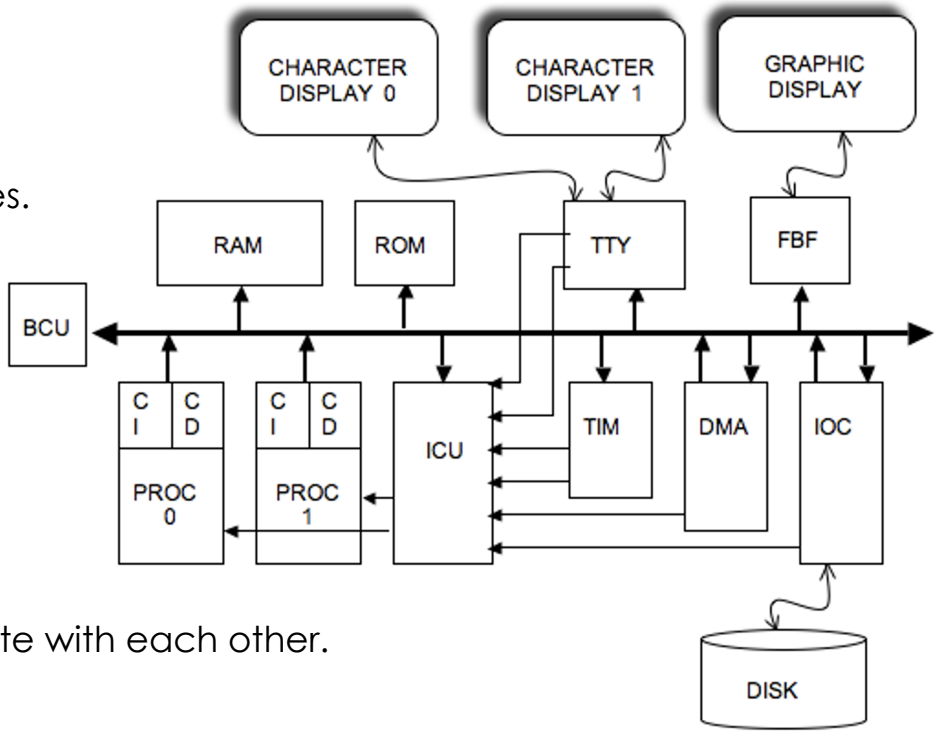
multi-tasking cooperative

MU4IN106 Multi

Multi-core platform

This platform can perform at least as many tasks as there are cores.



We will see how they can communicate with each other.

Problems

- Tasks will share data
- Each task can be a data producer or consumer, i.e. one task can produce data that another task will consume.
- It is not possible to know the execution time of the tasks because it often depends on the processed data, but even worse, it depends on the allocation of resources (bus, peripherals, memory, etc.) and cache misses.
- This means that it is not possible to synchronize tasks and that the communication mechanisms must accept this asynchronous behavior.

Atomic operation

Multi-core architectures must offer atomic operations:
→ read - modify - write

- If the architecture uses a bus, we can use the arbiter to keep the bus for several transactions but this is penalizing since it prohibits all transactions during this time and it is only possible if there is a bus.
- Processors therefore offer instructions for atomic operations.
- It is the memory controller that will guarantee the atomicity

Solutions for the lock

All the sharing mechanisms must guarantee that a memory cell has only one writer at each moment, otherwise the **hardware** must propose a mechanism allowing the atomic sequence **read** of a word - **modification** of its value - **writing** of its new value, a read-modify-write sequence

> *It is a problem of **consensus** between the threads which must agree on the value of the lock on the value of the lock and have the same vision. In the general case, there are solutions allowing participants to have a consensus on the states of a system even though these participants are unreliable and exchange messages on an unreliable network! (e.g. https://www.wikiwand.com/en/Raft_(algorithm)) (useless here but interesting too :-)*

In an SoC, the conditions are very simplified because the system is reliable (the processor, the system bus, the memories and the kernel code are reliable).

We will see 4 possible mechanisms and we will use the last one of this list:

1. Hardware lock memory
2. test-and-set instruction
3. Instruction compare-and-swap
4. Instruction pair **`ll / sc`** (load Link / store conditional)

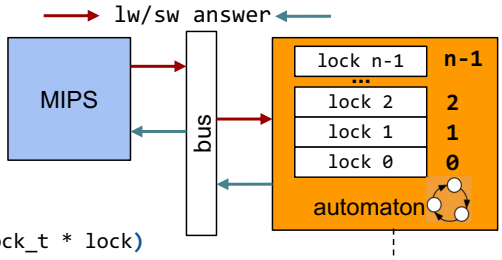
4 Hardware solutions for locks

1. Lock memory

Memory whose reading causes a writing

`lw $rt, imm($rs)` reads the value of the word at address `$rs+imm`
if its value is 0 then the word is set to 1

`sw $rt, imm($rs)` writes the value of $r to the address `$rs+imm`




```
void spin_lock (spinlock_t * lock)
spin_lock: lw $8, ($4)
           beqz $8, spin_lock
           jr $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw $0, ($4)
             jr $31
```

Useful if the processor does not offer atomic instructions

2. Test And Set (*this is not a MIPS instruction*)

`TAS $rt, imm($rs)`

writes the value of `$rt` to the word at `$addr+imm` and returns the old value

```
void spin_lock (spinlock_t * lock)
spin_lock: li $8, 1
           TAS $8, ($4)
           bnez $8, spin_lock
           jr $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw $0, ($4)
             jr $31
```

To take a lock, we try to write 1 until we succeed. We know this because the value returned by **heap** is then 0.

4 Hardware solutions for locks

3. Compare And Swap (*Attention this is not a MIPS instruction*)

`cas $old, $new, imm($addr)`

compare the value of `$old` with the value of the word at `$addr+imm`
if they are equal, then write `$new`

To take a lock, we try to write 1 until we succeed. We know this because the value returned by **case** is then 0

```
void spin_lock (spinlock_t * lock)
spin_lock: li $8, 0
           li $9, 1
           cas $8, $9, ($4)
           bnez $8, spin_lock
           jr $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw $0, ($4)
                    jr $31
```

Compare And Swap allows to make atomic counters

```
void atomic_add (int * counter, int val)
atomic_add: lw $8, ($4)
            addu $9, $8, $5
                      case $8, $9, ($4)
            bnez $8, atomic_add
            jr $31
```

ABA problem

If we have a variable **var** shared by several threads can take the values **A** and **B**

1. If a thread **T** reads the value **A**, then
2. tries to write the value **B** with **CAS A,B,(var)**

If it succeeds in doing so by returning the old value **A**, it may be that **var** has passed through the value **B** and then returned the value **A**

The thread **T** can't know it, it can be a problem, e.g. for atomic counters, if the number of counter values is small...

4 Hardware solutions for locks

4. Load Link / Store Conditional (*Solution present in MIPS*)

`ll $rt, imm($rs)`

reads in `$rt` the value of the word present at address $rs+imm
It's like an `lw` **but** the memory notes that an `ll` is in progress at this address (`link`)
The memory opens a "reservation" at this address (`ll` is also called Load Reserved)

`sc $rt, imm($rt)`

attempts to write `$rt` to the address $addr+imm
The write is accepted only if there is an open reservation for this address
But the reservation is closed by all memory instructions other than `ll`
If the blind has worked `$rt` contains 1 otherwise it contains 0

```
void spin_lock (spinlock_t * lock)
spin_lock_delay:
     addiu $8, $8, -1
     bnez $8, spin_lock_delay
spin_lock:
    ll $2, ($4)
    li $8, 50
    bnez $2, spin_lock_delay
spin_lock_sc:
    li $2, 1
    sc $2, ($4)
    beqz $2, spin_lock
    jr $31
```

```
void spin_unlock (spinlock_t * lock)
spin_unlock: sw $0, ($4)
                    jr $31
```

The `LL` / `SC` solution does not have the ABA problem because the memory guarantees that no operation takes place between a successful `ll` and `sc`, `LL` / `SC` can simulate a CAS and not the other way around
BUT it is not free, the memory must make reservations

Dead Lock

- If the tasks need to take several resources to work, there is a risk of dead lock (fatal embrace)
- Let A and B be two locks that protect two resources

T1	T2
takes A	
	takes B
wait B	
	wait A

- It is necessary to
  - to define a global order of resources
  - or a timeout mechanism if it is not possible to define an order

Toggle set / reset

Communication mechanism between a producer and a consumer

- The producer must inform that information is available
- The consumer must wait for the information

Description

- BUF : buffer for data
- STATE : boolean which defines the owner of the buffer
  - 0 the buffer belongs to the producer who can write data
  - 1 the buffer belongs to the consumer who can read the data

Usage

Producer

- while (STATE == 1);
- BUF ← data
- STATE ← 1

Consumer

- while (STATE == 0);
- get BUF
- STATE ← 1

Barrier

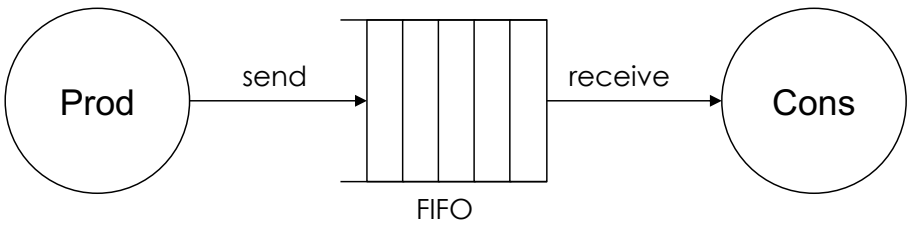
Allows several tasks to meet after the execution of a calculation step, two functions are needed:

- barrier_init (N) → N is the number of tasks that meet
- barrier_wait() → executed to make an appointment
  - all tasks are put on hold
  - except the last one to arrive, which wakes up the others
    and continues its execution

Requires an atomic increment

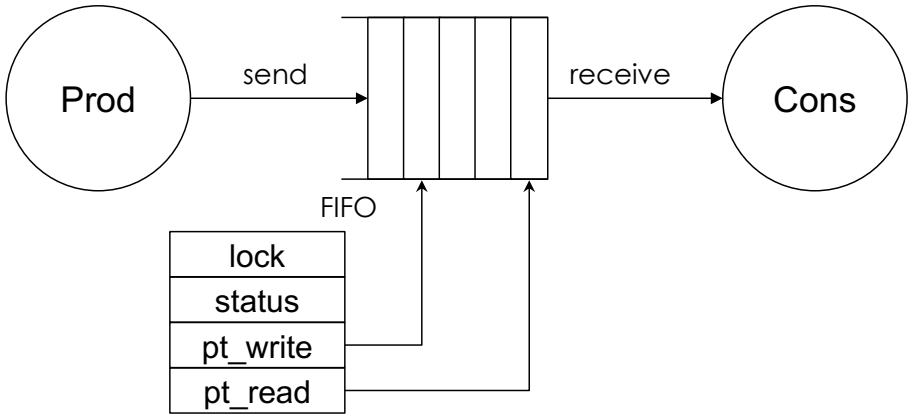
Channeled communication

- To work together, two tasks can use a send/receive communication model



- This is a model that does not require shared memory, but is implementable on a shared memory machine.

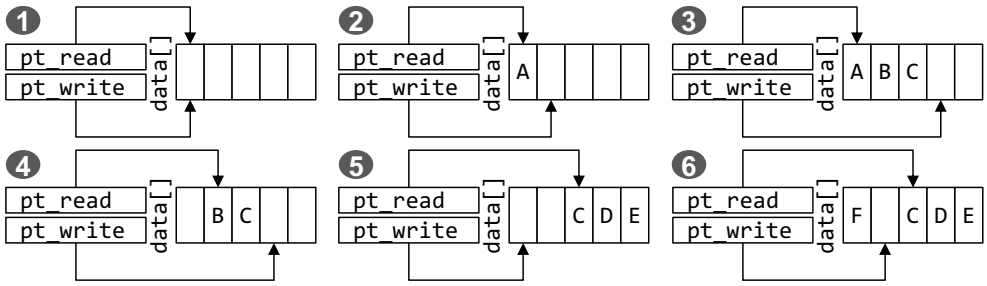
Channeled communication



1. take the lock
2. test status
3. if the transfer is impossible,
   release the lock and return to 1.
4. if the transfer is possible,
   Make the transfer, increment the concerned pointer,
   change status, then release the lock.

FIFO

Depending on the behavior of **send**()
we can have
a FIFO with
or without loss




```
struct tty_fifo_s {
        char data [20];
        int pt_read;
        int pt_write;
};

static int send (struct tty_fifo_s *fifo, int c) {
        int pt_write_next = (fifo->pt_write + 1) % sizeof(fifo->data);
        if (pt_write_next != fifo->pt_read) {
                fifo->data [fifo->pt_write] = c;
                fifo->pt_write = pt_write_next;
                return 1;
        }
        return 0;
}
```

```
static int receive (struct tty_fifo_s *fifo, int *c) {
        if (fifo->pt_read != fifo->pt_write) {
                *c = fifo->data [fifo->pt_read];
                fifo->pt_read = (fifo->pt_read + 1)
                        % sizeof(fifo->data);
                return 1;
        }
        return 0;
}
```

FIFO MWMR

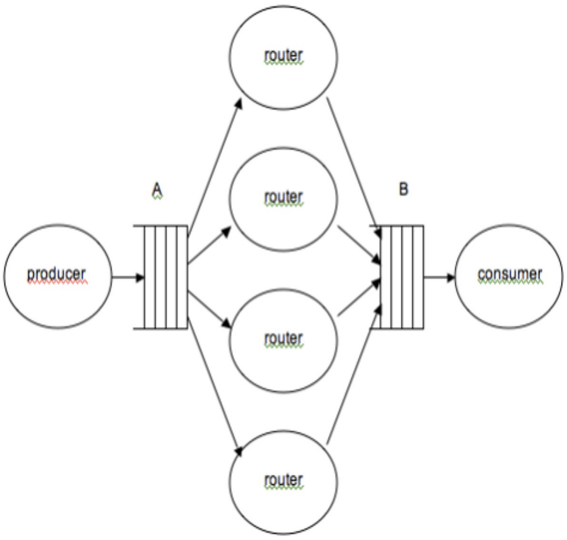
The model can be generalized to several producers and several consumers.

A FIFO is defined by

- a lock
- a reading pointer
- a writing pointer
- a number of data present
- a number of paces

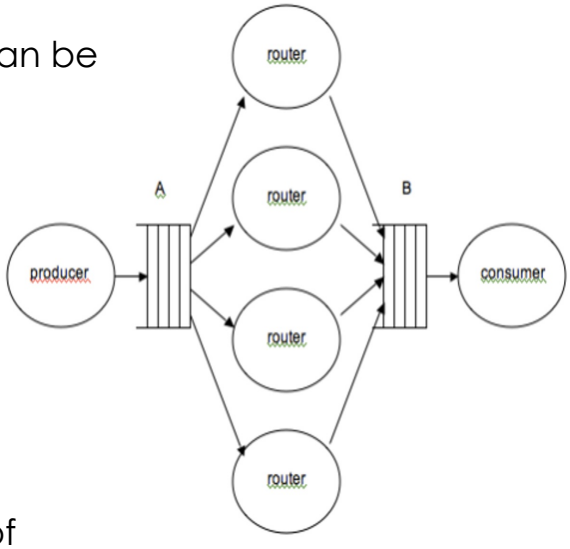
The 5 steps of send / receive operations

- take the lock
- is there any space or data
- if no, go out, if yes, make the transfer
- move the pointers
- release the lock



FIFO MWMR

- The producing and consuming tasks can be software or hardware
  - useful for a device or a coprocessor
- We can use 2 locks
  - for reading and writing but you can no longer use status to know the status of the FIFO
  - It is necessary to use the pointers of reading and writing pointers.



In TME

- You will experience several mechanisms of synchronization of communicating tasks
- You will experience problems related to the instruction reordering performed by GCC