили: синхронизацию производять только после накопления некоторого числа изменений.

Алгоритмы синхронизации стали нерабочими: теперь после каждого изменения shared переменной нужно атомарно синхронизировать кэши.

—— Аппаратная поддержка ——

Команда Test-And-Set

```
int Test-And-Set(int* a) {
    int tmp = *a;
    *a = 1;
    return tmp;
}
// Но выполняется процессором атомарно
```

```
shared int lcok = 0;
while(some condition) {
    while (Test-And-Set(&lock));
    critical section
    lock = 0;
    remainder section
}
```

Нарушается условие ограниченного ожидания, но это легко чиниться.

Команда Swap

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

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

Тоже нарушается условие ограниченного ожидания, но это легко чиниться.

——— Механизмы синхронизации ———

Механизмы внутри ОС

Недостатки программных алгоритмов:
1. Ослабленная модель консистентности памяти
2. Перевод цикла активного ожидания (busy wait) в цикл активного ожидания[1]: «Кручение» в while-замке жрет процессорное время
3. Классические алгоритмы плохо работают в случае введения приоритетов процессов:

   Проблема, если низко приоритетный процесс вошел в критическую секцию, а у него отобрали управление:

1. высоко приоритетный не может войти в критическую секцию, так так там низко приоритетный
2. низко приоритетный не может выйти из критической секции, так как у него отняли управление

—— Семафор Дейкстры ——

`S` — семафор — целая неотрицательная разделяемая переменная

При создании инициализируется любым неотрицательным значением

Допустимые **атомарные** операции:

1. `P(S)`:

   ```
   while (S == 0) block process
   S -= 1
   ```

2. `V(S)`:

   ```
   S += 1
   ```

«Задача об обедающих философах»

—— Проблема Producer–Consumer ——

Два процесса (в более сложной постановке процессов может быть больше):
1. Один производит информацию
2. Один — потребляет

Обмениваются информацией через буфер конченого размера:
1. Если в буфере нет места, то Producer блокируется
2. Если в буфере пусто, то Consumer блокируется

Критическая секция — работа с буфером

Решения с тремя семафорами

Семафоры:
1. Взаимоисключение работы буферов (`mut_ex`)
2. Блокировка Producer (`full`)
3. Блокировка Consumer (`empty`)

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

```
Producer:                Consumer:
while (1) {              while (1) {
    produce_item();          P(full)
    P(empty);                P(mut_ex)
    P(mut_ex);               get_item();
    put_item();              V(mut_ex);
    V(mut_ex):               V(empty);
    V(full);                 consume_item();
}                        }
```

[1] это не всегда эффективно из-за накладных расходов

Вдруг совершили ошибку: в `Consumere` перепутали местами строки `P(full)` и `P(mut_ex)`. Всё ломается: заходим в состояние вечного ожидания. Эту ошибку сложно отследить, так как она возникает только при определенных условиях.

—— Мониторы Хора (Hoare) ——

В ЯП встраиваются определенные конструкции — мониторы Хора.

```
Monitor monitor_name {
    Описание внутренних переменных;
    void m1(...) {...}
    void m2(...) {...}
    ...
    void mn(...) {...}
    Блок инициализации переменных;
}
```

Со внутренними переменными можно работать только используя методы монитора.

Только один метод монитора может быть вызван (это достигается ОС и компилятором языка).

Условные переменные

```
Condition C;
```

Всегда лежат внутри монитора.

Операции:

1. `C.wait`: Всегда блокирует данный процесс
2. `C.signal`: Разблокирует один процесс, который раньше выполнил `.wait`, если он есть. Процесс мгновенно вылетает из монитора.

Producer–Consumer

```
Monitor PC {
    Condition full, empty;
    int count;

    void put() {
        if (count == N) full.wait;
        put_item();
        ++count;
        if (count == 1) empty.signal; // Если ждал consumer, то разбудили его
    }

    void get() {
        if (count == 0) empty.wait;
        get_item();
        --count;
        if (count == N - 1) full.signal; // Если ждал producer, то разбудили его
    }

    { count = 0; }
}
```

```
Producer:           Consumer:
while (1) {         while (1) {
    produce_item();     PC.get();
```

```
    PC.put();           consume_item();
}                   }
```

В этом методе хорошо то, что сложно налажать. Плохо, что нужен ЯП с соответствующей конструкцией.