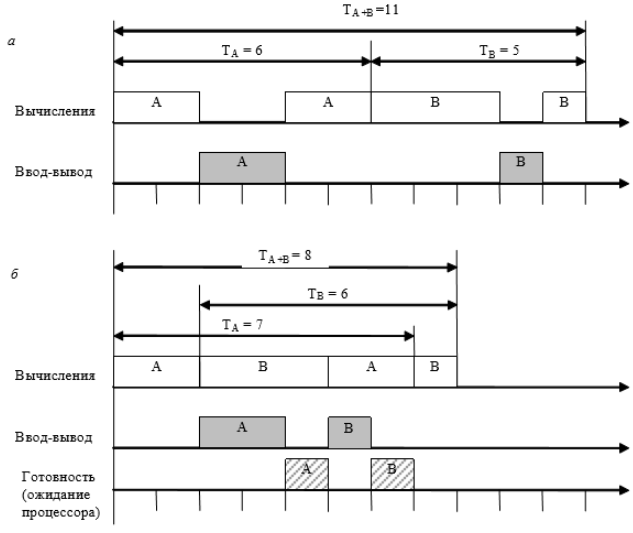
Процессы

- Процесс — абстрактное понятие, описывающее выполнение программы процессором
- Процесс — выполняемая программа, включая значения регистров, адресное пространство, стек, открытые файлы и прочие ресурсы
- Многозадачность — возможность одновременного использования различных ресурсов машины (процессор, память, устройства ввода/вывода) разными приложениями
- В многопроцессорных (многоядерных) системах выполнение процессов действительно происходит одновременно
- В системах с одним процессором для реализации многозадачности в памяти должны находиться несколько процессов. В определённые моменты времени происходит переключение между ними, т.е. выбор одного из процессов и его выполнение

Многозадачность



Создание процесса

- При инициализации системы процесс создаётся загрузчиком путём загрузки кода (части операционной системы) и передачи ему управления
- Последующее создание процессов происходит с помощью системных вызовов, например, при:
  - вводе команды в консоли (командой строке);
  - двойном щелчке на иконке;
  - непосредственном использовании системного вызова в коде.
- Системный вызов **fork** — создание процесса, аналогичного родительскому
- Системный вызов **execv (execl)** — загрузка новой программы в память

Завершение процесса

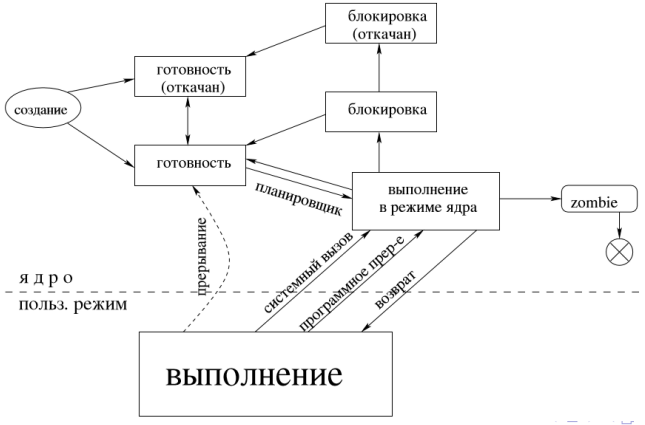
- Способы завершения процесса
  - Нормальное завершение, добровольное
  - Завершение вследствие ошибки, добровольное
  - Завершение вследствие фатальной ошибки, принудительное
  - Уничтожение другим процессом, принудительное
- Системные вызовы **exit**, **kill**

Состояние процесса

- Действие (процесс использует процессор)
- Готовность (процесс временно приостановлен, чтобы позволить выполняться другому процессу)
- Блокировка (процесс не может быть запущен прежде, чем произойдет какое-либо внешнее событие)
- Переход 1 осуществляется при выполнении операции ввода/вывода или при системном вызове (**sleep**)
- Переход 4 осуществляется при возникновении внешнего события (окончание операции ввода/вывода, системный вызов **wakeup**)
- Переходы 2, 3 выполняет планировщик — специальная подсистема внутри ОС

Жизненный цикл процесса

- В режиме ядра исполняется только код ядра, не процесса
- Zombie — завершившийся процесс, для которого процесс-родитель не может принять сигнал о завершении процесса)



Реализация процессов

- Таблица процессов — каждый элемент таблицы описывает отдельный процесс
- **MINIX3** — три раздела в таблице процессов для разных целей

Ядро	Управление процессами	Управление файлами
Регистры	Указатель на текстовый сегмент	Маска UMASK
Счетчик команд	Указатель на сегмент данных	Корневой каталог
Слово состояния программы	Указатель на сегмент стека	Рабочий каталог
Указатель стека	Статус завершения	Дескрипторы файлов
Состояние процесса	Состояние сигналов	Реальный идентификатор пользователя
Текущий приоритет	Идентификатор процесса	Эффективный идентификатор пользователя
Максимальный приоритет	Родительский процесс	Реальный идентификатор группы
Оставшееся число тактов	Группа процессов	Эффективный идентификатор группы
Размер кванта	Процессорное время дочернего процесса	Управление
Использованное процессорное время	Реальный идентификатор пользователя (UID)	Область сохранения для чтения/записи
Указатели очереди сообщений	Эффективный идентификатор пользователя	Параметры системного вызова
Биты действующих сигналов	Реальный идентификатор группы (GUID)	Различные битовые флаги
Различные флаги	Эффективный идентификатор группы	
Имя процесса	Файловая информация о совместном использовании текста	
	Битовый маски для сигналов	
	Различные битовые флаги	
	Имя процесса	

Переключение процессов

- Дескриптор процесса — информационная структура, в которой представлена внешняя (по отношению к процессу) информация
  - Идентификатор процесса
  - Состояние процесса
  - Информация о привилегиях процесса
- Контекст процесса — содержит информацию о внутреннем состоянии процесса и аппаратуры
  - Содержимое регистров
  - Информация об открытых файлах
  - Информация о работе внешних устройств
- Аппаратный контекст — набор данных, который должен быть загружен в регистры до возобновления работы процесса
- Linux
  - Переключение глобального каталога страниц с целью установки адресного пространства
  - Переключение стека режима ядра и аппаратного контекста
    - В ранних версиях: аппаратно, far jmp (переход на селектор дескриптора сегмента нового процесса, автоматически сохраняется старый аппаратный контекст и загружается новый из сегмента состояния задачи TSS
    - В версии 2.6: пошаговое переключение выполняемое с помощью последовательности инструкций mov, обеспечивается строгий контроль допустимости загружаемых данных
    - Время переключения приблизительно одинаково
  - Сохранение и загрузка регистров FPU, MMX, XMM

Потоки

- Переключение между процессами очень длительная задача
- Традиционно у каждого процесса есть адресное пространство и один поток управления
- Возможно добавление в процесс нескольких потоков управления
- Такие потоки управления называются программными потоками или легковесными процессами

Процесс	Поток
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Действующие аварийные сигналы	
Сигналы и их обработчики	
Учетная информация	

Межпроцессное взаимодействие

- IPC — InterProcess Coomunication
- Аспекты проблемы
  - Передача информации между процессами
  - Контроль над деятельностью процессов (гарантия, что процессы не пересекутся в критических ситуация)
  - Согласование действий процессов
- Для обмена информацией между процессам можно использовать общую память, файлы и т.п.
- Необходимо предусматривать взаимное исключение — запрет одновременной записи и чтения совместно используемых данных более чем одним процессом (если один процесс использует общие данные, другому процессу это запрещено)
- Критическая область (секция) — часть программы, в которой есть обращение к совместно используемым данным
- Задача: избежать нахождение двух процессов в критических областях

Взаимное исключение с активным ожиданием

- Запрещение прерываний (при входе в критическую область)
- Добавление переменных блокировки: возникает проблема атомарной проверки и изменения такой переменной
- Строгое чередование (переменная отслеживает, чья очередь войти в критическую область)

```
/* Процесс 1 */
while(TRUE){
    while(turn!=0)
    critical_region();
    turn=1;
    noncritical_region();
}
```

```
/* Процесс 2 */
while(TRUE){
    while(turn!=1)
    critical_region();
    turn=0;
    noncritical_region();
}
```

Алгоритм Петерсона

- 1981, Гарри Петерсон существенно улучшил программный алгоритм контроля входа в критическую секцию
- Без строгого чередования

```
#define FALSE 0
#define TRUE 1
#define N 2                                                 // Количество процессов
int turn;                                                   // Чья очередь
int interested[N] = {0};                                    // Индикатор "интереса" критической областью

void enter_region(int process)
{
    int other;
    other = 1 - process;                                    // Номер другого процесса
    interested[process] = TRUE;
    turn = process;                                         // Занимаем очередь
    while (turn == process && interested[other] == TRUE);   // Пустой цикл
    /* вход в критическую секцию осуществлён */
}

void leave_region(int process)                              // Покидание критической секции
{
    interested[process] = FALSE;
}
```

Алгоритм Петерсона, последовательный вход в секцию

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N] = {0};

void enter_region(int process)
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

Время	Поток 0	Поток 0
t_1	int other = 1;	
t_2	interested[0] = TRUE;	
t_3	turn = 0;	
t_4	while (turn == 0 && interested[1]);	
t_5	Критическая секция	int other = 0;
t_6		interested[1] = TRUE
t_7		turn = 1;
t_8		while (turn == 1 && interested[0]);
t_9		while (turn == 1 && interested[0]);
t_{10}	interested[0] = FALSE	
t_{11}		Критическая секция
t_{12}		
t_{13}		interested[1] = FALSE

Алгоритм Петерсона, попытка одновременного входа в секцию

```
#define FALSE 0
#define TRUE 1
#define N 2
int turn;
int interested[N] = {0};

void enter_region(int process)
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE);
}

void leave_region(int process)
{
    interested[process] = FALSE;
}
```

Время	Поток 0	Поток 0
t_1	int other = 1;	
t_2		int other = 0;
t_3		interested[1] = TRUE;
t_4	interested[0] = TRUE;	
t_5	turn = 0;	
t_6		turn = 1;
t_7		while (turn == 1 && interested[0]);
t_8		while (turn == 1 && interested[0]);
t_9	while (turn == 0 && interested[1]);	
t_{10}	Критическая секция	while (turn == 1 && interested[0]);
t_{11}		while (turn == 1 && interested[0]);
t_{12}		while (turn == 1 && interested[0]);
t_{13}	interested[0] = FALSE;	
t_{14}		Критическая секция
t_{15}		
t_{16}		interested[1] = FALSE

Аппаратная реализация взаимного исключения

- Реализуется с помощью команды **TSL** (Test and Set Lock)
- В регистр считывается значение ячейки памяти, а в ячейке сохраняется ненулевое число, регистр и адрес ячейки передаются в качестве параметров инструкции
- Атомарная операция, в время её выполнения не может произойти переключения контекста
- Пример:
  - **REGISTER** — имя регистра для хранения промежуточного значения
  - **LOCK** — ячейка памяти, общая для процессов, желающих войти в критическую секцию

```
enter_region;
    TSL REGISTER, LOCK
    CMP REGISTER, #0
    JNE enter_region        // Если в ячейке было ненулевое значение,
                            // значит какой-то процесс заблокировал вход в критическую секцию
    RET

leave_region:
    MOVE LOCK, #0
    RET
```

Аппаратная реализация взаимного исключения - спинлок

- Spinlock (циклическая блокировка) — низкоуровневый примитив синхронизации, применяемый для реализации взаимного исключения
- Представляет собой переменную в памяти и реализуется на атомарных операциях, которые должны присутствовать в системе команд процессора
- Пример для x86: инструкция xchg производит обмен содержимого двух операндов

```
mov eax, spinlock_address
mov ebx, SPINLOCK_BUSY

wait_cycle:
xchg [eax], ebx
cpm ebx, SPINLOCK_FREE
jnz wait_cycle

; < начало критической секции

mov eax, spinlock_address
mov ebx, SPINLOCK_FREE
xchg [eax], ebx
```

Семафоры

- Алгоритмы активного ожидания:
  - расходуют ресурсы процессора
  - возможна инверсия приоритетов: задача с высоким приоритетом выполнения ждет освобождения критической секции у задачи с низким приоритетом
- Существуют примитивы, блокирующие процесс в случае запрета входа в критическую секцию
- **sleep** — системный вызов, блокирующий процесс
- **wakeup** — пробудить процесс
- 1965 год: Дейкстра предложил обобщение этих примитивов, предложив новый тип переменной «семафор»
- Значение семафора может быть нулем или положительным числом
- Над семафорами возможны две операции:
  - **down**: сравнивает значение семафора с нулем. Если значение больше нуля, оно уменьшается на единицу, иначе процесс блокируется
  - **up**: увеличивает значение семафора. Если с этим семафором связаны заблокированные (ожидающие) процессы, происходит пробуждение одного из них
- Операции **down** и **up** атомарны

Мьютексы

- Мьютексы — упрощенная версия семафора
- Без подсчета, только для управления взаимными исключениями
- Мьютекс — переменная, которая может находиться в одном из двух состояний: заблокирована / свободна
- Над мьютексами возможны две операции:
  - **mutex_lock**: проверяет, заблокирован ли мьютекс и блокирует его, иначе процесс «засыпает»
  - **mutex_unlock**: разблокирует мьютекс и передает управление заблокированному процессу, связанному с этим мьютексом (если такой есть)
- Операции **mutex_lock** и **mutex_unlock** атомарны
- Используется для управления взаимными исключениями в потоках

Мониторы

- Использование примитивов — сложный процесс, возможно появление трудно обнаружимых ошибок
- Монитор — примитив синхронизации высокого уровня
- Монитор — набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет
- Процессы могут вызывать процедуры монитора, но у процедур вне монитора нет доступа ко внутренним структурам данных монитора
- Свойство монитора: при обращении к монитору активным может быть только один процесс
- Обычно при вызове монитора первые несколько команд проверяют, нет ли в мониторе активных процессов

- Бринч Хансен, Чарльз Хоар

```
monitor account {
    int balance := 0

    function withdraw(int amount){
        if amount <0 then error "Отрицательный счет"
        else if balance < amount the error "Недостаточно средств"
        else balance := balance - amount
    }
    function deposit(int amount){
        if amount < 0 then error "Отрицательная сумма"
        else balance := balance + amount
    }
}
```

Передача сообщений

- Рассмотренные примитивы не применимы в распределенных системах
- Вариант: передача сообщений
  - send (destination, message): посылает сообщение приемнику
  - recieve(source, message): получает сообщение от источника
- Проблемы:
  - Подтверждение приема
  - Аутентификация
  - Производительность

Барьеры

- Барьер — метод синхронизации в распределённых вычислениях, при котором выполнение параллельного алгоритма или его части можно разделить на несколько этапов, разделённых барьерами
- Идея заключается в том, чтобы в определенной точке ожидания собралось заданное число потоков управления
- Функции
  - Инициализация и удаление (разрушение) барьеров
  - Синхронизация на барьере
  - Инициализация и удаление (разрушение) атрибутных объектов барьеров
  - Опрос и установка атрибутов барьеров
- Когда у барьера собирается заданное число потоков, одному из них передается именованная константа, а другим — нули
- После этого барьер возвращается в начальное (инициализированное) состояние, а выделенный поток может выполнить соответствующие объединительные действия.