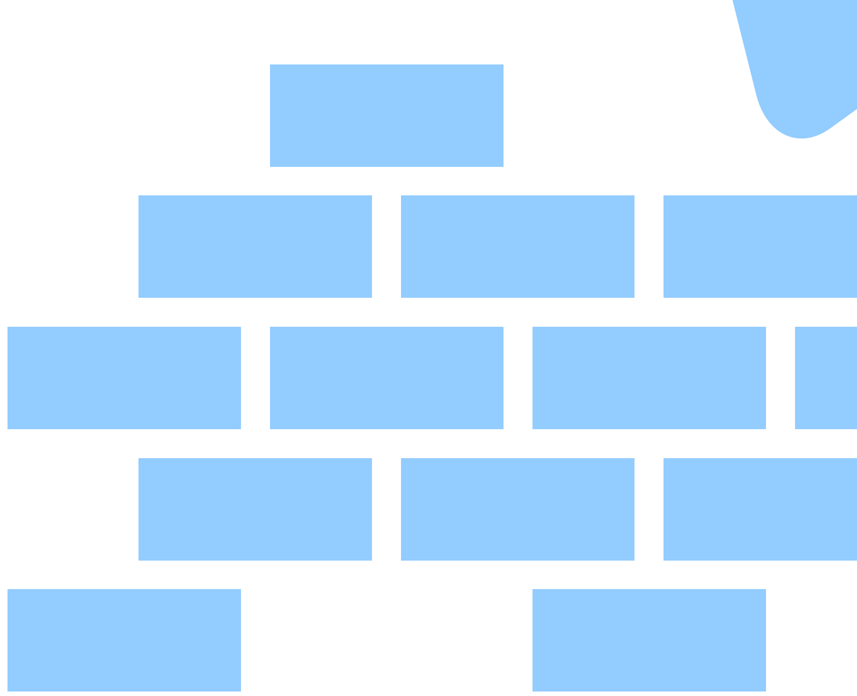
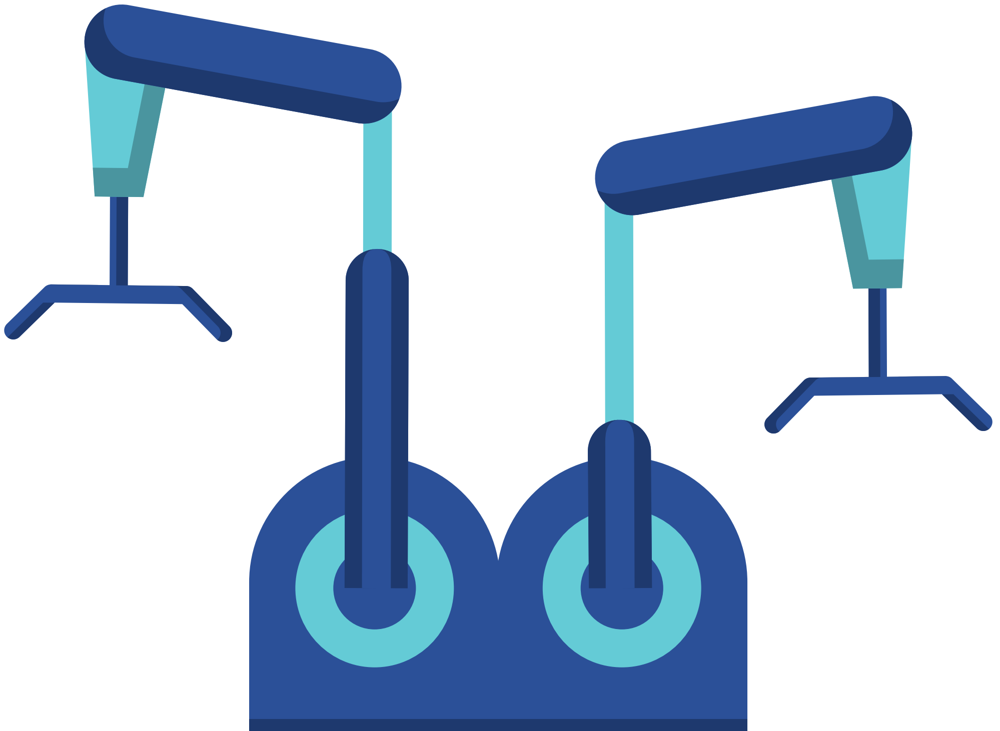


Implémentation du multicœur en couche logicielle basse



Summary

- Objectif
- Qu’est-ce qu’un Spinlock ?
- Monocœur → Multicœur
- Mise en place d’un big lock
- Passage aux small spinlocks
- Conclusion

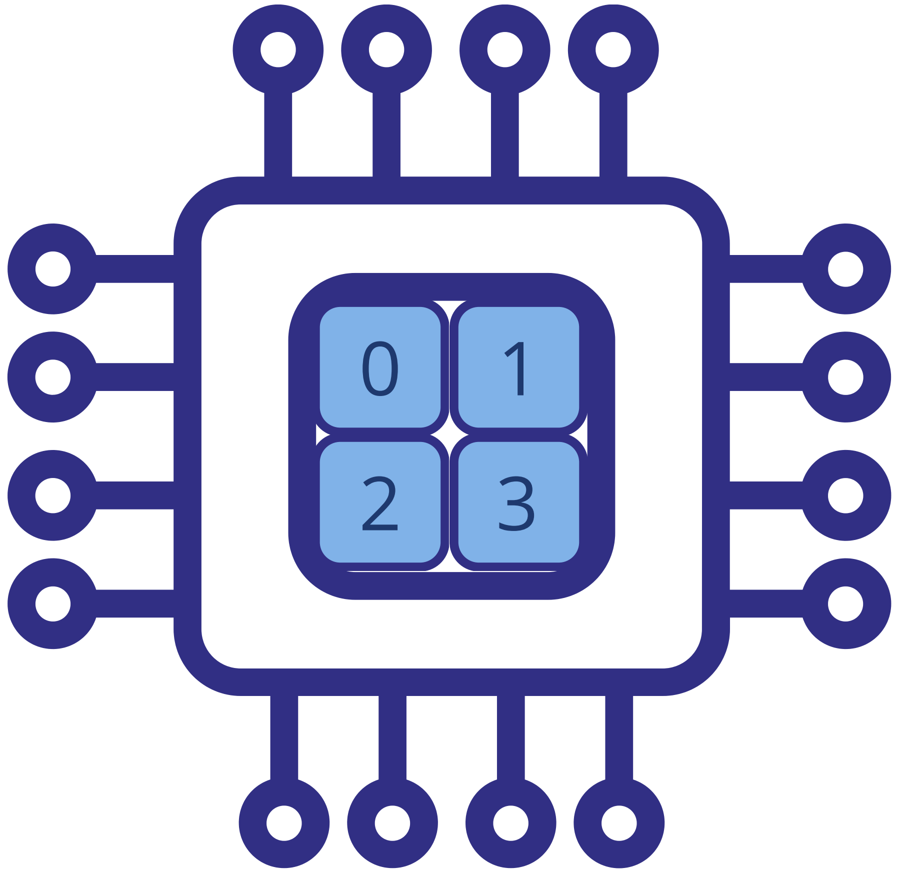
Objectif

Support multicœur et gestion de la concurrence

Concurrence

Cœur 0 :
uart_send_string(“Hello, World!”)

Cœur 1 :
uart_send_string(“Hello, World!”)

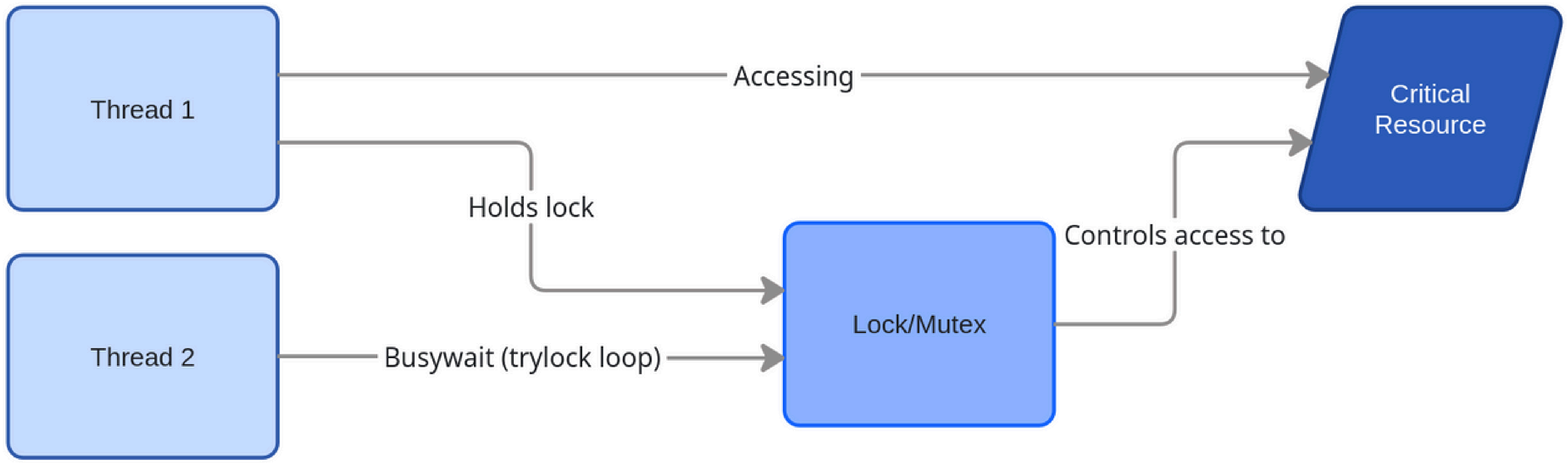


Cœur 2 :
uart_send_string(“Hello, World!”)

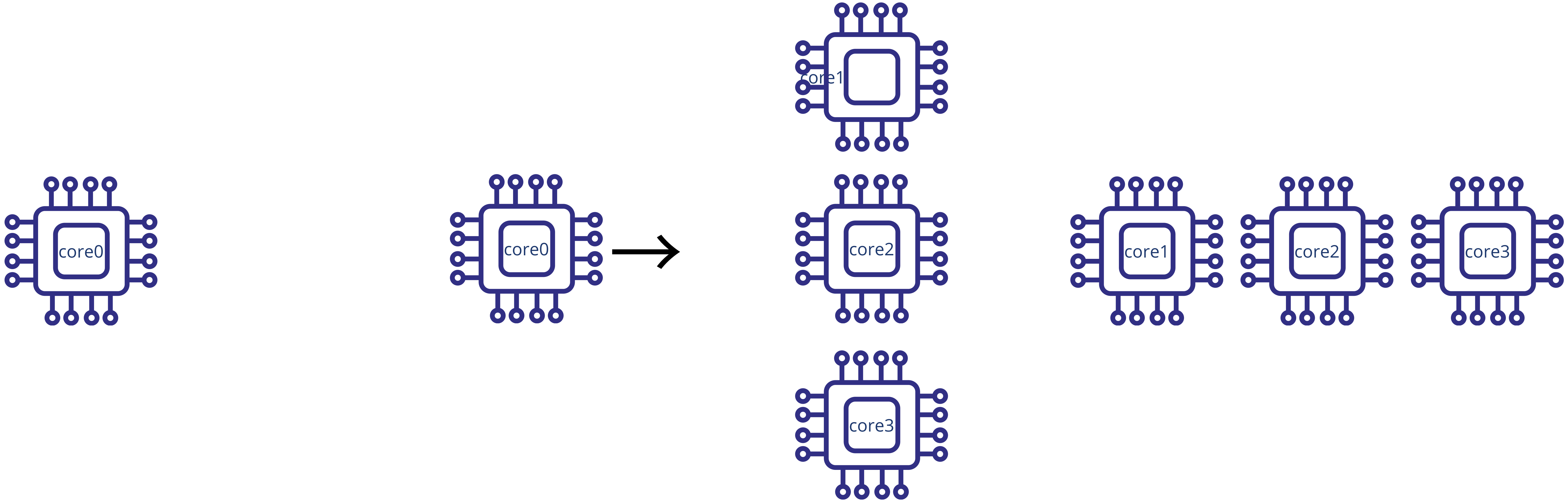
Cœur 3 :
uart_send_string(“Hello, World!”)

Sortie :
HHeellllloo,, WWHoerrllddo!!, HWeolrllod,! World!

Qu’est-ce qu’un spinlock ?



Monocœur → Multicœur



Etape 1 : Initialiser notre système et le cœur 0

Etape 2 : Réveiller les autres cœurs

Etape 3 : Initialiser tous les cœurs

Monocœur → Multicœur

```
.globl start_cores
start_cores:
  adrp x2, _start
  add x2, x2, #:lo12:_start
  ldr x3, =VA_START
  sub x2, x2, x3 // Since tl
give the other cores a phys:
virtual offset
  mov x1, #0xd8
  str x2, [x1,#8]!
  str x2, [x1,#8]!
  str x2, [x1,#8]
  ret
```

Etape 2 : Réveiller les autres cœurs

```
if (core_id == 0){
  uart_init();
  init_printf(NULL, putc);
  printf("kernel boots ...\n\r");
  enable_interrupt_controller();
  int res = copy_process(PF_KTHREAD, (unsigned
long)&kernel_process, 0);
  if (res < 0) {
      printf("error while starting kernel process");
      return;
  }
  start_cores();
  printf("Starting other cores...\n\r");
}
irq_vector_init();
generic_timer_init();
enable_irq();

printf("Core %d : Successfully started\n", core_id);
unlock();
while(1){
  lock();
  schedule(core_id);
  unlock();
  asm("wfi");
}
```

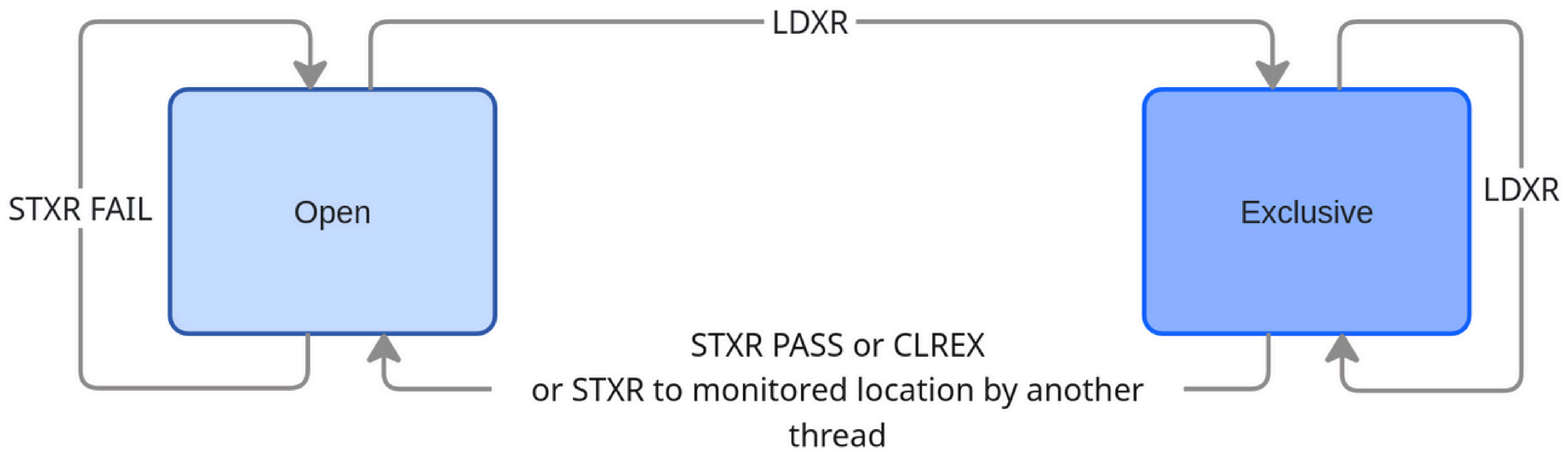
Etape 1 : Initialiser notre système et le cœur 0

```
// Fetch PMD address
adrp    x11, pg_dir
add     x11, x11, #(3 * PAGE_SIZE)

// Map second PMD address
orr     x12, x11, #MM_TYPE_PAGE_TABLE
str     x12, [x10, #8]

// Map timer addresses
mov     x0, x11
ldr     x1, =GENERIC_TIMER_START                // Physical
address of timer region
ldr     x2, =(VA_START + GENERIC_TIMER_START)    // Virtual
address start
ldr     x3, =(VA_START + GENERIC_TIMER_START)    // Virtual
address end : same address as start to enforce the mapping
of a single section
```

```
void enable_interrupt_controller()
{
  // Enables Core 0-3 Timers interrupt control for the
generic timer
  put32(TIMER_INT_CTRL_0, TIMER_INT_CTRL_0_VALUE);
  put32(TIMER_INT_CTRL_1, TIMER_INT_CTRL_0_VALUE);
  put32(TIMER_INT_CTRL_2, TIMER_INT_CTRL_0_VALUE);
  put32(TIMER_INT_CTRL_3, TIMER_INT_CTRL_0_VALUE);
}
```

Etape 3 : Initialiser tous les cœurs

Implémentation: Spinlock



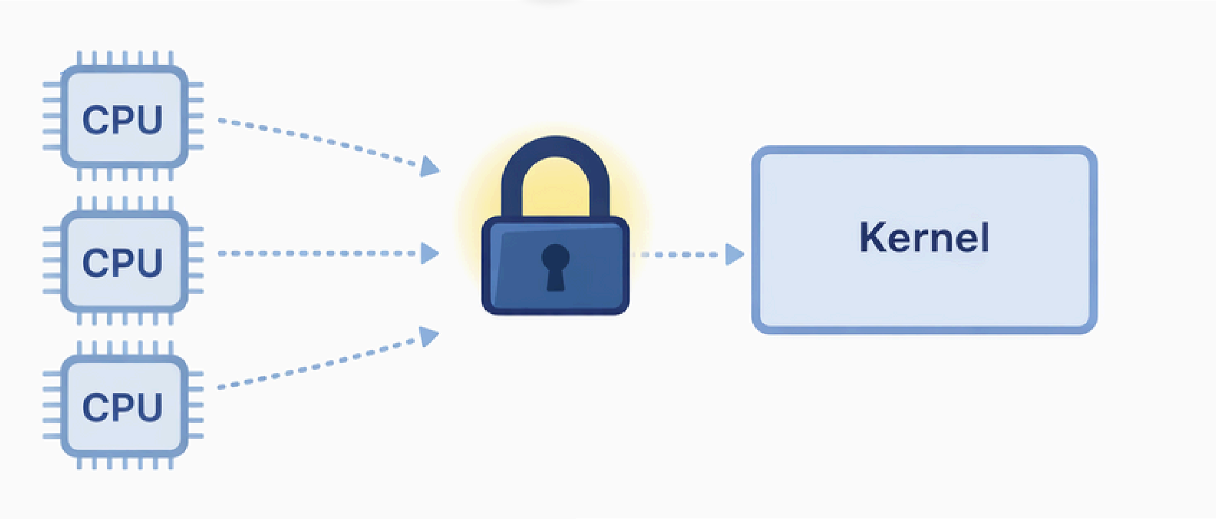
Load exclusif et Store exclusif

- ldxrb : enregistre l'adresse physique en tant qu'accès exclusif au cœur qui a load
- stxrb : store dans l’adresse physique à condition qu’elle soit enregistrée

```
.globl lock
lock:
    mrs x9, mpidr_el1
    and x9, x9, #0xFF           // x9 = core_id
    adrp x10, mutex
    add x10, x10, #:lo12:mutex

    // Check ownership
    ldrb w11, [x10]
    cmp w11, w9
    b.eq 1f

    // Spinloop
spinloop:
    ldxrb w11, [x10]
    cmp w11, #0xFF
    b.ne spinloop // Check mutex availability
    stxrb w11, w9, [x10]
    cbnz w11, spinloop // Check if memory access was exclus
    dmb sy

1:  ret
```

```
.globl unlock
unlock:
    mrs x9, mpidr_el1
    and x9, x9, #0xFF
    adrp x10, mutex
    add x10, x10, #:lo12:mutex

    ldrb w11, [x10]
    cmp w11, w9
    b.ne 1f

    dmb sy
    mov w11, #0xFF
    strb w11, [x10]

1:  ret

.section ".data"
.globl mutex
mutex :
  .byte 0xFF
```

Le Big Lock

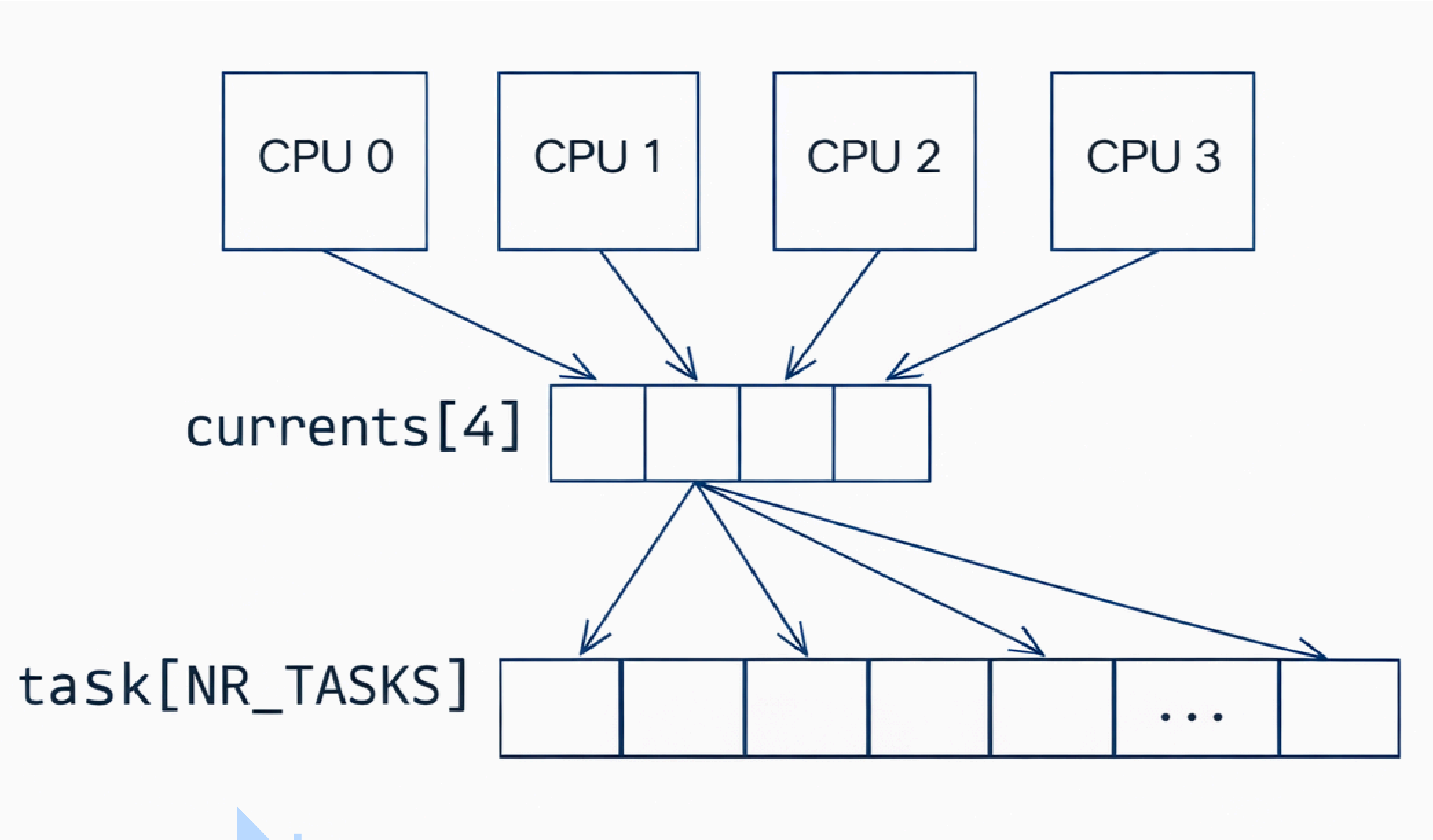


entry.S

```
el1_irq:
  kernel_entry 1
  bl lock    // KERNEL LOCK
  bl handle_irq
  bl unlock // KERNEL UNLOCK
  kernel_exit 1

el0_irq:
  kernel_entry 0
  bl lock    // KERNEL LOCK
  bl  handle_irq
  bl unlock // KERNEL UNLOCK
  kernel_exit 0

el0_sync:
  kernel_entry 0
  bl lock    // KERNEL LOCK
  ...

el0_svc:
  ...
ret_from_syscall:
  str x0, [sp, #S_X0]       // returned x0
  bl unlock // KERNEL UNLOCK
  kernel_exit 0
```

```
el0_da:
  ...
  bl unlock // KERNEL UNLOCK
  kernel_exit 0

.globl ret_from_fork
ret_from_fork:
  bl  schedule_tail
  bl unlock // KERNEL UNLOCK
  cbz x19, ret_to_user      // not a kernel thread
  mov x0, x20
  blr x19
ret_to_user:
//  bl unlock // Free the lock applied to kernel process
  bl disable_irq
  kernel_exit 0
```

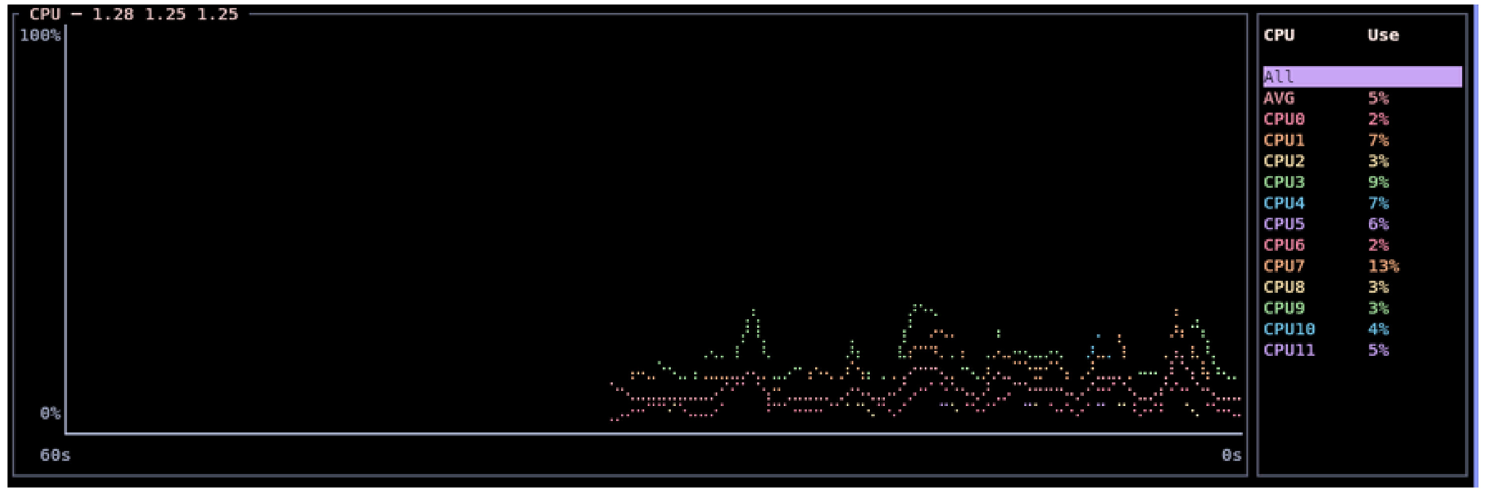
Nouveau Scheduler




```c
static struct task_struct init_task_0 = INIT_TASK;
static struct task_struct init_task_1 = INIT_TASK;
static struct task_struct init_task_2 = INIT_TASK;
static struct task_struct init_task_3 = INIT_TASK;
struct task_struct *currents[NB_CPU] = { &(init_task_0), &(init_task_1),
&(init_task_2), &(init_task_3)};
struct task_struct *task[NR_TASKS] = {
    &(init_task_0),
    &(init_task_1),
    &(init_task_2),
    &(init_task_3),
};
int nr_tasks = NB_CPU;
```

```c
struct task_struct {
    struct cpu_context cpu_context;
    long state;
    long counter;
    long priority;
    long preempt_count;
    unsigned long flags;
    struct mm_struct mm;
    unsigned char taken;
};
```

Méthode de test de la v1

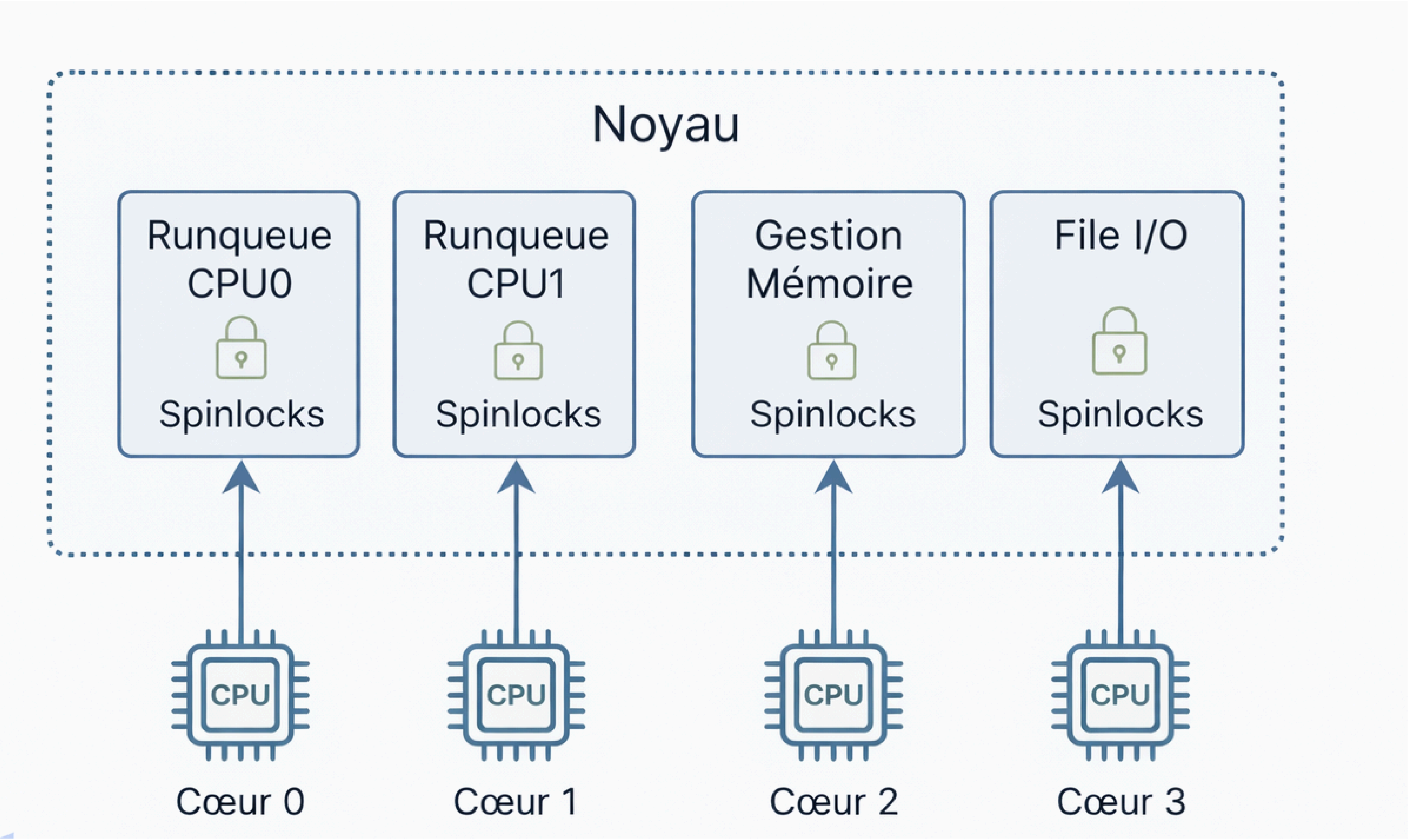


- Bottom

```c
void create_and_loop(char* str, char* arg)
{
  char buf[25] = {""};
  for (int i = 0; i < 25; i++){
    buf[i] = str[i];
  }
  call_sys_write(buf);
  int pid = call_sys_fork();
  if (pid < 0) {
    call_sys_write("Error during fork\n\r");
    call_sys_exit();
    return;
  }
  if (pid == 0)
    loop(arg);

  call_sys_write("Process over\n\r");
  call_sys_exit();
}
```

```c
void user_process()
{
  call_sys_write("User process\n\r");
  // First child
  call_sys_write("Creating process 1\n\r");
  int pid = call_sys_fork();
  if (pid < 0) {
    call_sys_write("Error during fork\n\r");
    call_sys_exit();
    return;
  }
  if (pid == 0)
    create_and_loop("Child 1 creating
child\n\r","azazazaz");

  // Second child ..
  // Third child..
  // Four child ..
  }
```

Principe des Small Spinlocks



Implémentation des Small Spinlocks

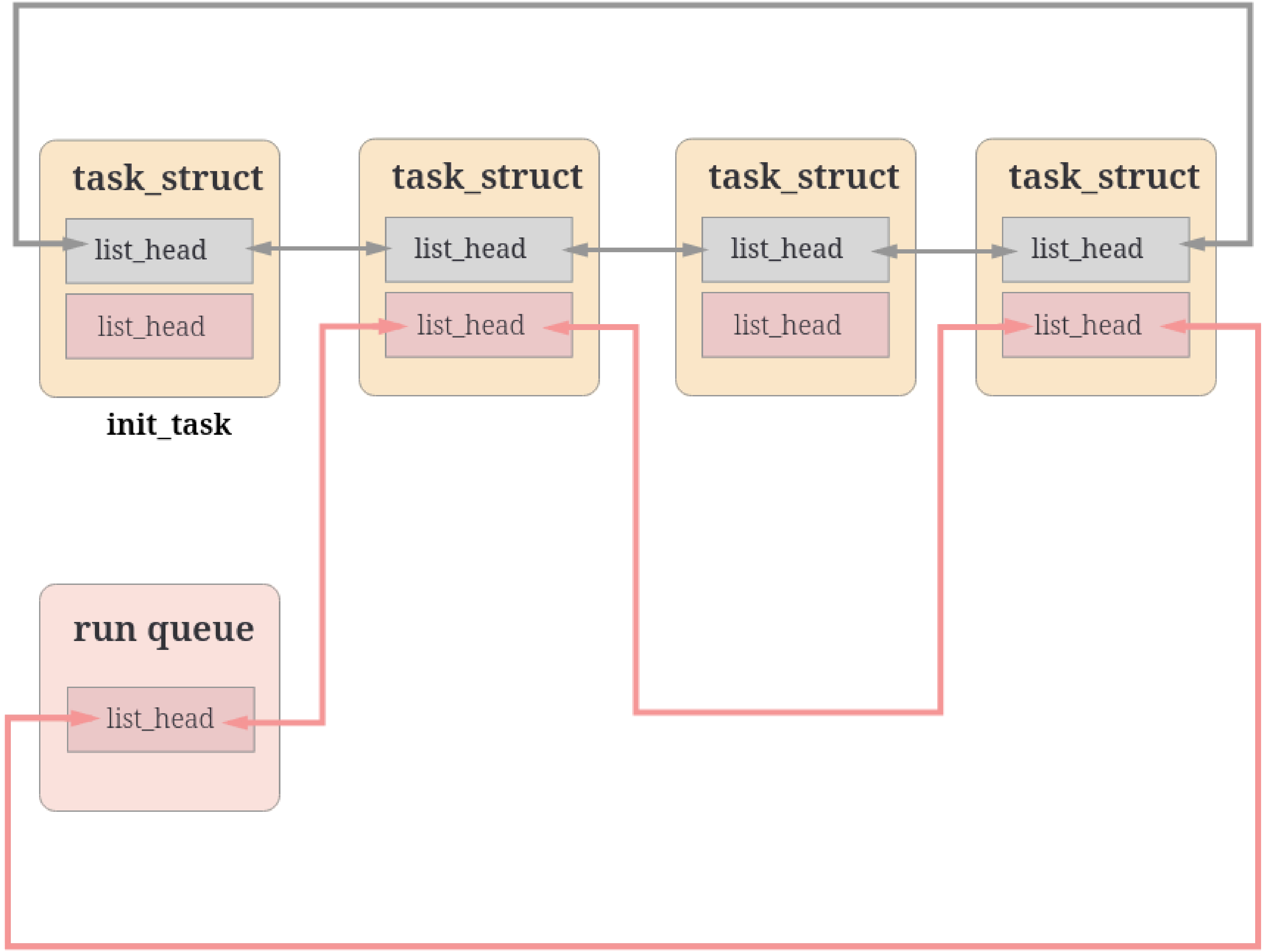
Nouvelles variables

```
.section ".data"
.globl mutex_write
mutex_write :
  .byte 0xFF
.globl mutex_shed
mutex_sched :
  .byte 0xFF
.globl mutex_mem
mutex_mem :
  .byte 0xFF
```

Addition atomique

```
.globl atomic_add
atomic_add:
  ldxr w2, [x0]
  add w2, w2, w1
  stxr w3, w2, [x0]
  cbnz w3, atomic_add
  dmb sy
  mov w0, w2
  ret
```

Optimisation : Scheduling per-cpu



Conclusion

- Alternatives:
  - CAS
  - WFE
- Axes d’amélioration:
  - équilibrage de charge (load balancing)
  - migration dynamique des tâches entre cœurs