Embedded OS IPC (InterProcess Communication)

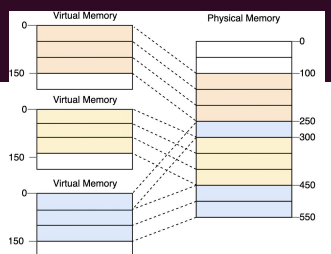
```
#Disable Address space layout randomization (ASLR)
sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

```c
#include <stdio.h>

#define SIZE (1024 * 2)
char taskBBuf[SIZE];

void TaskB(void)
{
  printf("buf start addr: %p\buf end addr: %p\n",
         &taskBBuf[0], &taskBBuf[SIZE]);
}

int main(void)
{
  TaskB();
  while(1);
}
```

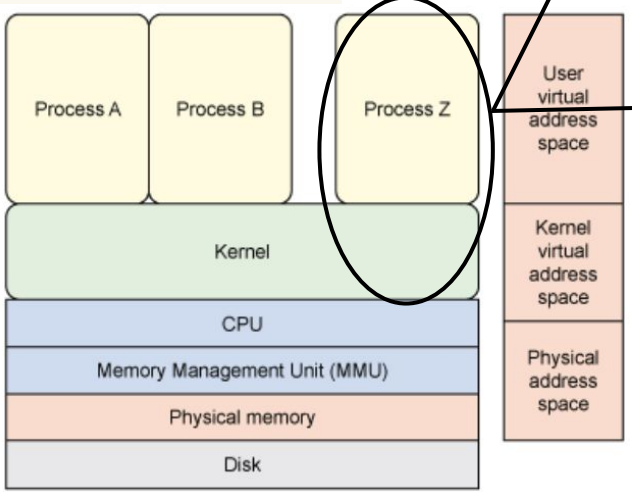
```c
#include <stdio.h>

#define SIZE (1024 * 2)
char taskABuf[SIZE];

void TaskA(void)
{
  printf("buf start addr: %p\buf end addr: %p\n",
         &taskABuf[0], &taskABuf[SIZE]);
}

int main(void)
{
  TaskA();
  while(1);
}
```

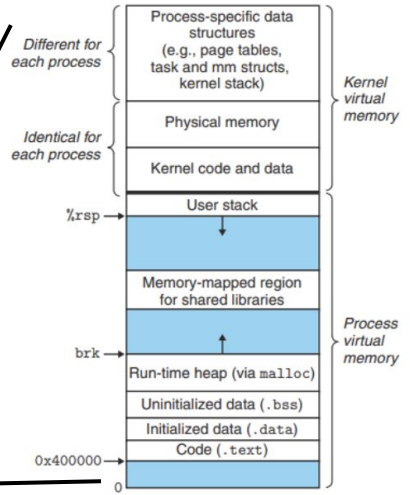
```
clang ./taskA.c -fno-PIE -o taskA
clang ./taskB.c -fno-PIE -o taskB
```

```
alex@alexPC:~/lectures/processTests$ sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
alex@alexPC:~/lectures/processTests$ clang -fno-PIE ./taskB.c -o taskB
alex@alexPC:~/lectures/processTests$ clang -fno-PIE ./taskA.c -o taskA
alex@alexPC:~/lectures/processTests$
alex@alexPC:~/lectures/processTests$ ./taskA&
[1] 3977
alex@alexPC:~/lectures/processTests$ buf start addr: 0x55555555804uf end addr: 0x555555558840

alex@alexPC:~/lectures/processTests$ ./taskB&
[2] 3978
alex@alexPC:~/lectures/processTests$ buf start addr: 0x55555555804uf end addr: 0x555555558840

alex@alexPC:~/lectures/processTests$
alex@alexPC:~/lectures/processTests$ ps
    PID TTY          TIME CMD
   2616 pts/1    00:00:00 bash
   3977 pts/1    00:00:08 taskA
   3978 pts/1    00:00:03 taskB
   3979 pts/1    00:00:00 ps
alex@alexPC:~/lectures/processTests$
```



Linux on 32-bit CPU with MMU



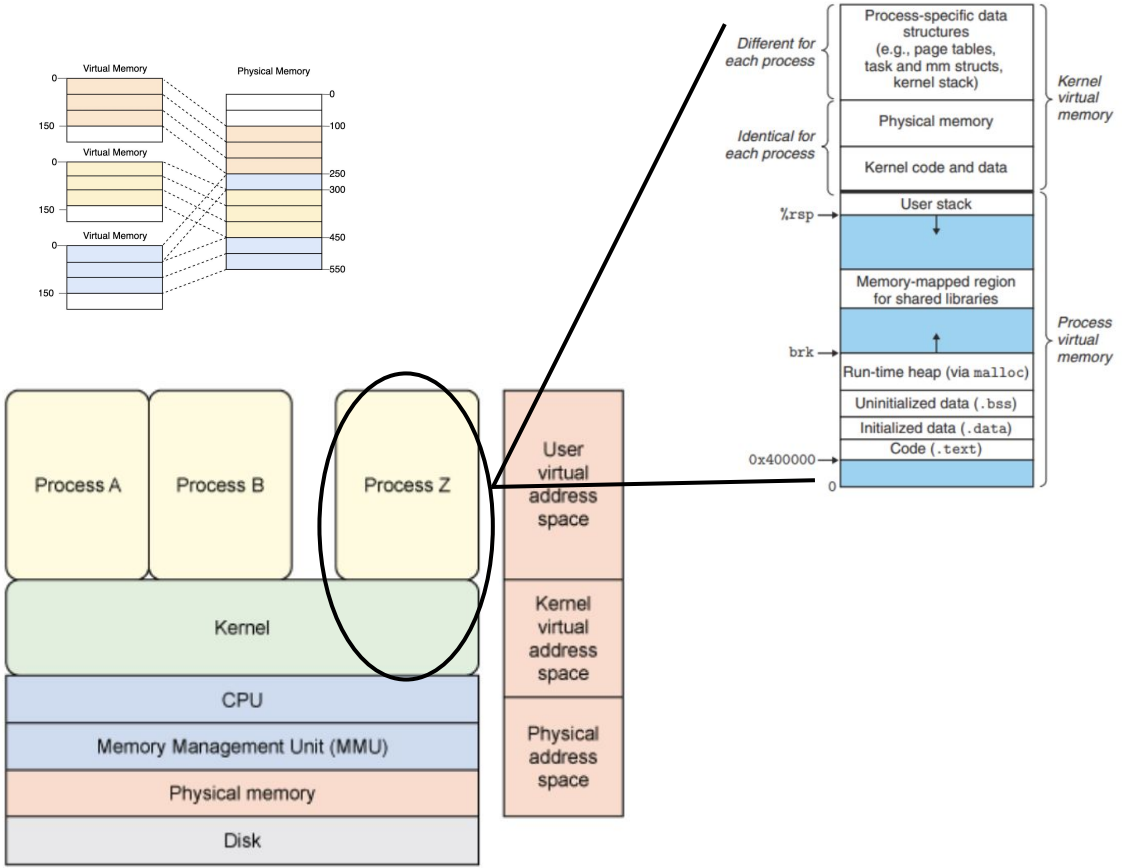


Преимущества виртуального адресного пространства:

+ Все процессы изолированы друг от друга и не могут повредить данные другого процесса

+ Программа всегда линкуется в одно адресное пространство (всегда по одним и тем же адресам)

Linux on 32-bit CPU with MMU



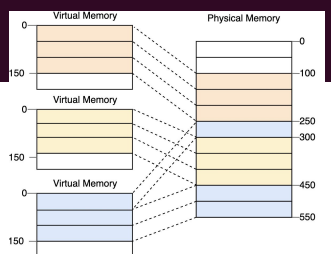
```
#Disable Address space layout randomization (ASLR)
sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

```c
#include <stdio.h>

#define SIZE (1024 * 2)
char taskBBuf[SIZE];

void TaskB(void)
{
  printf("buf start addr: %p\buf end addr: %p\n",
         &taskBBuf[0], &taskBBuf[SIZE]);
}

int main(void)
{
  TaskB();
  while(1);
}
```

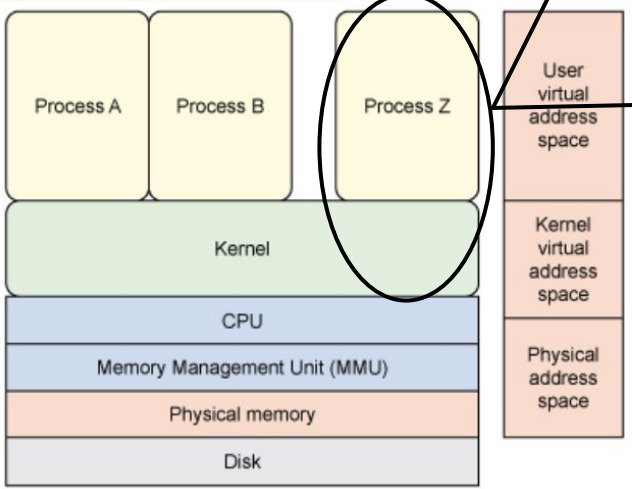
```c
#include <stdio.h>

#define SIZE (1024 * 2)
char taskABuf[SIZE];

void TaskA(void)
{
  printf("buf start addr: %p\buf end addr: %p\n",
         &taskABuf[0], &taskABuf[SIZE]);
}

int main(void)
{
  TaskA();
  while(1);
}
```

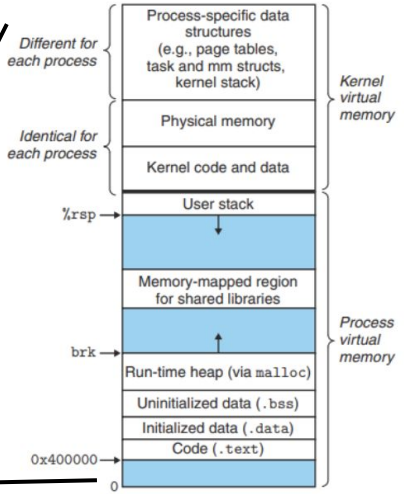
```
clang ./taskA.c -fno-PIE -o taskA
clang ./taskB.c -fno-PIE -o taskB
```

```
alex@alexPC:~/lectures/processTests$ sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
alex@alexPC:~/lectures/processTests$ clang -fno-PIE ./taskB.c -o taskB
alex@alexPC:~/lectures/processTests$ clang -fno-PIE ./taskA.c -o taskA
alex@alexPC:~/lectures/processTests$
alex@alexPC:~/lectures/processTests$ ./taskA&
[1] 3977
alex@alexPC:~/lectures/processTests$ buf start addr: 0x55555555804uf end addr: 0x555555558840

alex@alexPC:~/lectures/processTests$ ./taskB&
[2] 3978
alex@alexPC:~/lectures/processTests$ buf start addr: 0x55555555804uf end addr: 0x555555558840

alex@alexPC:~/lectures/processTests$
alex@alexPC:~/lectures/processTests$ ps
    PID TTY          TIME CMD
   2616 pts/1    00:00:00 bash
   3977 pts/1    00:00:08 taskA
   3978 pts/1    00:00:03 taskB
   3979 pts/1    00:00:00 ps
alex@alexPC:~/lectures/processTests$
```

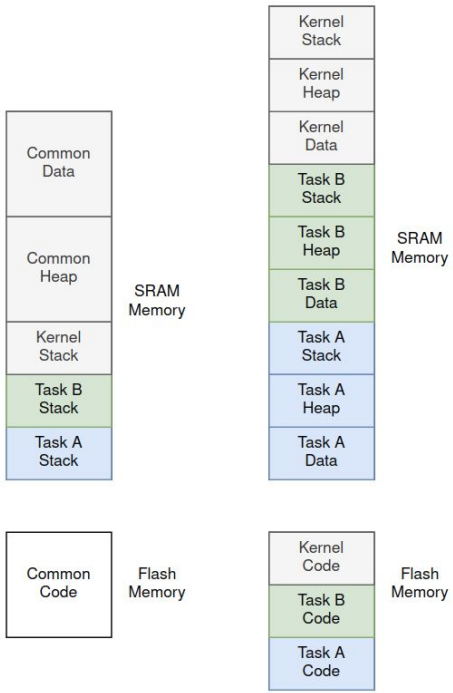


Linux on 32-bit CPU with MMU

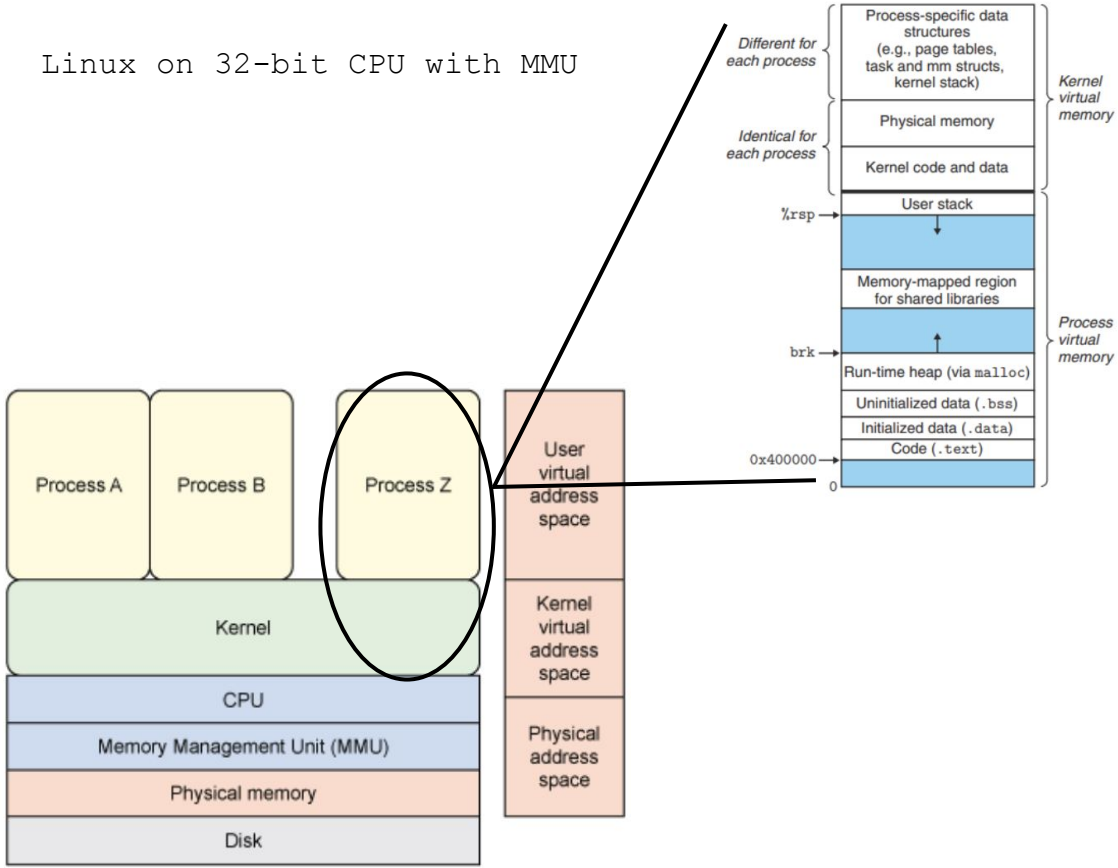




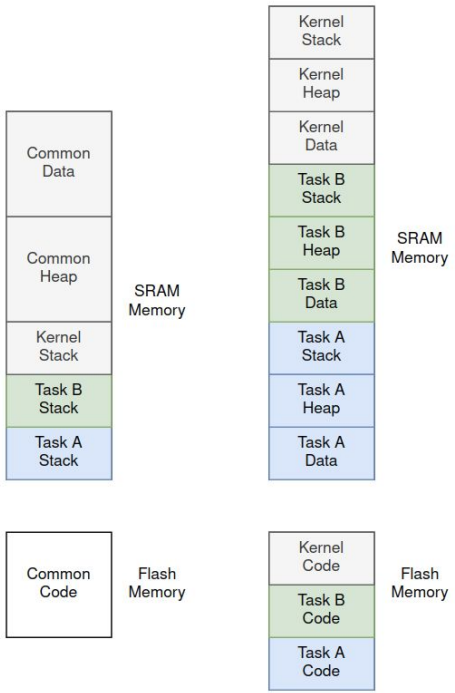
Embedded OS on Cortex-M (without MMU)



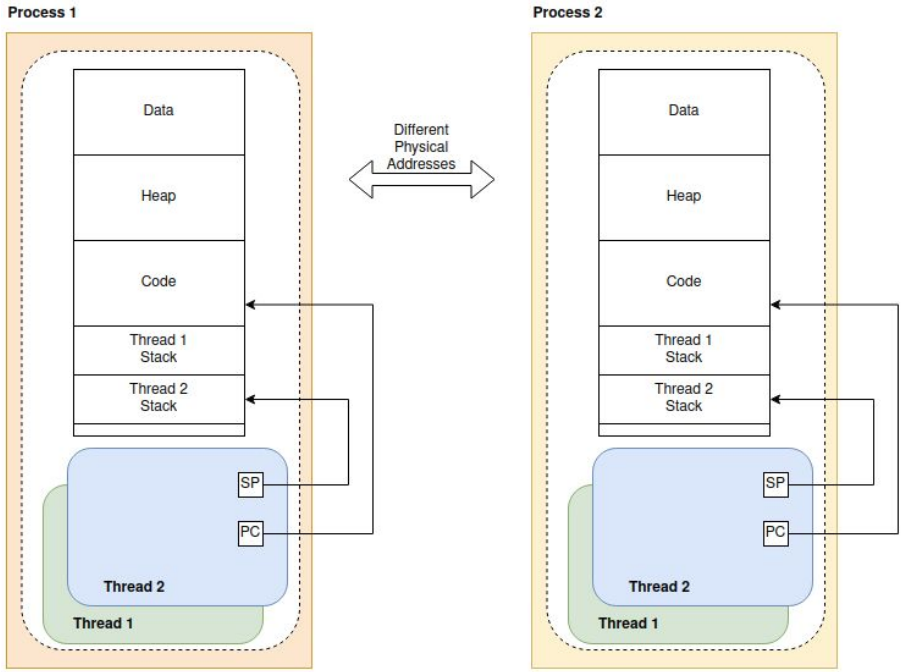
Linux on 32-bit CPU with MMU



Task on Embedded OS (without MMU)



Process vs Thread



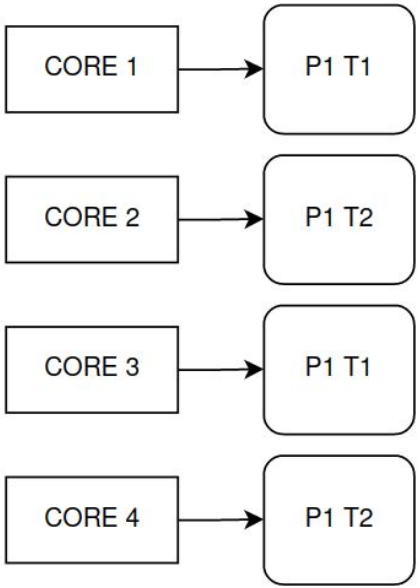

```
#define SIZE 100
int global_buf[SIZE];

void taskA(int in)
{
    for (int i = 0; i < SIZE; ++i)
        global_buf[i] = in;
    while(1);
}

void taskB(int in)
{
    for (int i = 0; i < SIZE; ++i)
        global_buf[i] += in;
    while(1);
}

int main()
{
  std::thread  first (taskA, 7);
  std::thread second (taskB, 42);
  first.join();
  second.join();
}
```

Parallelism

N CORES

На разных ядрах параллельно выполняются задачи;

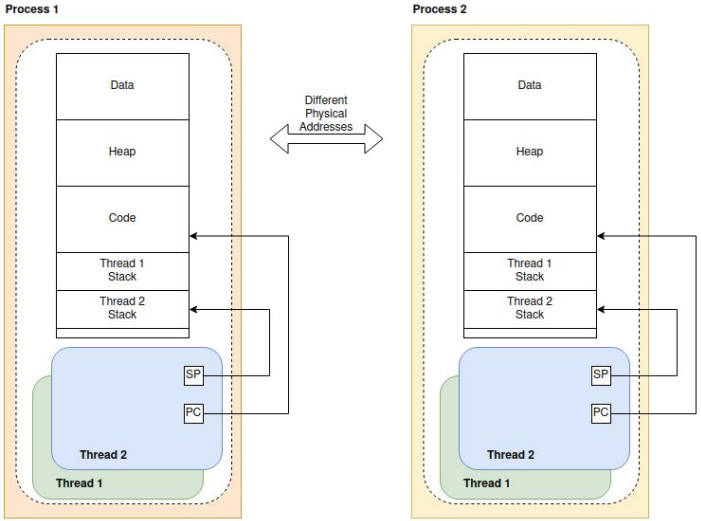
Scheduler переключает задачи;

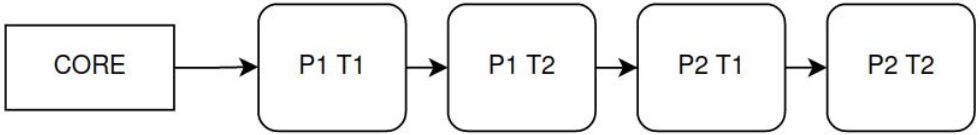


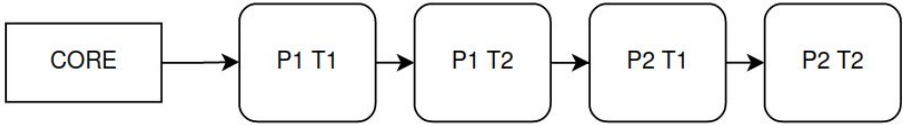
Concurrency

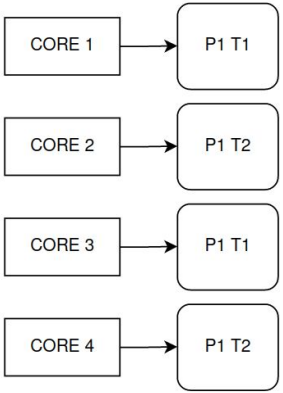
Все задачи выполняются на одном и том же ядре

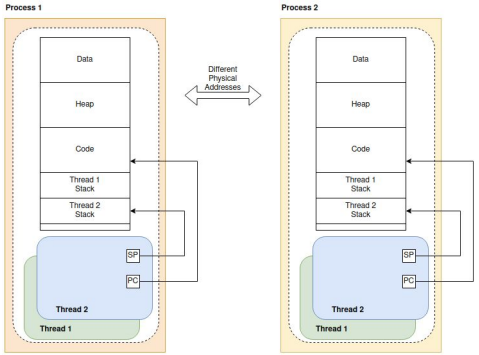
Scheduler переключает задачи, создавая иллюзию, что задачи выполняются параллельно.









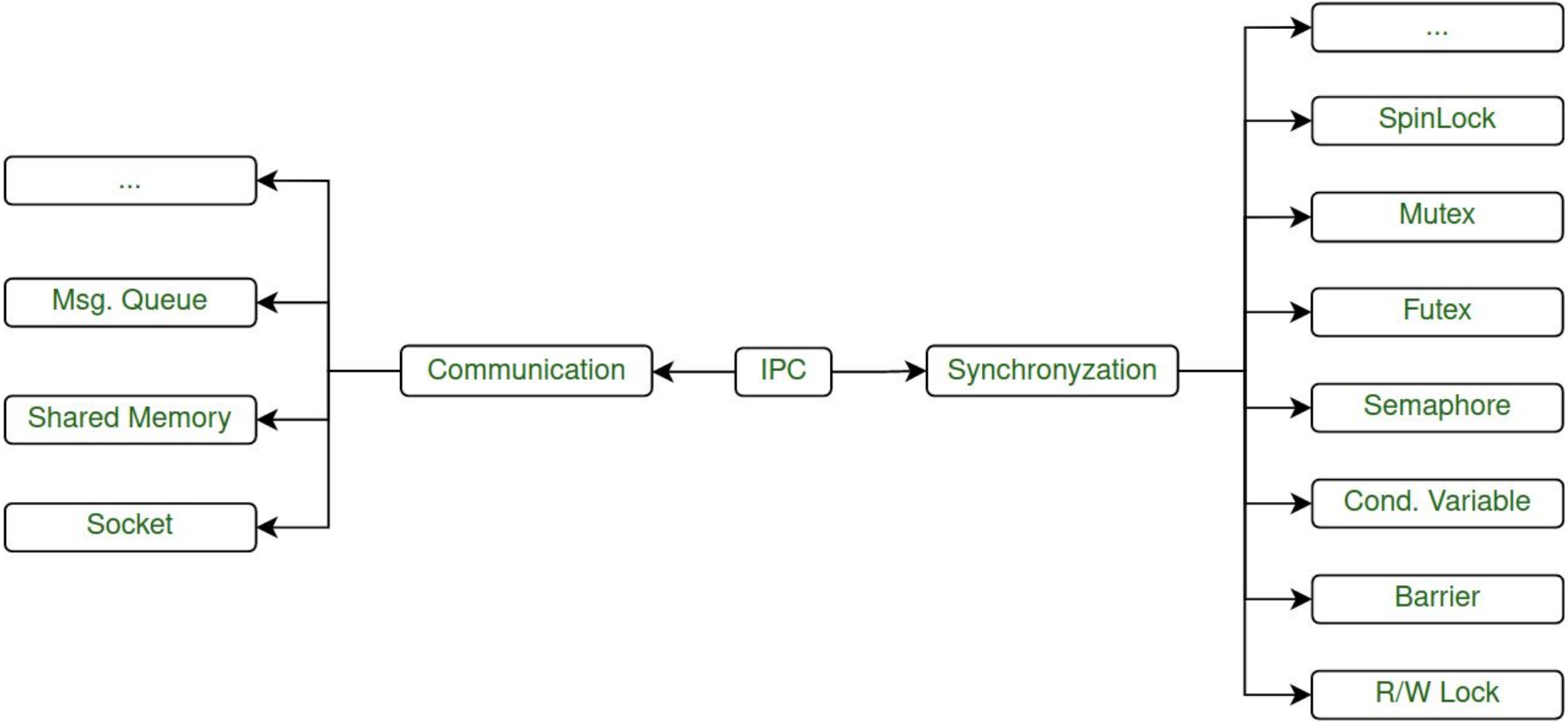


- Как один процесс может передать информацию другому процессу ?
  Example: *Process 1* подготавливает данные, *Process 2* передает подготовленные данные посети.

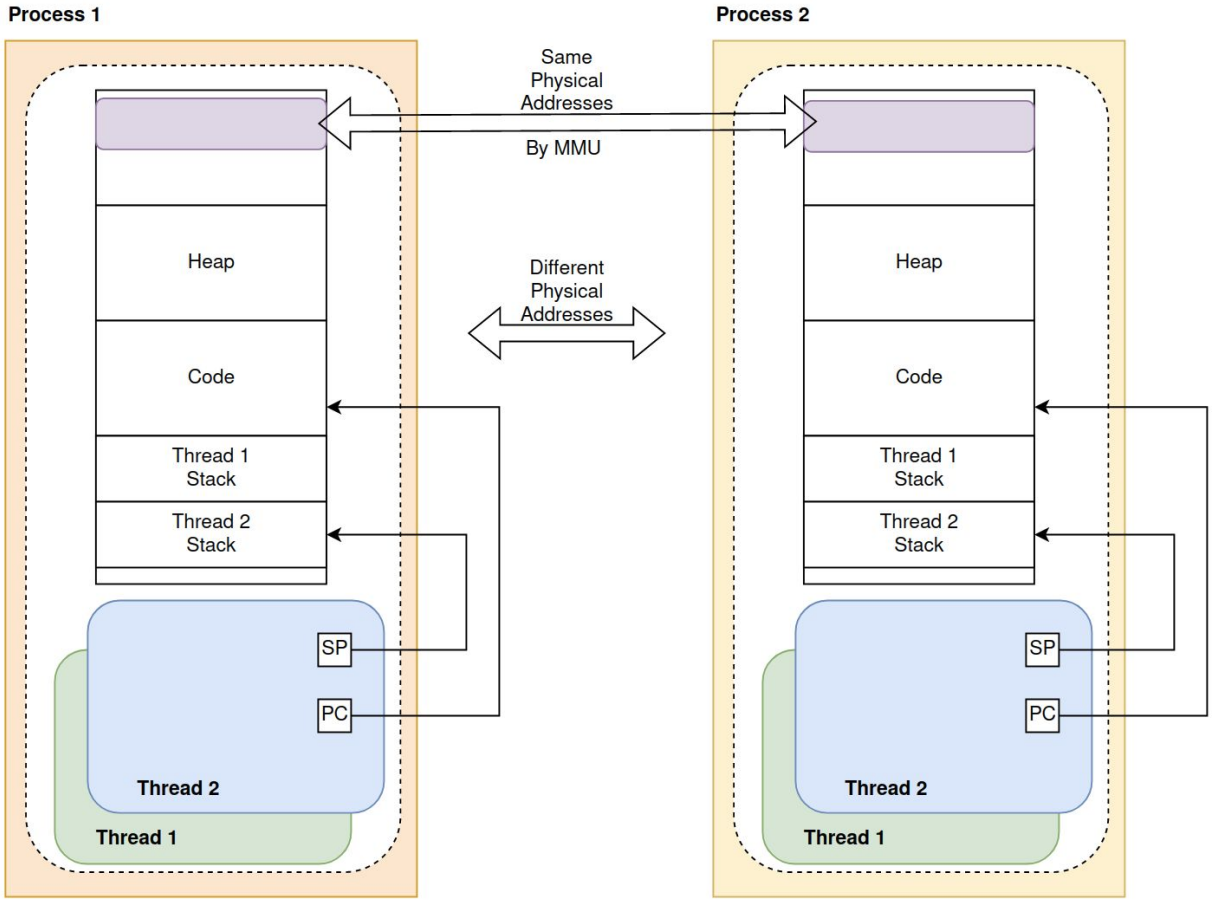
- Как гарантировать что 2 или более процессов (или тредов) не получат доступ к одному и тому же разделяемому ресурсу одновременно ?
  Как обеспечить конкурентный доступ к разделяемому ресурсу ?
  Example: Запретить одновременный доступ к периферии процессора.

- Как обеспечить некую последовательность выполнения процессов(или тредов) ?
  Example: Сначала *thread 1* должен чтото посчитать, и только потом *thread 2* может передавать посчитанные данные.

```
#define SIZE 100
int global_buf[SIZE];

void taskA(int in)
{
    for (int i = 0; i < SIZE; ++i)
        global_buf[i] = in;
    while(1);
}

void taskB(int in)
{
    for (int i = 0; i < SIZE; ++i)
        global_buf[i] += in;
    while(1);
}

int main()
{
  std::thread  first (taskA, 7);
  std::thread second (taskB, 42);
  first.join();
  second.join();
}
```



https://opensource.com/sites/default/files/gated-content/inter-process_communication_in_linux.pdf



Shared Memory позволяет делить 2-м процессам одну и ту же физическую память, что позволяет обмениваться данными между процессами.

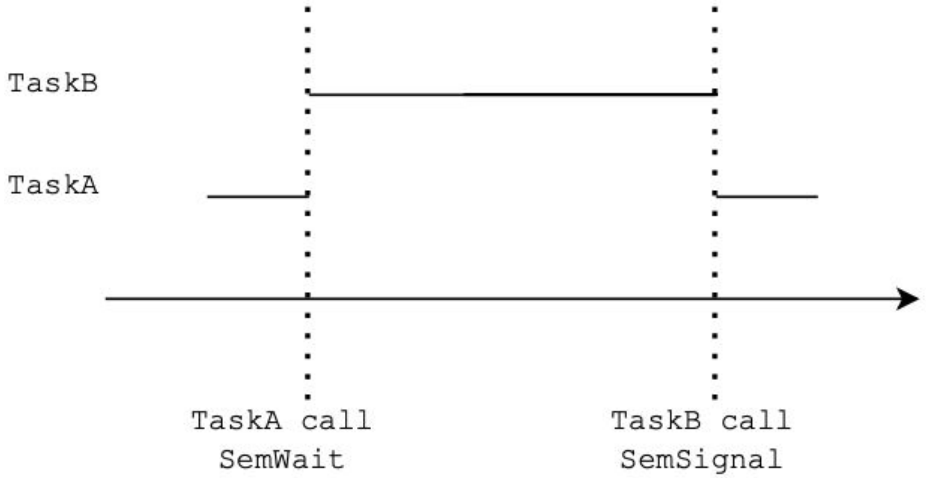
```c
void TaskA(void)
{
  SysCallSemWait(&evt_sem);
  while(1) {
    cntA++;
    SysCallSwitchContext();
  }
}

void TaskB(void)
{
  while(1) {
    cntB++;
    SysCallSwitchContext();
    if (cntB > 100)
      SysCallSemSignal(&evt_sem);
  }
}
```

sem_wait

sem_signal

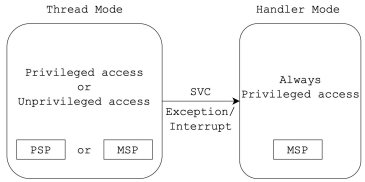
- Как обеспечить некую последовательность выполнения процессов(или тредов) ?

  TaskA должна заснуть(не планироваться планировщиком) до момента пока **TaskB** не досчитает до 100. Потом проснуться и начать выполняться.



https://github.com/badembed/simple_sched_sem/blob/master/Src/main.c

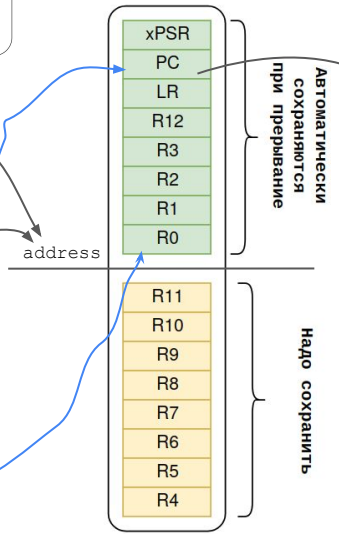
```
SysCallSwitchContext:
  svc #42
  bx lr

SysCallSemWait:
  svc #1
  bx lr

SysCallSemSignal:
  svc #2
  bx lr
```

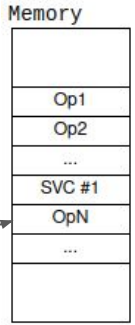



```
SVC_Handler:
  mrs r0, psp    // r0 = psp user mode stack pointer
  B SVC_SysCall
```

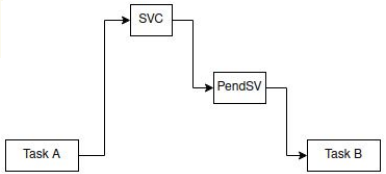
```
void SVC_SysCall(uint32_t *sp)
{
  uint32_t saved_pc = sp[6];
  saved_pc -= 2; // svc opcode is 16 bit, so get it by -2
  uint16_t svc_opcode = *((uint16_t *)saved_pc);
  uint8_t svcnum = (svc_opcode & 0x00ff);
  uint32_t saved_R0 = sp[0]; // it's first arg by ABI

  switch(svcnum) {
  case 1:
    sem_wait((sem_t *)saved_R0);
    break;
  case 2:
    sem_signal((sem_t *)saved_R0);
    break;
  case 42:
    SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;
    break;
  }
}
```

```
void SysCallSemWait(sem_t *sem);
```





16 bit/2 bytes OpCode Size

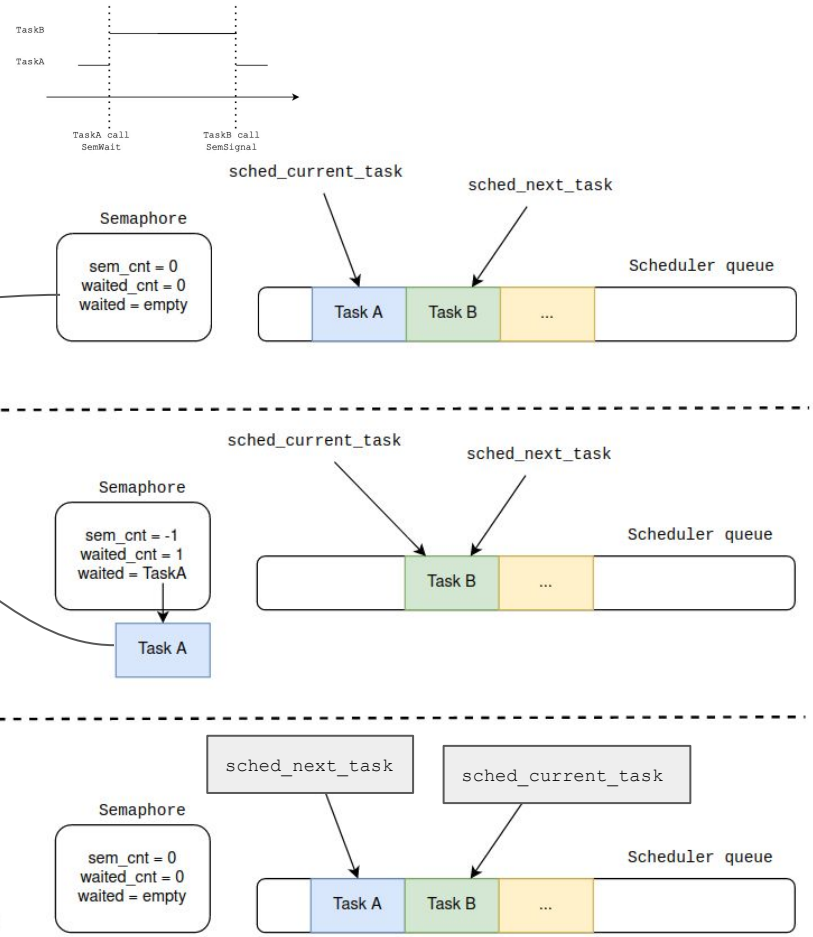



```
PendSV_Handler:
  cpsid i // // Disable Interrupts

  mrs r0, psp   // r0 = psp user mode stack pointer
  sub r0, #32   // allocate 16 byte on the stack for r4-r11 registers (8 registers)

  // save r4-r11 registers at the stack - stmia  r0!, {r4-r11}
  str r4, [r0]
  str r5, [r0, #4]
  str r6, [r0, #8]
  str r7, [r0, #12]
  str r8, [r0, #16]
  str r9, [r0, #20]
  str r10, [r0, #24]
  str r11, [r0, #28]

  ldr r2, =running_task // r2 = addess of cur_task pointer
  ldr r1, [r2]      // r1 = get real task address from cur_task, first value in the task is SP
  str r0, [r1]      // rewrite/save SP address with new in cur_task

  bl ChangeRunningTask // Update cur_task to new/next task

  ldr r2, =running_task
  ldr r1, [r2]
  ldr r0, [r1]      // r0 = SP for cur_task(new/next task)

  // restore r4-r11 registers from the stack - ldmia  r0!,{r4-r11}
  ldr r4, [r0]
  ldr r5, [r0, #4]
  ldr r6, [r0, #8]
  ldr r7, [r0, #12]
  ldr r8, [r0, #16]
  ldr r9, [r0, #20]
  ldr r10, [r0, #24]
  ldr r11, [r0, #28]
  add r0, #32
```

https://github.com/badembed/simple_sched_sem/blob/master/Src/sched.s

```c
void sem_init(sem_t *sem)
{
  if (sem == NULL)
    return;
  sem->sem_cnt = 0;
  sem->waited_cnt = 0;
}

void sem_wait(sem_t *sem)
{
  if (sem == NULL)
    return;

  __disable_irq();
  sem->sem_cnt--;
  if (sem->sem_cnt < 0) {
    sem->waited[sem->waited_cnt] = sched_current_task;
    sem->waited_cnt++;
    sched_current_task = sched_next_task;
    SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;
  }
  __enable_irq();
}

void sem_signal(sem_t *sem)
{
  if (sem == NULL)
    return;

  __disable_irq();
  sem->sem_cnt++;
  if (sem->waited_cnt > 0) {
    sem->waited_cnt--;
    sched_next_task = sem->waited[sem->waited_cnt];
    SCB->ICSR |= SCB_ICSR_PENDSVSET_Msk;
  }
  __enable_irq();
}
```

```c
#define STACK_SIZE 256
typedef struct {
  unsigned int sp;
  unsigned int stack[STACK_SIZE];
} task_t;

typedef struct {
  int sem_cnt;
  task_t *waited[10];
  int waited_cnt;
} sem_t;
```



https://github.com/badembed/simple_sched_sem/blob/master/Src/sem.c

Semaphores in our simple scheduler

```c
#define SIZE 10
#define TASK_A_MAGIC 0xAA
#define TASK_B_MAGIC 0xBB

char buf[SIZE];
volatile int cnt = 0;

void TaskA(void)
{
  while(1) {
    __disable_irq();
    buf[cnt] = TASK_A_MAGIC;
    cnt++;
    if (cnt == SIZE)
      cnt = 0;
    __enable_irq();

    // some long  logic
  }
}

void TaskB(void)
{
  while(1) {
    __disable_irq();
    buf[cnt] = TASK_B_MAGIC;
    cnt++;
    if (cnt == SIZE)
      cnt = 0;
    __enable_irq();

    // some long  logic
  }
}
```

Проблема: Task A и **Task B** пытаются использовать один и тот же ресурс одновременно. Возникает гонка за доступ к данным.

Решение:

- Если это одноядерный процессор, то достаточно просто вставить критическую зону в этот участок кода - например запретить прерывания.

- Если это многоядерный процессор, то отключение прерываний не поможет - Task 1 на CPU1 и Task 2 на CPU2 могут устроить гонку в этом месте, выполняясь одновременно.
  Тогда используется специальный механизм синхронизации - **SpinLock.**

```
static inline void arch_spin_lock(arch_spinlock_t *lock)
{
        unsigned long tmp;

        __asm__ __volatile__(
"1:     ldrex   %0, [%1]\n"
"       teq     %0, #0\n"
        WFE("ne")
"       strexeq %0, %2, [%1]\n"
"       teqeq   %0, #0\n"
"       bne     1b"
        : "=&r" (tmp)
        : "r" (&lock->lock), "r" (1)
        : "cc");

        smp_mb();
}
```

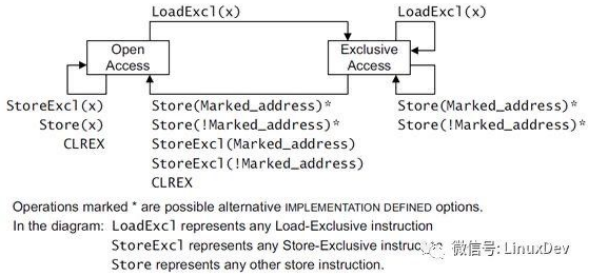
```
static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
        smp_mb();

        __asm__ __volatile__(
"       str     %1, [%0]\n"
        :
        : "r" (&lock->lock), "r" (0)
        : "cc");

        dsb_sev();
}
```

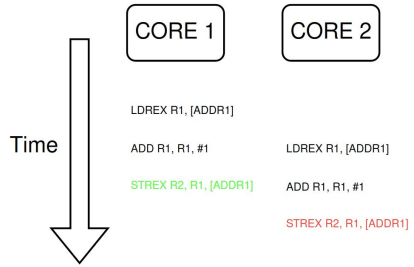
```
ldrex   r1, [r0]
add     r1, r1, #1
strex   r2, r1, [r0]
```

LDREX читает в регистр **R1** значение из памяти в регистре **R0** и устанавливает эксклюзивный доступ к памяти в регистре **R0;**

STREX пытается записать значение из регистра **R1** в память из регистра **R0,** результат записи (прошла ли эксклюзивная запись в эту память или нет) записывается в регистр **R2;**

если другой CPU выполнит запись в эксклюзивную память между выполнениями этих **LDREX** и **STREX,** то strex не произведет запись, а вернет ошибку в **R2.**

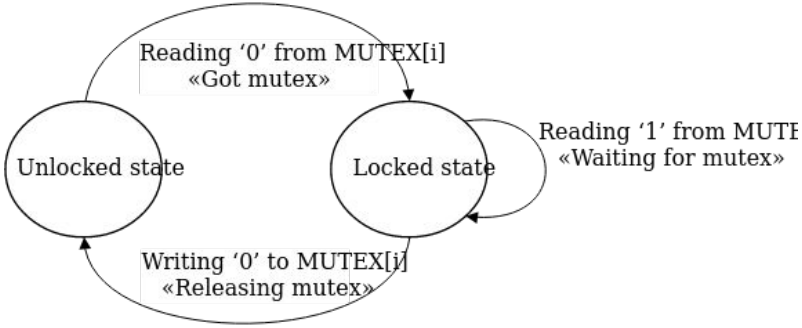


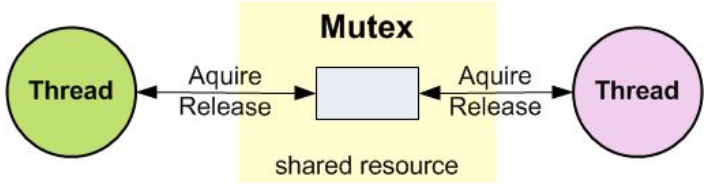


https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Exclusive-accesses

https://linux-concepts.blogspot.com/2018/05/spinlock-implementation-in-arm.html

SpinLock, ARM ldrex/strex





Linux

```
struct mutex {
  atomic_t              count;
  spinlock_t            wait_lock;
  struct list_head      wait_list;
};
```

count - mutex state:
1 = unlocked state.
0 = locked state.

wait_lock - spinlock for the protection of a wait queue.

wait_list - list of waiters which represents wait queue for a certain lock.

If the lock is unable to be acquired by a process, this process will be added to wait queue.

spinlock	mutex
Если lock захвачен другой задачей: Бесконечно крутится в цикле пытаясь захватить lock, до тех пор пока lock не будет освобожден другой задачей	**Если lock захвачен другой задачей:** Вызывается планировщик и задача убирается из списка на планирования, до тех пор пока mutex не будет освобожден другой задачей

https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-4.html

```c
pthread_mutex_t lock;
pthread_t tid[2];
int counter;

void* doSomeThing(void *arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i=0; i<(0xFFFFFFFF);i++);

    printf("\n Job %d finished\n", counter);

    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    pthread_mutex_init(&lock, NULL);
    for(int i = 0; i < 2; ++i)
    {
        pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

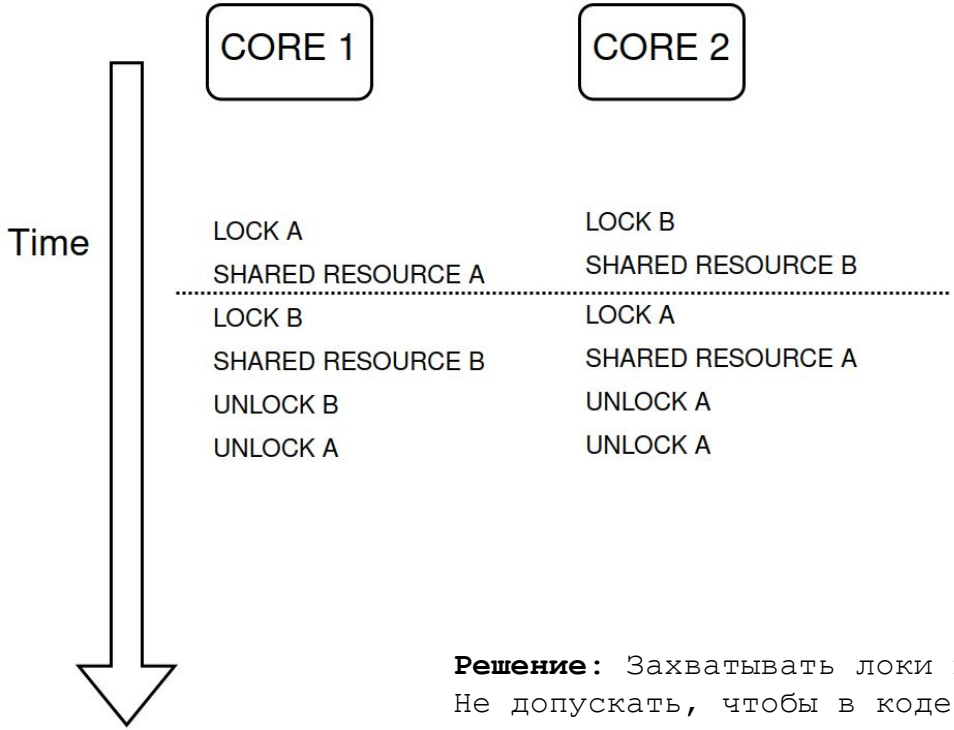
AS WAS:

```
Job 1 started

Job 2 started

Job 2 finished

Job 2 finished
```

AS IS:

```
Job 1 started

Job 1 finished

Job 2 started

Job 2 finished
```

NOTE: ***pthread_mutex*** is library call from ***LibC***.
It uses ***futex*** to implement ***pthread_mutex***.

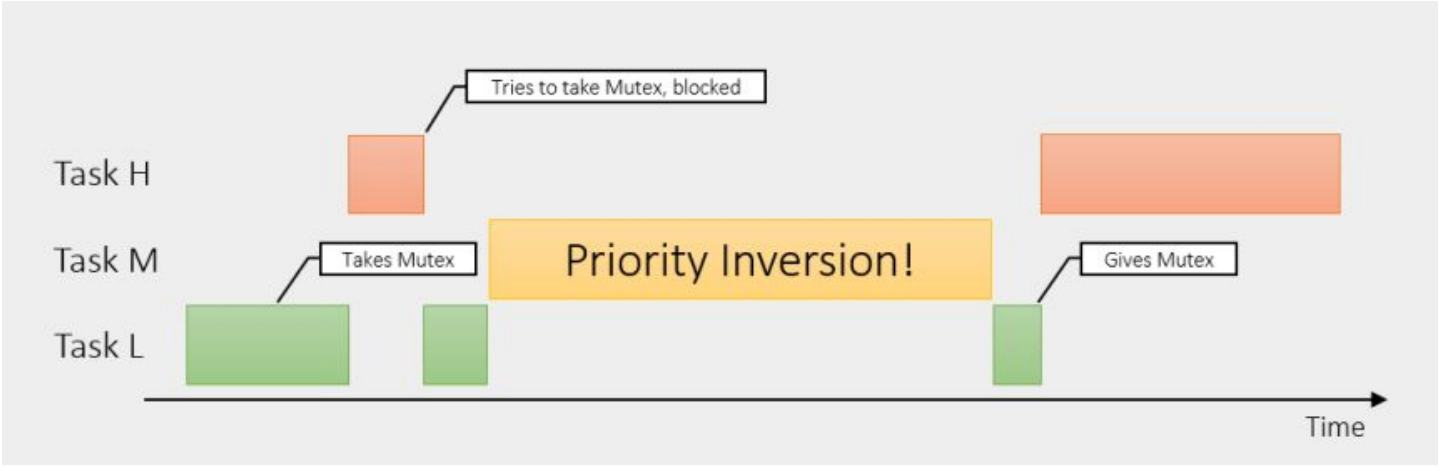


ABBA Deadlock: Происходит когда 2 лока (A и B) захватываются в разном порядке. В этом случае возможно:

1. **Thread 1** захватил **Lock A**
2. **Thread 2** захватил **Lock B**
3. **Thread 1** пытается захватить **Lock B**, но **Lock B** уже захвачен **Thread 2**.
4. **Thread 2** пытается захватить **Lock A**, но **Lock A** уже захвачен **Thread 1**.
5. Происходит взаимный **Deadlock** - ни **Thread A**, ни **Thread B** не может продолжить выполнение !!!

Решение: Захватывать локи всегда в одном порядке: Либо A->B, либо B->A. Не допускать, чтобы в коде был разный порядок захвата локов.

Lock Validators:
Dynamic analyzers: LockDep (In Linux), Witness (in FreeBSD)
Static analyzers: ...



Task L - Низкий приоритет,
Task M - Средний приоритет,
Task H - Высокий приоритет

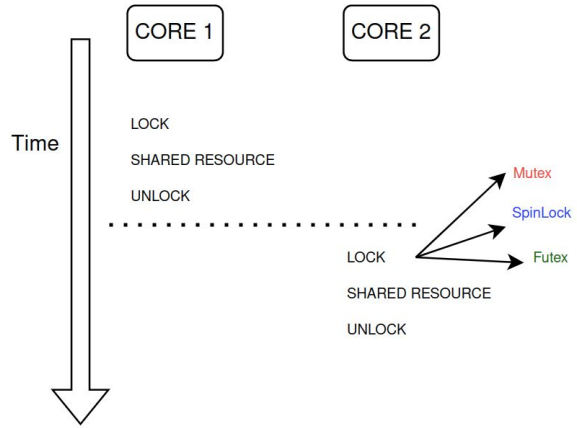
Проблема: ***Task H*** и ***Task L*** борются за один и тоже **mutex**. ***Task L*** захватывает ***mutex***. ***Task H*** имеет высокий приоритет и должна ***закончится как можно раньше***. Но
она блокируется на ***mutex,*** который захвачен низко приоритетной задачей***Task L.***
Task L должна как можно быстрее закончить работу с***shared resource*** и отпустить **mutex**. Но ее вытесняет ***Task M***. А это значит, что ***Task H*** дополнительно будет ждать завершения задачи***Task H***.

Решение:

1. **Priority Inheritance:** Когда ***Task H*** попытается захватить ***mutex*** и поймет что он захвачен задачей с более низким приоритетом (***Task L***), приоритет ***Task L*** будет автоматически ***повышен до приоритета задачи Task H.***
2. **Priority Ceiling:** Когда инициализируется ***mutex, указывают самый приоритет, до которого автоматически повышается приоритет задачи,*** захватившей этот ***mutex***. Таким образом можно повысить приоритет задачи***Task L,*** до более высоких значений, чтобы не допустить ее вытеснения задачей***Task M.***

https://habr.com/ru/company/pvs-studio/blog/315496/

Criteria	Mutex	Spinlock
Mechanism	Test for lock. If available, use the resource If not, go to wait queue	Test for lock. If available, use the resource. If not, loop again and test the lock till you get the lock
When to use	Used when putting process is not harmful like user space programs. Use when there will be considerable time before process gets the lock.	Used when process should not be put to sleep like Interrupt service routines. Use when lock will be granted in reasonably short time.
Drawbacks	Incur process context switch and scheduling cost.	Processor is busy doing nothing till lock is granted, wasting CPU cycles.



Если использовать **mutex** - будет вызван syscall и mutex counter будет инкрементирования в режиме ядра. Если **counter == 0** (никто не занял mutex), то мы выйдем из syscall, не сделав никакой полезной работы в syscall (т.к. в этом случае mutex может быть захвачен, и блокировка задачи не требуется).
Если использовать **spinlock**, то syscall не будет - мы можем проверить counter в userspace. Но если **counter != 0** (другая задача заняла spinlock) - мы будем висеть в пустом цикле в userspace spinlock (блокировки задач не будет), до тех пор пока другая задача не освободит spinlock.

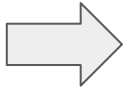
Возьмем лучшее из 2х вариантов: Futex - проверять counter в userspace с помощью атомарных команд (ldrex/strex), и только если **counter != 0**, делать syscall для блокировки задачи !!!

https://www.akkadia.org/drepper/futex.pdf
http://www.rkoucha.fr/tech_corner/the_futex.html#Futex_syscall

```c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
pthread_rwlock_t rwlock;

void *write_thread(void *temp) {
  printf("write_thread started\n");
  pthread_rwlock_wrlock(&rwlock);
  printf("write_thread acquire rwlock\n");

  sleep(1);

  pthread_rwlock_unlock(&rwlock);
  printf("write_thread release rwlock\n");
}

void * read_thread(void *temp) {
  static int cnt = 0;
  int localcnt = cnt++;

  printf("read_thread %d started\n", localcnt);
  pthread_rwlock_rdlock(&rwlock);
  printf("read_thread %d acquire rwlock\n", localcnt);

  sleep(1);

  pthread_rwlock_unlock(&rwlock);
  printf("read_thread %d release rwlock\n", localcnt);
}

main() {
  pthread_t tid[3];

  pthread_rwlock_init(&rwlock,NULL);

  pthread_create(&tid[1], NULL, &read_thread, NULL);
  pthread_create(&tid[0], NULL, &write_thread, NULL);
  pthread_create(&tid[2], NULL, &read_thread, NULL);

  pthread_join(tid[0], NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[0], NULL);

  pthread_rwlock_destroy(&rwlock);
}
```

Read-Write Lock позволяет блокировать shared resource только когда в него производится запись.

Несколько thread'ов могу одновременно читать shared resource (R-Lock). Но когда какой то thread пишет в shared resource (W-Lock), ресурс блокируется как на чтение так и на запись для других thread'ов.



```
read_thread 0 started
read_thread 1 started
read_thread 1 acquire rwlock
read_thread 0 acquire rwlock
write_thread started
read_thread 1 release rwlock
read_thread 0 release rwlock
write_thread acquire rwlock
write_thread release rwlock
```

```c
#include <pthread.h>
#include <stdio.h>

pthread_barrier_t barrier;

void* thread_func(void* aArgs)
{
  pthread_barrier_wait(&barrier);

  printf("Entering thread %p\n", (void*)pthread_self());
  int i;
  for(i = 0 ; i < 5; i++)
    printf("val is %d in thread %p \n", i, (void*)pthread_self());
}

int main()
{
  pthread_t thread_1, thread_2;
  pthread_barrier_init(&barrier, NULL, 2);

  pthread_create(&thread_1, NULL, (void*)thread_func, NULL);
  printf("Thread %p created\n", (void*)thread_1);

  usleep(500);

  pthread_create(&thread_2, NULL, (void*)thread_func, NULL);
  printf("Thread %p created\n", (void*)thread_2);

  pthread_join(thread_1, NULL);
  pthread_join(thread_2, NULL);

  pthread_barrier_destroy(&barrier);

  return 0;
}
```

Posix Barrier IPC позволяет выставлять специальный барьер, которого должны достигнуть все треды, чтобы продолжить выполнение программы за этим барьером.

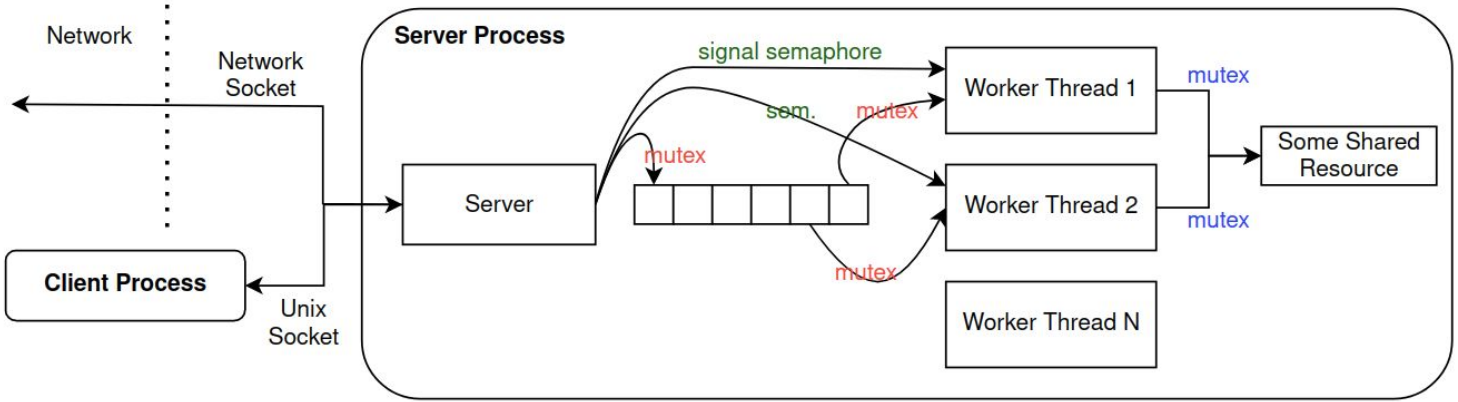
В примере: Posix Barrier IPC гарантирует что N тредов (в примере N=2, задается в pthread_barrier_init) должны достигнуть pthread_barrier_wait. Когда 1-й thread достигнет pthread_barrier_wait он заблокируется. Когда 2-й thread достигнет pthread_barrier_wait оба (1-й и 2-й) thread продолжат выполнять программу (1-й thread разблокируется).

Данный тип синхронизации бывает полезен когда вам в каждом thread'е, необходимо произвести инициализацию неких данных, до старта неких других действий.

Что вынести из данной лекции:

- Process vs Thread, IPC - why ??
- SysCall implementation
- Semaphore
- Mutex vs Spinlock vs Futex
- RwLock, Unix Barriers

1. Если у mutex, counter может принимать 2 состояния (mutex захвачен или не захвачен), то у semaphore counter может возрастать бесконечно (до unsigned int MAX value в нашем примере);
2. Если у mutex есть хозяин(тот кто захватил mutex в данный момент) и только хозяин может отпустить mutex, то у semaphore нет хозяина и его “захватить” может одна задача, а “отпустить” другая. Это позволяет использовать его в дополнительном наборе задач (помимо доступа к разделяемому ресурсу - счетный семафор) - сигналить о неком событие - сигнальный семафор.



Server Process запускает N+1 Threads:

1. Server для получения данных от клиента. Данные приходят либо по сети через ***Network Socket***, либо от клиентского процесса (на тойже машине) через ***Unix Socket***. Сервер получает данные, обрабатывает запрос, кладет данные в очередь (**mutex** используется чтобы синхронизировать одновременное чтение и запись в очередь) и запускает задачу на выполнение в Worker Thread (**semaphore** используется чтобы разбудить Worker Thread).

2. N "Worker Threads" для выполнения некой работы (обсчета данных, чтения с устройства и т.д.). Каждый Worker Thread спит на **semaphore**, и просыпается когда Server установит **semaphore**. Каждый Worker Thread читает данные из очереди и выполняет соответствующую работу. Для синхронизации очереди используется **mutex**.

3. Worker Threads могут попытаться получить доступ к некому разделяемому ресурсу. Для синхронизации доступа к нему используется **mutex**.

```cpp
// condition_variable example
#include <iostream>           // std::cout
#include <thread>             // std::thread
#include <mutex>              // std::mutex, std::unique_lock
#include <condition_variable> // std::condition_variable

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void print_id (int id) {
  std::unique_lock<std::mutex> lck(mtx);
  while (!ready) cv.wait(lck);
  // ...
  std::cout << "thread " << id << '\n';
}

void go() {
  std::unique_lock<std::mutex> lck(mtx);
  ready = true;
  cv.notify_all();
}

int main ()
{
  std::thread threads[10];
  // spawn 10 threads:
  for (int i=0; i<10; ++i)
    threads[i] = std::thread(print_id,i);

  std::cout << "10 threads ready to race...\n";
  go();                       // go!

  for (auto& th : threads) th.join();

  return 0;
}
```

Conditional Variable позволяет сигналить другим thred'ам о неком событии, по которому эти thread'ы должны проснуться.

В примере: print_id threads блокируются на conditional variable (**cv**), а main thread из функции **go** сигналит (через conditional variable (**cv**)) чтобы разбудить **print_id** threads.

```
10 threads ready to race...
thread 2
thread 0
thread 9
thread 4
thread 6
thread 8
thread 7
thread 5
thread 3
thread 1
```