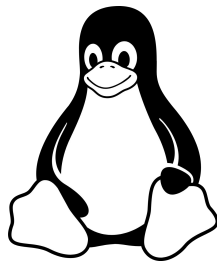
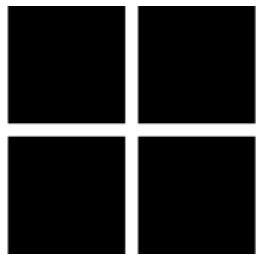
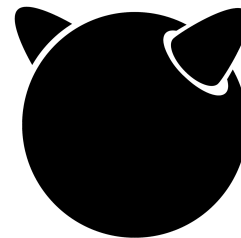


Operating systems



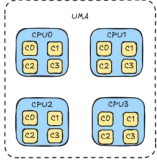
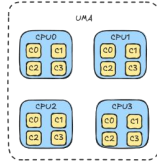
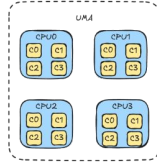
macOS



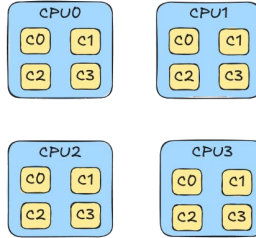
МНОГОПОТОЧНОСТЬ

GRID

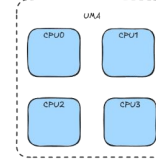
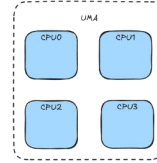
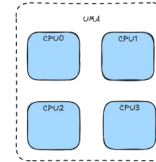
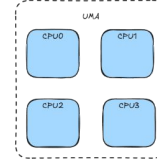
CLUSTER



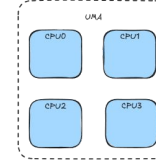
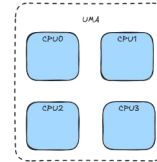
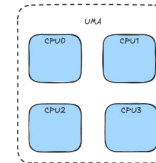
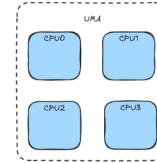
UMA



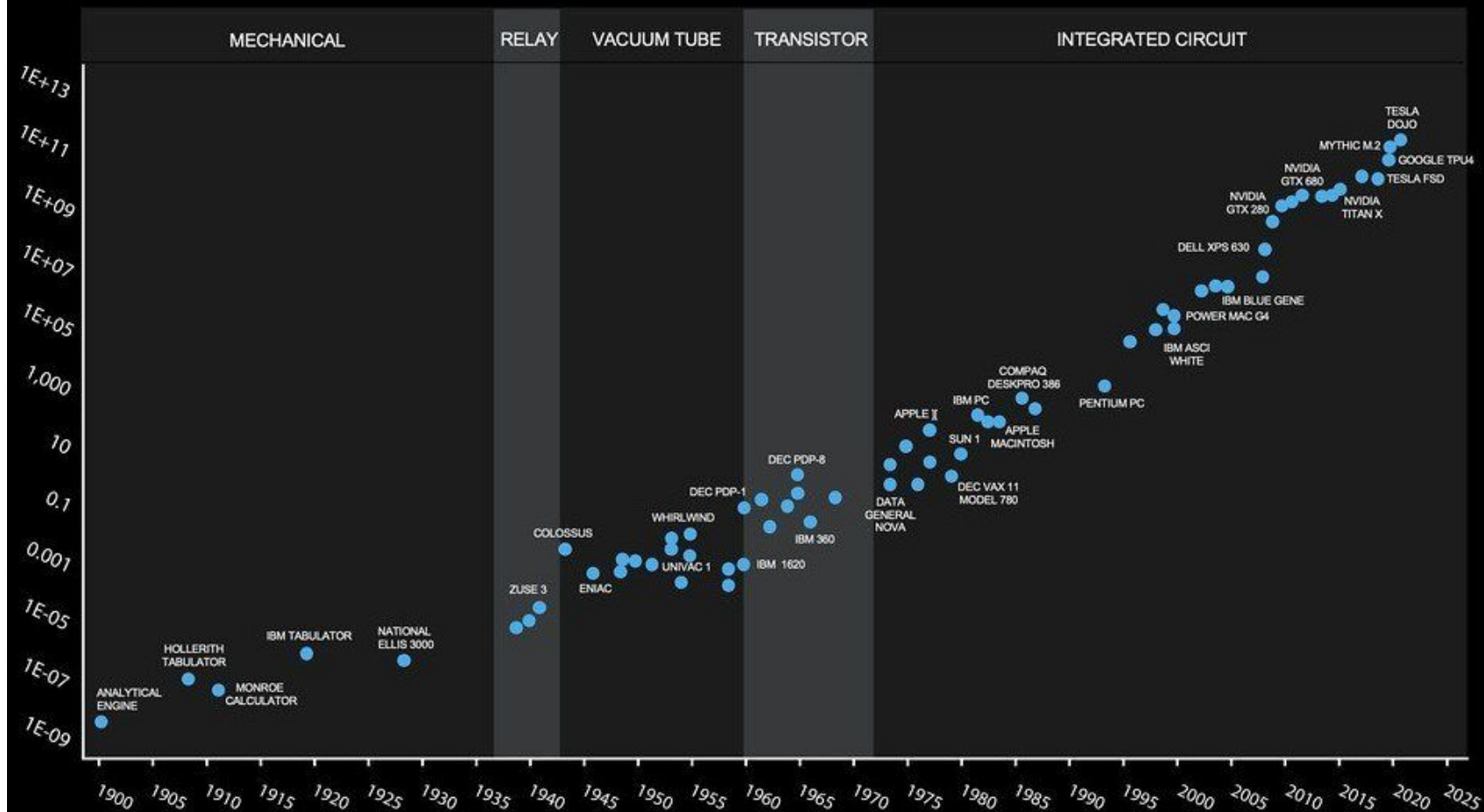
CLUSTER



CLUSTER



122 YEARS OF MOORE'S LAW



Классификация Флинна

- **SISD** (Single Instruction, Single Data) – одноядерные системы; параллельности нет
- **SIMD** (Single Instruction, Multiple Data) – векторные и графические процессоры; параллельная обработка данных (обработка изображений)
- **MISD** (Multiple Instruction, Single Data) – отказоустойчивые системы; несколько потоков команд выполняются над одним потоком данных; редко встречается, экзотика
- **MIMD** (Multiple Instruction, Multiple Data) – многоядерные процессоры, кластеры, суперкомпьютеры; полный, честный параллелизм; наиболее универсальные и распространенные архитектуры

SIMD

Процессоры предоставляют возможность обрабатывать группы данных (векторы) за “один” такт процессора:

- Векторные расширения x86_64: **MMX**, **SSE1/2/3/4**, **AVX1/2**, **AVX-512**
- Векторные расширения ARM: **Neon**

Используете специальные инструкции процессора, код становится не кроссплатформенным, компилятор редко видит возможность их использования, поэтому программисты пишут SIMD код самостоятельно

[Пример с применением функции MADD \(multiply-add\) для большого потока данных](#)

MIMD

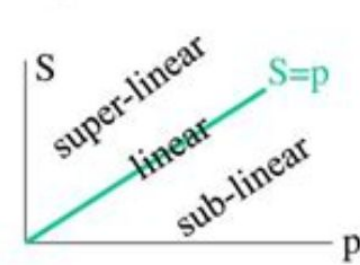
Процессоры предоставляют возможность параллельного выполнения программ с помощью средств операционной системы:

- Процессы
 - преимущество в простоте написания кода
 - могут исполняться на множестве машин
 - операционная система использует защищенную модель памяти, что предотвращает доступ к памяти других процессов
 - более устойчивы к ошибкам, более надежны
- Потоков: преимущество в **быстродействии!!!**

Говоря про быстродействие...

Мы будем пользоваться следующими метриками:

- **Ускорение:** $S = T_s / T_p$, где
 - T_s – время последовательной реализации,
 - T_p – время параллельной реализации,
 - $1 \leq S \leq p$,
 - p – количество ядер



- **Эффективность:** $X = S / p$, где $X < 1$

Важный закон в параллельном программировании

Закон Амдала:

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

- α – доля последовательного кода ($0 < \alpha \leq 1$)
- p – количество ядер

Показывает верхнюю оценку ускорения: чем больше последовательных участков в коде будет организовано, тем меньше потенциальное ускорение.

Потоки

Threads (нити, треды, легковесные процессы) — потоки исполнения в общем адресном пространстве. Способ выполнения множества программ в одном процессе.

Они полезны, потому что:

- можно одновременно преобразовывать единые данные разными способами (функциями)
- создание потоков быстрее, чем создание процесса
- наиболее эффективны в мультипроцессорных системах

Свойства потоков

- У каждого процесса при старте есть единственный поток управления (*main*)
- Потоки могут создавать новые в пределах одного процесса
- Все потоки в пределах одного процесса имеют общие сегменты данных и кода (находятся в одном адресном пространстве процесса)
- Каждый поток имеет свой стек

Единицей планирования в ядре ОС является именно **поток**!

Отличия потоков от процессов

Процесс	Поток
<ul style="list-style-type: none">• Данные для управления памятью процесса• Учетная информация (им, пользователь, время, ID и пр.)• Информация об используемых процессом объектах OS• Исполняемый код и данные в адресном пространстве• Таблица потоков	<ul style="list-style-type: none">• Состояние процесса• Указатель инструкции / программный счетчик• Содержание регистров• Данные для планирования использования CPU• Стек

Стратегии реализации потоков

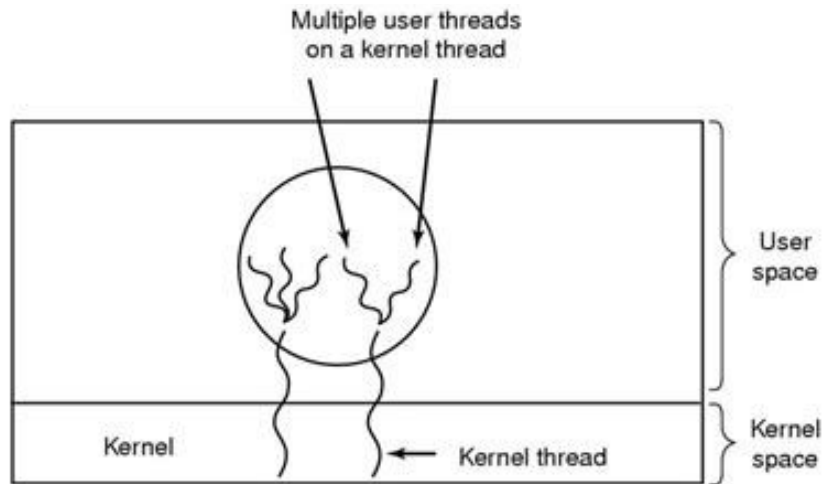
- Потоки уровня ядра (KLT):
 - Потоки полностью управляются ядром ОС и имеют свою запись в таблице потоков ядра
 - *Высокая степень параллелизма и устойчивость :)*
 - Высокие накладные расходы на переключение контекста, так как ядро участвует в каждом переключении :(
- Потоки уровня пользователя (ULT):
 - Потоки управляются библиотекой в пользовательском пространстве, ядро не знает о них, и переключение потоков происходит без обращения к ядру
 - Высокая производительность и гибкость в реализации :)
 - Блокировка одного потока может заблокировать весь процесс :(

Эффективная реализация лежит где-то по-середине...

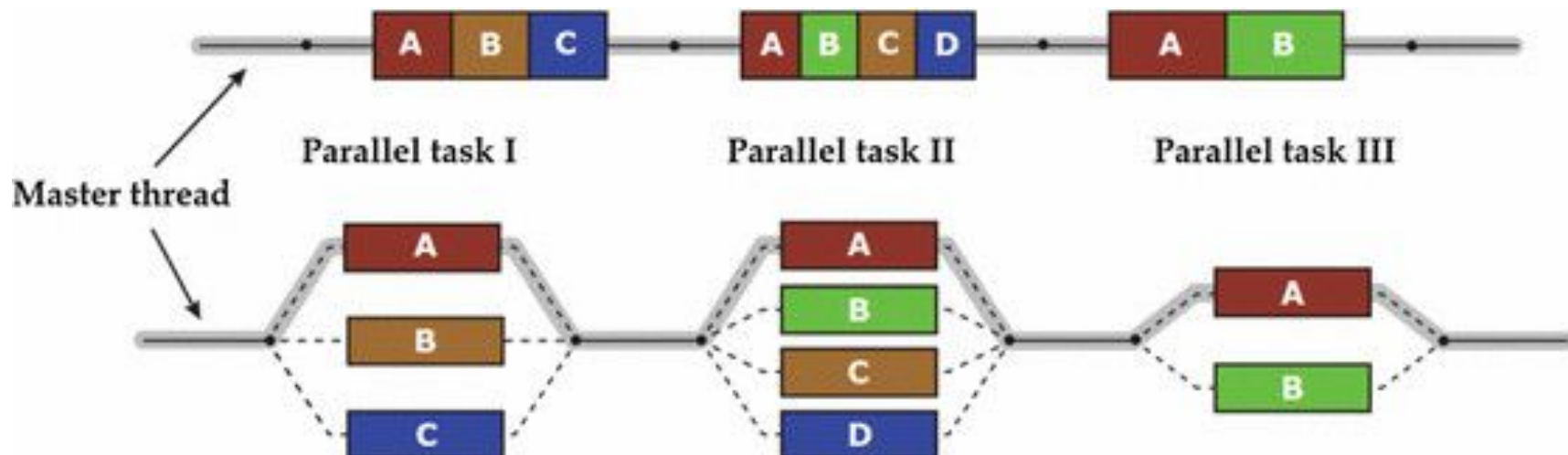
Стратегии реализации потоков

Гибридная модель

- Пользовательские потоки мультиплексируются на меньшее количество ядерных потоков
- Соблюдается баланс между производительностью и параллелизмом, но сложны в реализации



Fork & Join модель



Псевдопараллелизм

В Fork & Join модели **нет ограничения по количеству** создаваемых потоков.

Если создать потоков больше, чем логических процессоров в системе, то вы будете наблюдать **псевдопараллелизм** – ситуация, когда задачи будут исполнены на ядрах количестве больше одного.

Поскольку ядру ОС придётся планировать больше потоков, соответственно ядро будет испытывать **накладные расходы (overhead)** на переключение контекста исполнения и истинное ускорение вы не увидите.

А увидите даже замедление, если overhead больше чем выполненная работа

Ещё больше накладок

В Fork & Join модели подразумевается, что выполнение параллельных заданий должно сопровождаться **созданием, ожиданием и уничтожением** потоков. **И так каждый раз, когда нужно выполнить задание!**

Создание, ожидание и удаление – дополнительные накладки по эффективности вашей программы. С ними нужно бороться!

То есть Fork & Join модель **не позволяет** выделить группу потоков и менять им задание как только они отработали. Есть более эффективный механизм, но Fork & Join прост в реализации, поэтому будем пользоваться им

Примитивы синхронизации

Это механизмы в программировании, предоставляемые средствами ядра ОС и библиотекой, которые используются для управления доступом к **общим ресурсам**, в которые одновременно **пишут и читают**, в параллельных системах, чтобы избежать **гонки данных** и обеспечить **монопольный** доступ **одному** потоку.

Общими ресурсами могут быть память, файлы, базы данных и т.п. куда производятся и запись, и чтение одновременно.

Гонка данных – это ошибка :(

Два или более потока одновременно могут обратиться к одной и той же области памяти, и хотя бы один из них выполнит запись. Это приведет к непредсказуемым результатам, так как порядок операций не определён.

Порядок операций не определён, потому что потоки выполняются параллельно и независимо друг от друга. Кто-то выполняется быстрее, кто-то медленнее и эта скорость выполнения постоянно меняется

Программирование между строк

Посмотрим на простейший пример гонки данных – инкремент числа:

```
increment(volatile int *x):
```

```
    mov     eax, DWORD PTR [rdi] ; Загружаем 'x' из памяти в регистр
    inc     eax                  ; Инкрементируем
    mov     DWORD PTR [rdi], eax ; Загружаем новый 'x' обратно в память
```

Первый поток

```
mov: eax = x = 0
inc:  eax = 1
mov:  x = eax = 1
```

**Мы хотим
получить
x = 2, но...**

Второй поток

```
mov: eax = x = 0
inc:  eax = 1
mov:  x = eax = 1
```

Такое **может** произойти, а может и нет. Если ***когда-то может***, то это нужно исправлять!

Mutex, как простое решение проблемы!

Мьютекс – простейший примитив синхронизации, используемый для предоставления монопольного доступа к общему ресурсу.

- Имеет два состояния: разблокированный и заблокированный, т.е. похож на **bool**
- Имеет простой интерфейс: **lock** и **unlock**
- Оборачивайте куски кода в lock и unlock и проблема “решена”

Если поток пытается заблокировать разблокированный mutex, то поток блокирует mutex и имеет монопольный доступ к ресурсу, который он охраняет

Если поток пытается заблокировать заблокированный mutex, то поток будет спать, пока mutex вдруг не станет разблокированным

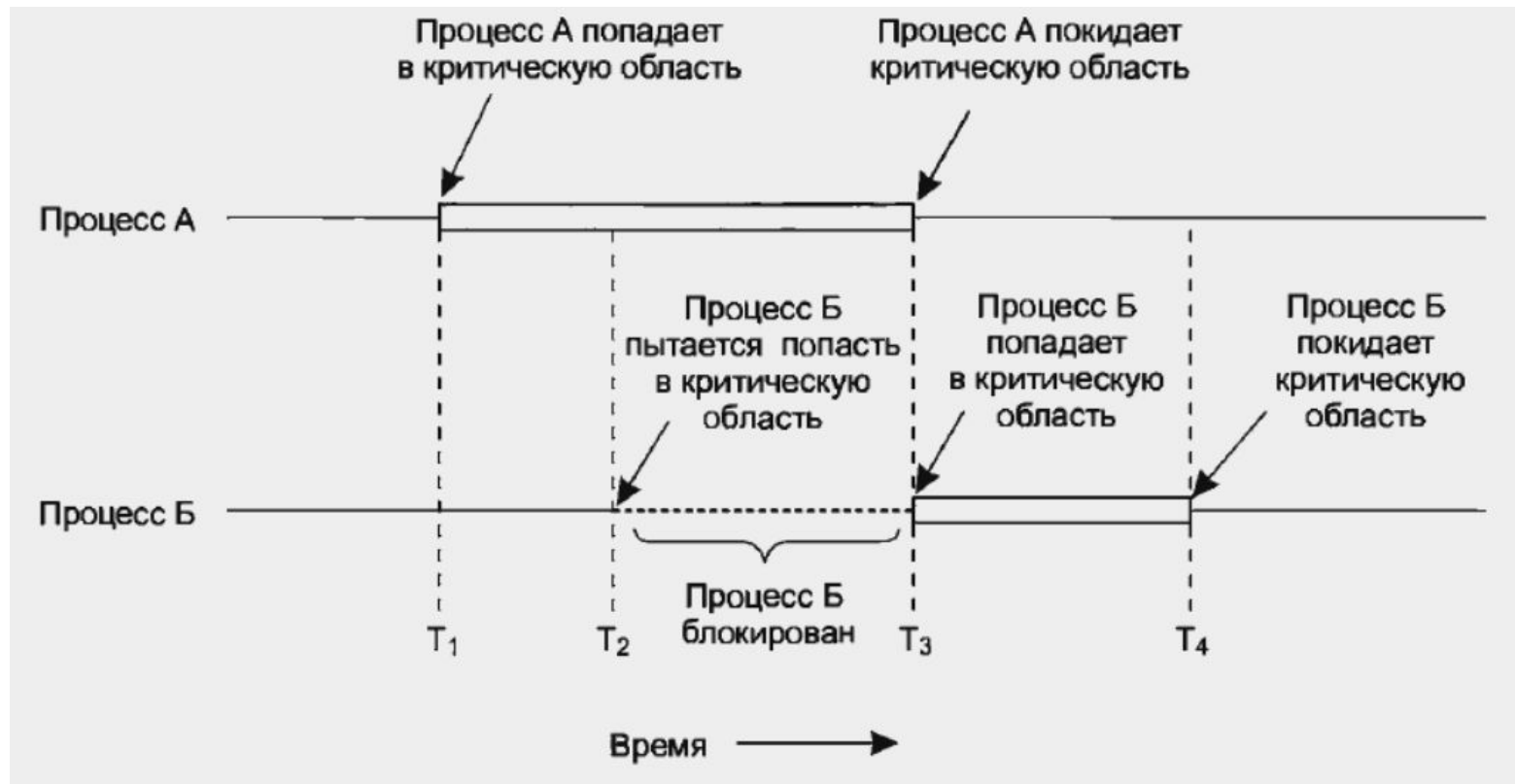
Mutex – простое “решение” проблемы

На самом деле, каждый раз, когда мы оборачиваем куски кода в lock и unlock, мы организовываем больше последовательных участков.

Это значит, что злоупотребление примитивами синхронизации снижает потенциальное ускорение по закону Амдала!

Такие гарантированно последовательные участки кода, где организован монопольный доступ к ресурсам (захват, пользование, освобождение), называются ***критическими секциями***.

Критические секции



Пытайтесь делать таких участков меньше!

Пытайтесь делать так, чтобы каждый поток работал только со своей копией данных! Таким образом, отпадает нужда в синхронизации.

Если потоки модифицируют свои области общих входных данных, то старайтесь записывать результат в свои, для каждого потока, массивы, которые вы потом объедините!

Но если архитектура кода и задача так сделать не позволяет, то используйте примитивы синхронизации.

Я рекомендую подумать над организацией памяти для потоков при решении задачи с помощью параллельного программирования, прежде чем везде лепить мьютексы!

Попробуем написать mutex сами!

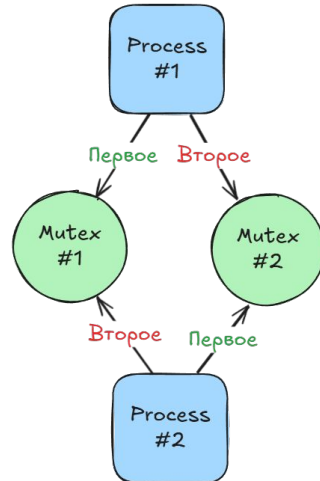
```
struct Mutex {  
    bool flag[2];  
};  
  
void lock(Mutex *mutex) {  
    int i = getpid(); // Для этого примера считаем, что  
    int j = 1 - i;    // потоков всего может быть 2 на процесс  
    mutex->flag[i] = true;  
    while (flag[j]);  // Просто ждём, ничего не делаем  
}  
  
void unlock(Mutex *mutex) {  
    int i = getpid();  
    flag[i] = false;  
}
```

Тут, кажется, есть проблема...

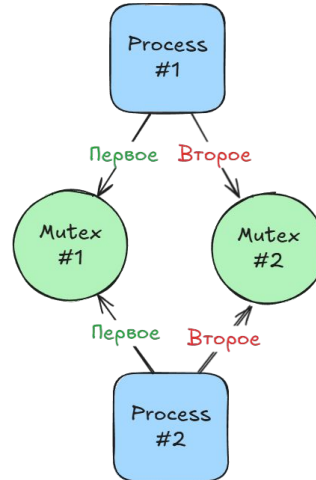
Взаимоблокировка (deadlock)

Такая ситуация в многопоточных программах, когда два или более потока бесконечно ждут друг друга, чтобы освободить ресурсы, которые они удерживают, и не могут продолжить выполнение.

ТУТ ВСЁ СЛОМАЛОСЬ :(



ТУТ ВСЁ ХОРОШО :)



Мораль истории

Делайте захват и освобождение ресурсов в одном порядке в коде, который будет исполнен множеством потоков

Такие ошибки можно исправлять следующими способами:

- Пройгнорить (не надо)
- Упорядочить захват ресурсов (надо)
- Использовать таймауты при блокировке (лениво)
- Применять алгоритмы избежания взаимоблокировок; например, алгоритм банкира (overkill)

Усвоили урок, перепишем реализацию mutex

```
struct Mutex {  
    int victim;  
};  
  
void lock(Mutex *mutex) {  
    int i = getpid(); // Для этого примера считаем, что  
    mutex->victim = i; // потоков всего может быть 2 на процесс  
    while (mutex->victim == i); // Просто ждём, ничего не делаем  
}  
  
void unlock(Mutex *mutex) {}
```

Взаимоблокировки тут не будет, а что не так?..

Алгоритм Петерсона

```
struct Mutex {  
    bool flag[2];  
    int victim;  
};  
  
void lock(Mutex *mutex) {  
    int i = getpid(); // Для этого примера считаем, что  
    int j = 1 - i;    // потоков всего может быть 2 на процесс  
    mutex->flag[i] = true;  
    victim = i;  
    while (flag[j] && victim == i); // Просто ждём, ничего не делаем  
}  
  
void unlock(Mutex *mutex) {  
    int i = getpid();  
    flag[i] = false;  
}
```

Семафоры

Примитив синхронизации, используемый для управления доступом к ограниченному количеству ресурсов или координации потоков.

- Имеет в себе счётчик, начальное значение которого можно задать при создании семафора, т.е. похож на **int**
- Имеет простой интерфейс: **wait** и **post**
- Применяется, если с ресурсом можно работать параллельно больше, чем одному потоку

Захват семафора (**wait**) декрементирует внутренний счетчик, а освобождение – инкрементирует.

Поток будет ждать при захвате, пока счётчик не станет больше 0.

Спины!

Spin-mutex – примитив синхронизации с **активным ожиданием**, который в точности повторяет поведение обычного мьютекса. Но есть отличия:

- Spin-mutex является **великолепным `while(true)`**
- Потоки, которые пытаются заблокировать такой мьютекс, если его уже захватили, будут **висеть в бесконечном цикле**, вместо того чтобы спать
- Спин-мьютекс реализован стандартной библиотекой **в пользовательском коде**, в отличие от обычного мьютекса, где логика работы реализована в ядре (syscall)
- Его использование оправдано в коде, если он защищает участок, который не имеет системных вызовов и занимает мало процессорного времени

Реализация spin-mutex

```
struct SpinMutex {
    volatile bool flag;
};

extern void lock_spin_mutex(SpinMutex *spin);

void unlock_spin_mutex(SpinMutex *spin) {
    spin->flag = false;
}

;;; spin_mutex.asm
Spin_mutex_lock: ; Аргумент-указатель на SpinMutex находится в регистре rdi
    mov al, 0      ; Ожидаем увидеть во флаге 0
    mov bl, 1      ; Если увидели 0, то флаг должен стать 1
try_lock:
    lock cmpxchg [rdi], bl ; Атомарно сравнить mutex->flag с al и установить bl
    je lock_acquired      ; Если обмен успешен (ZF=1), мьютекс захвачен
    pause                 ; Инструкция PAUSE для снижения нагрузки
    jmp try_lock          ; Повторить попытку
lock_acquired:
    ret                  ; Возврат из функции
```

Атомики!

Атомарные операции – это низкоуровневые операции, выполняемые за один такт процессора без прерывания другими потоками.

- Используют аппаратные инструкции процессора и зависят от его архитектуры для атомарного выполнения
- Поддерживают операции чтения, записи, инкремента, сравнения-и-обмена (compare-and-swap)
- Используются для простых операций (например, счётчики, флаги).
- Атомарные операции **не бесплатны** – процессор **приостановит** все остальные ядра, пока одно ядро выполняет атомарную операцию!

Condition variable

Условная переменная – примитив синхронизации для координации потоков, позволяющий им ожидать выполнения определённого условия

- Имеет простой интерфейс: **wait** и **broadcast**
- **Связан с другим примитивом синхронизации** (обычно мьютекс), поэтому cond. var. **является просто сигналом для другого ПС**
- Вы сами пишете **условие (if-else)** на языке программирования, когда потоки должны подождать (**wait**) и потом проснуться (**broadcast**)

Condition variable

- Поток вызывает **wait**, чтобы приостановиться и ждать сигнала (спать), если условие не выполнено. При этом он освобождает связанный примитив синхронизации
- Другой поток, изменивший условие, вызывает **broadcast**, чтобы возобновить один или все ждущие потоки. При это все ждущие потоки захватывают связанный примитив синхронизации

Отличия condition variable от semaphore

Condition variable

- Ожидание выполнения условия, связанного с общим ресурсом
- **broadcast** освобождает **все** потоки
- Требуется примитив синхронизации для защиты условия и критической секции
- Поток ждёт сигнала **broadcast**, если ранее заснул на **wait**

Semaphore

- Управление доступом к ограниченному количеству ресурсов
- **post** освобождает место для работы **одного** потока
- Является самостоятельным примитивом синхронизации
- **wait** инкрементирует внутренний счетчик, **post** – декрементирует

Практический пример: барьер потоков

```
typedef struct {
    pthread_mutex_t mutex; // Мьютекс для защиты данных барьера
    pthread_cond_t cond;   // Условная переменная для ожидания
    int count;             // Текущий счётчик достигших потоков
    int total;             // Общее количество потоков для барьера
} Barrier;

void barrier_wait(Barrier *barrier) {
    pthread_mutex_lock(&barrier->mutex);
    barrier->count++;

    if (barrier->count < barrier->total) {
        // Поток ждёт, пока все не достигнут барьера
        pthread_cond_wait(&barrier->cond, &barrier->mutex);
    } else {
        // Последний поток уведомляет все ждущие
        barrier->count = 0; // Сбрасываем счётчик для следующего использования
        pthread_cond_broadcast(&barrier->cond);
    }

    pthread_mutex_unlock(&barrier->mutex);
}
```

Барьер потоков

