Föreläsning 9 IS1300 Inbyggda system

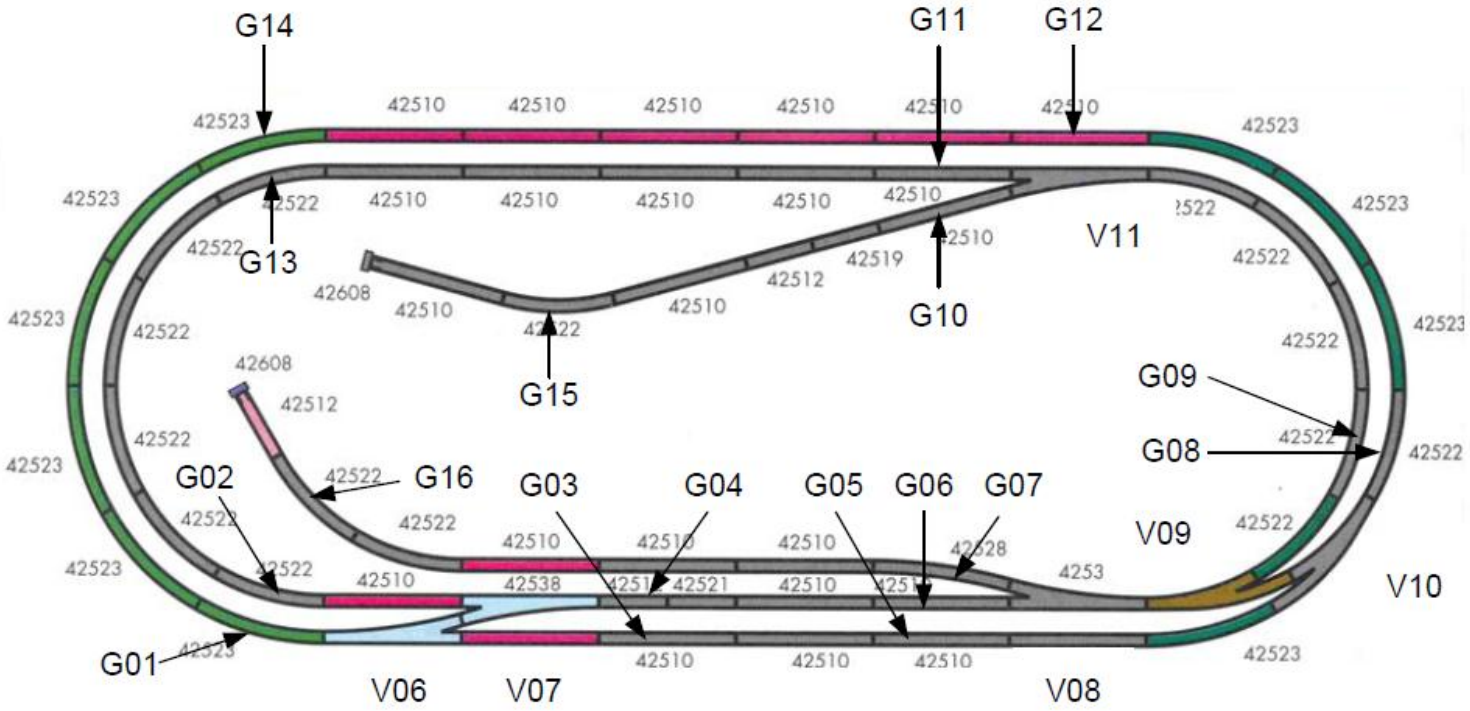
- Real Time Operating System
  - Communication principles
  - µC/OSII (used in lab exercise)

Interprocess communication mechanisms

- Blocking / Nonblocking
- Shared memory and Message passing
- Events
  - Semaphores
  - Message Mailboxes
  - Message Queues

Real time lab exercise

16 givare (G01-G16) som känner när lok passerar
5 växlar (V06-V11) som kan styras
Lokets hastighet kan styras
Två lok skall kunna köras samtidigt enligt given bana utan att kollidera



Avoiding Interference

- The parts of a process that access shared variables must be executed indivisibly with respect to each other
- These parts are called critical sections
- The required protection is called mutual exclusion

Mutual Exclusion

- In computer science, mutual exclusion refers to the problem of ensuring that no two processes or threads (henceforth referred to only as processes) are in their critical section at the same time.
- Here, a critical section refers to a period of time when the process accesses a shared resource, such as shared memory.

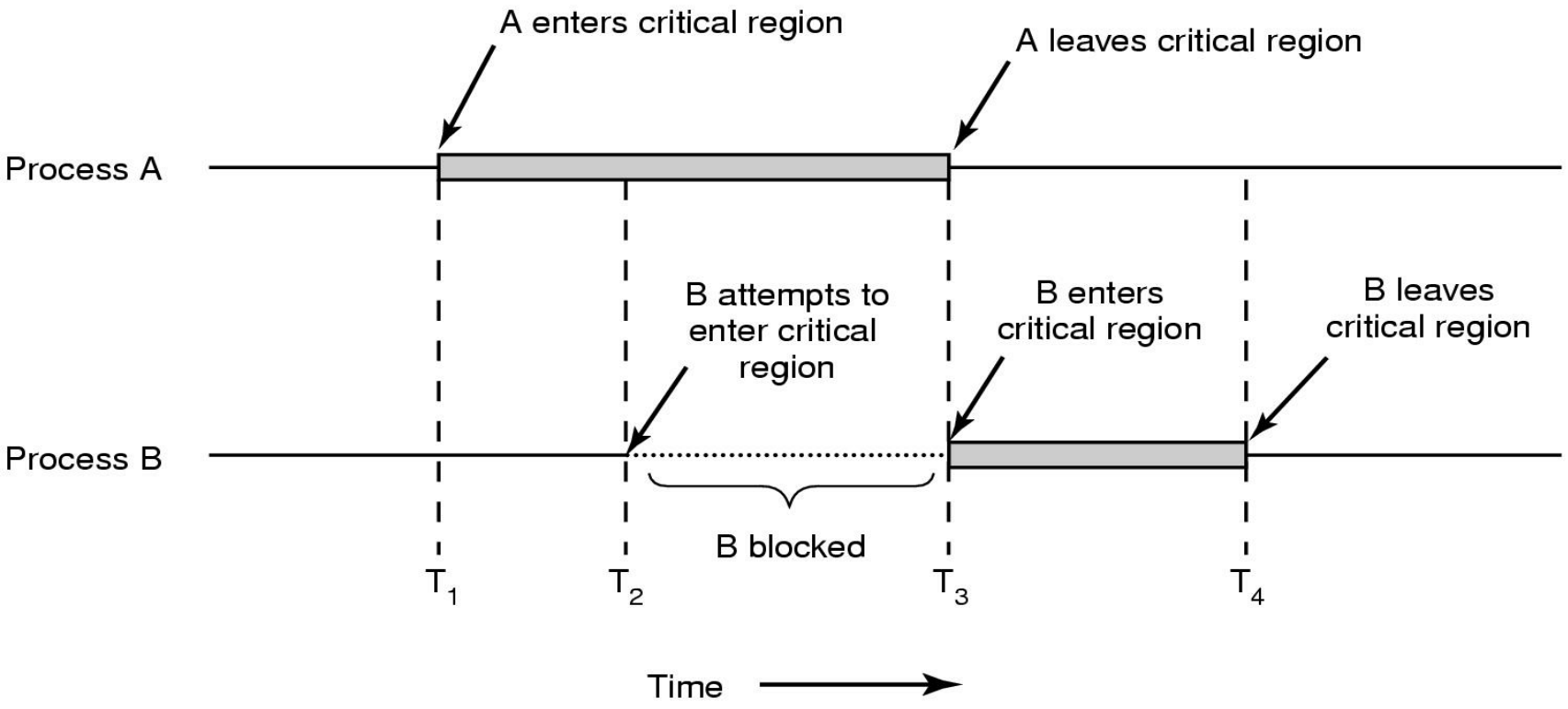
http://en.wikipedia.org/wiki/Mutual_exclusion

Critical Section

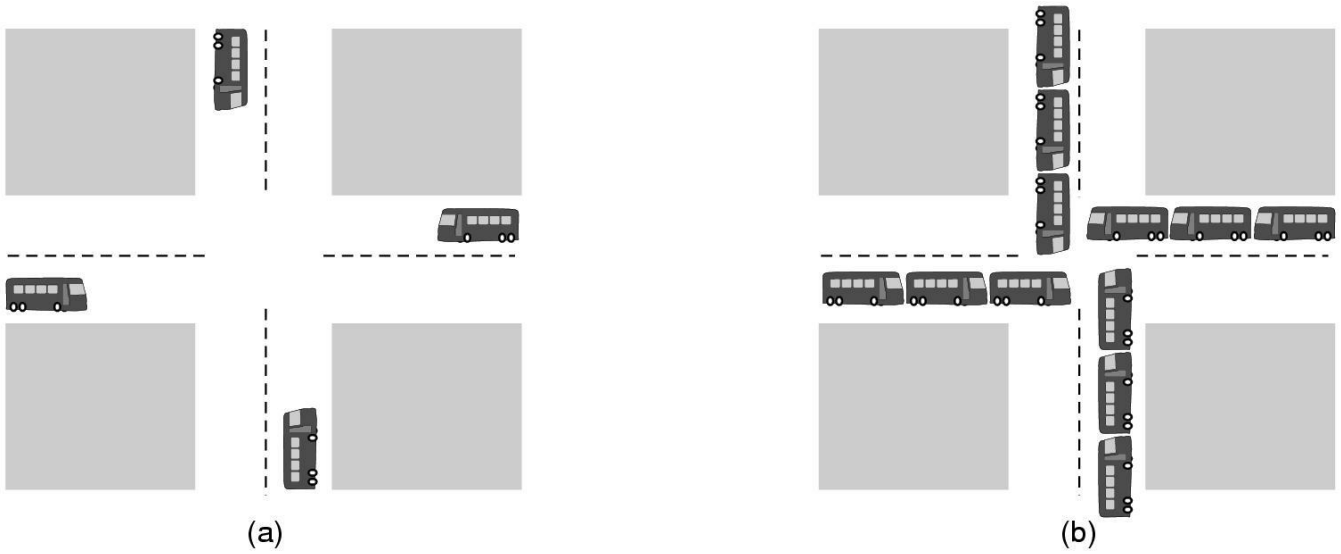
- No two processes may be simultaneously inside their critical regions.
- No processes running outside its critical region may block other processes
- No process should have to wait forever to enter its critical region.

http://en.wikipedia.org/wiki/Critical_section

Cricital Section



Deadlock



Circular Wait causes deadlock

Deadlock

```
typeT buffer;
sem empty = n;
sem full = 0;
sem mutex = 1;


process Producer
{
  while(true)
  {
    //producera data
    wait(mutex);
    wait(empty); //fel ordning
    insert(data);
    wait(mutex);
    signal(full);
  }
}
```

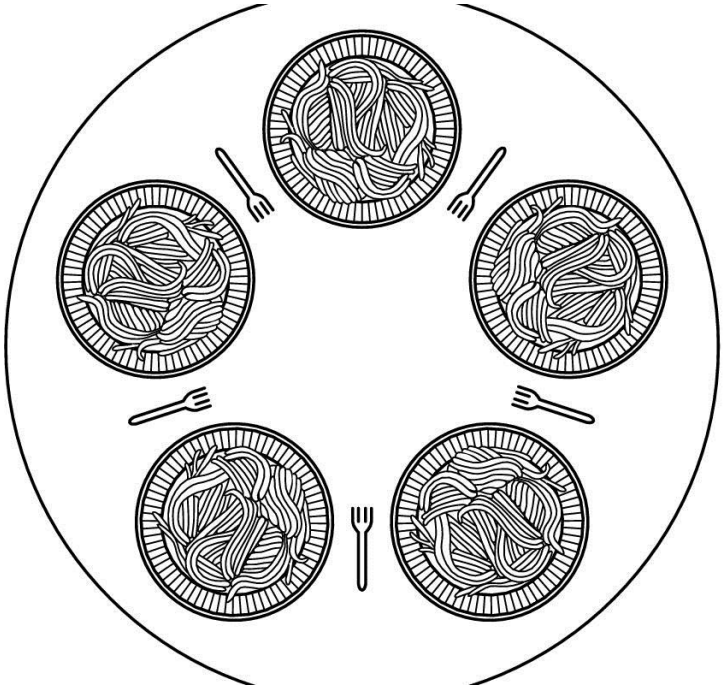
Deadlock if the buffer is full

```
process Consumer
{
  while(true)
  {
    //hämta data

    wait(full);
    wait(mutex);
    data = remove();
    wait(mutex);
    signal(empty);
  }
}
```

Dining Philosophers

- Philosophers either eat or thinks.
- A philosopher needs two forks to be able to eat the spaghetti.
- When a philosopher gets hungry, she tries to acquiring her left and right fork, one at a time.
- How do you avoid deadlock or starvation?



Dining Philosophers

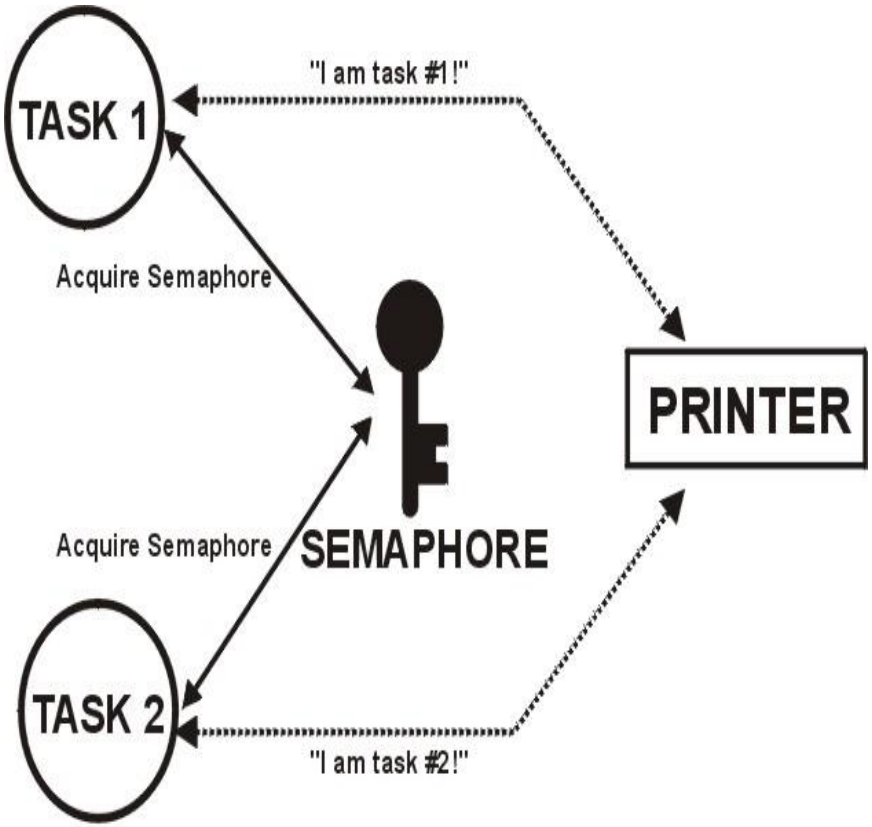
```
sem fork[5] = {1, 1, 1, 1, 1};

//i=0 to 3
process Philospher[i]
{
  while(true)
  {
    wait(fork[i]);//get left
    wait(fork[i+1]);//get right
    //eat;
    signal(fork[i]);
    signal(fork[i+1]);
    //think;
  }
}
```

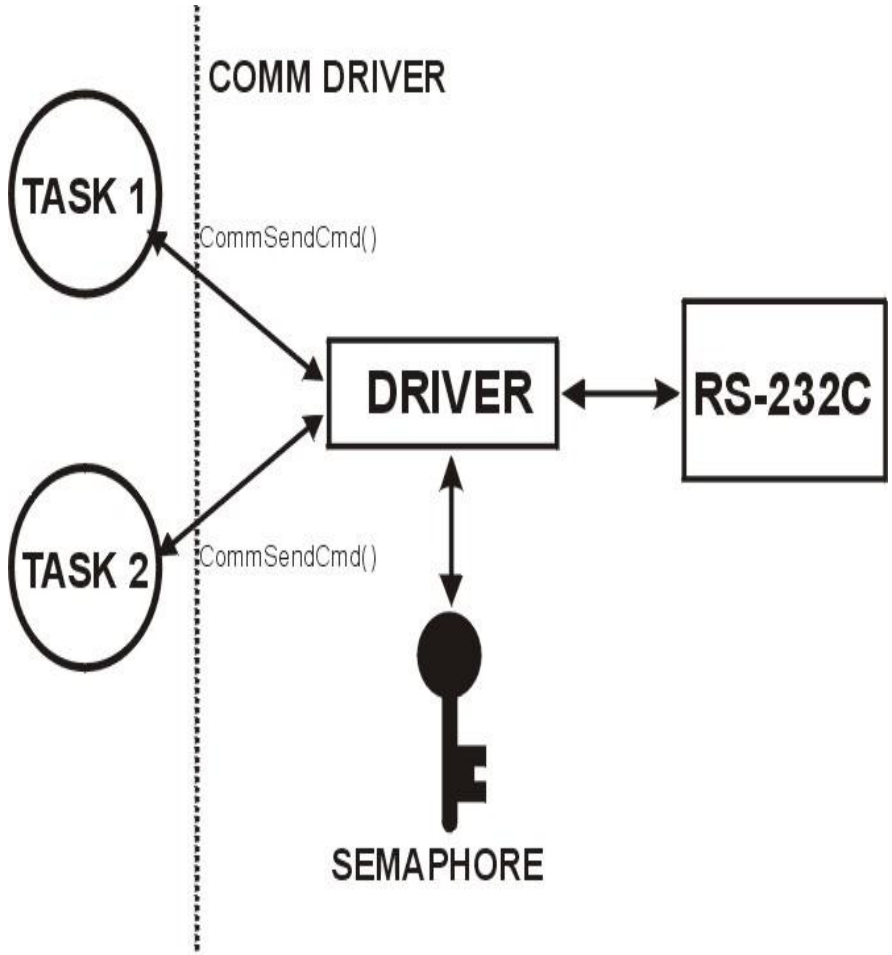
```
process Philospher[4]
{
  while(true)
  {
    wait(fork[0]);//get right fork
    wait(fork[4]);// then left fork
    //eat;
    signal(fork[0]);
    signal(fork[4]);
    //think
  }
}
```

http://en.wikipedia.org/wiki/Dining_philosophers_problem

Semaphores

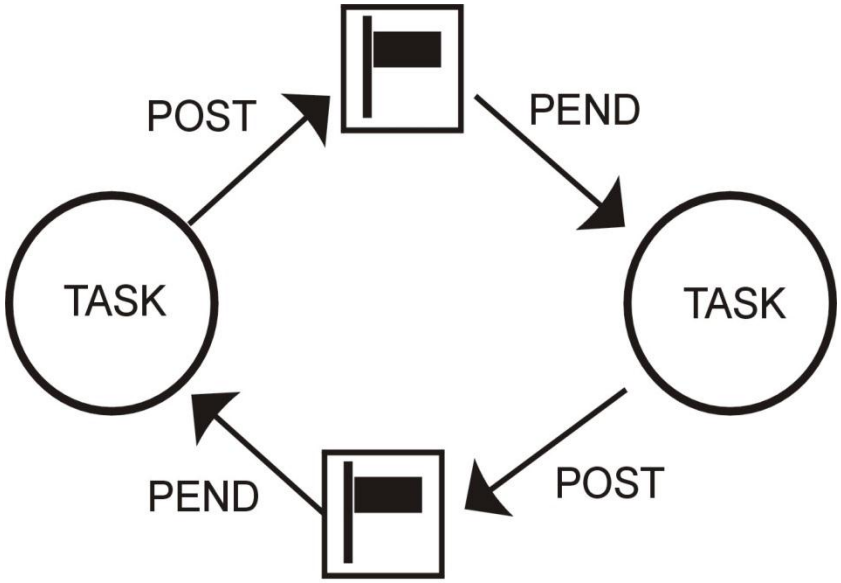


Two tasks competing for a printer

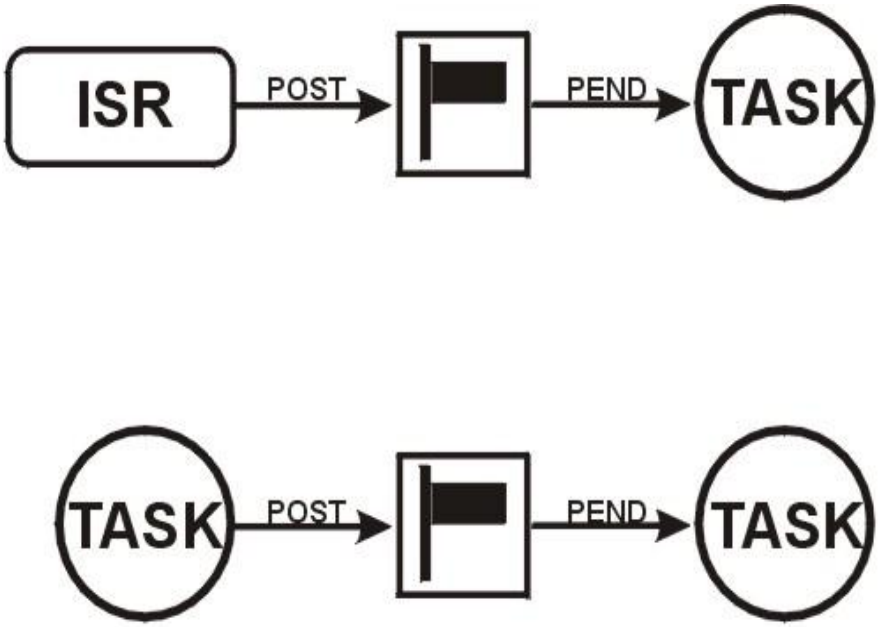


Comm Driver Example

Semaphores

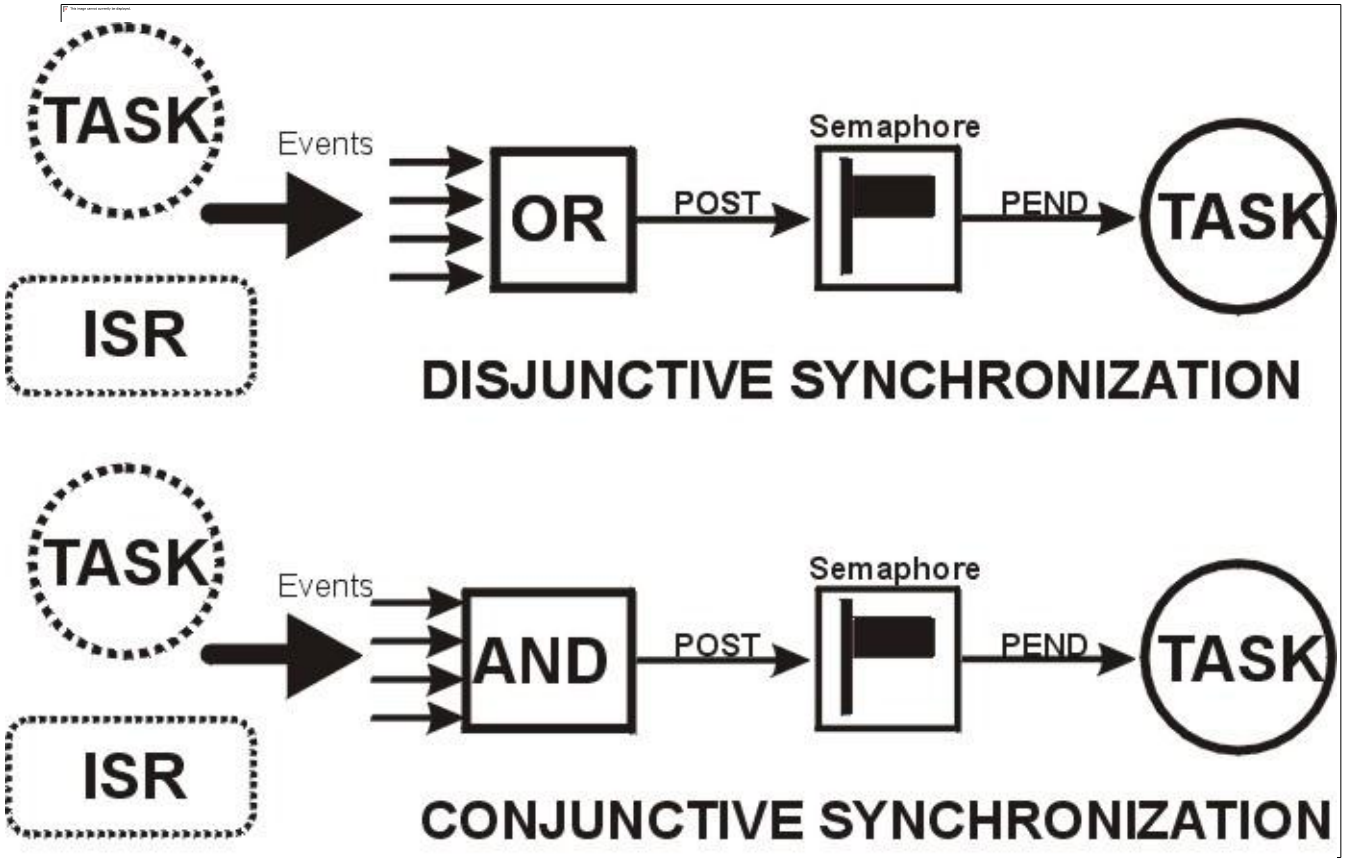


Synchronizing two Tasks



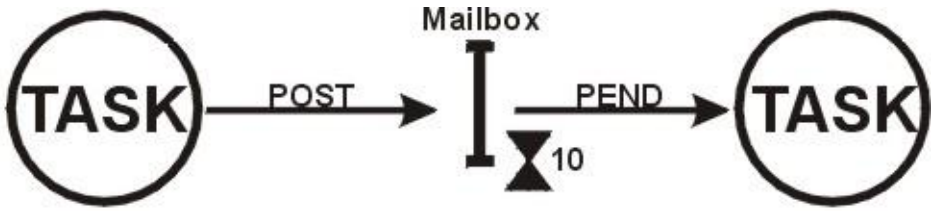
Signaling Events through Semaphores

Semaphores



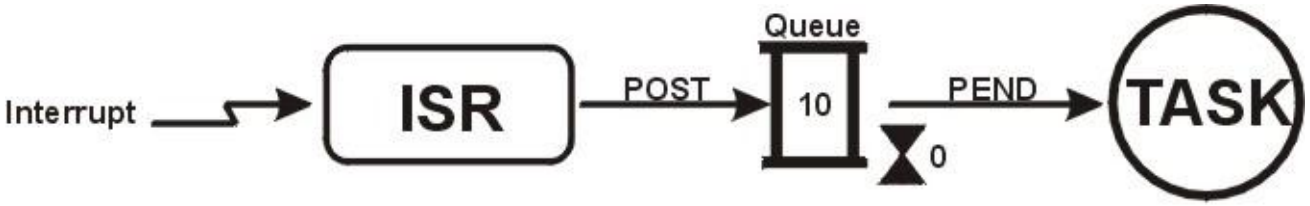
Disjunctive and Conjunctive Synchronization

Mailbox and Queue



Note: POST deposites a pointer size variable in the mailbox

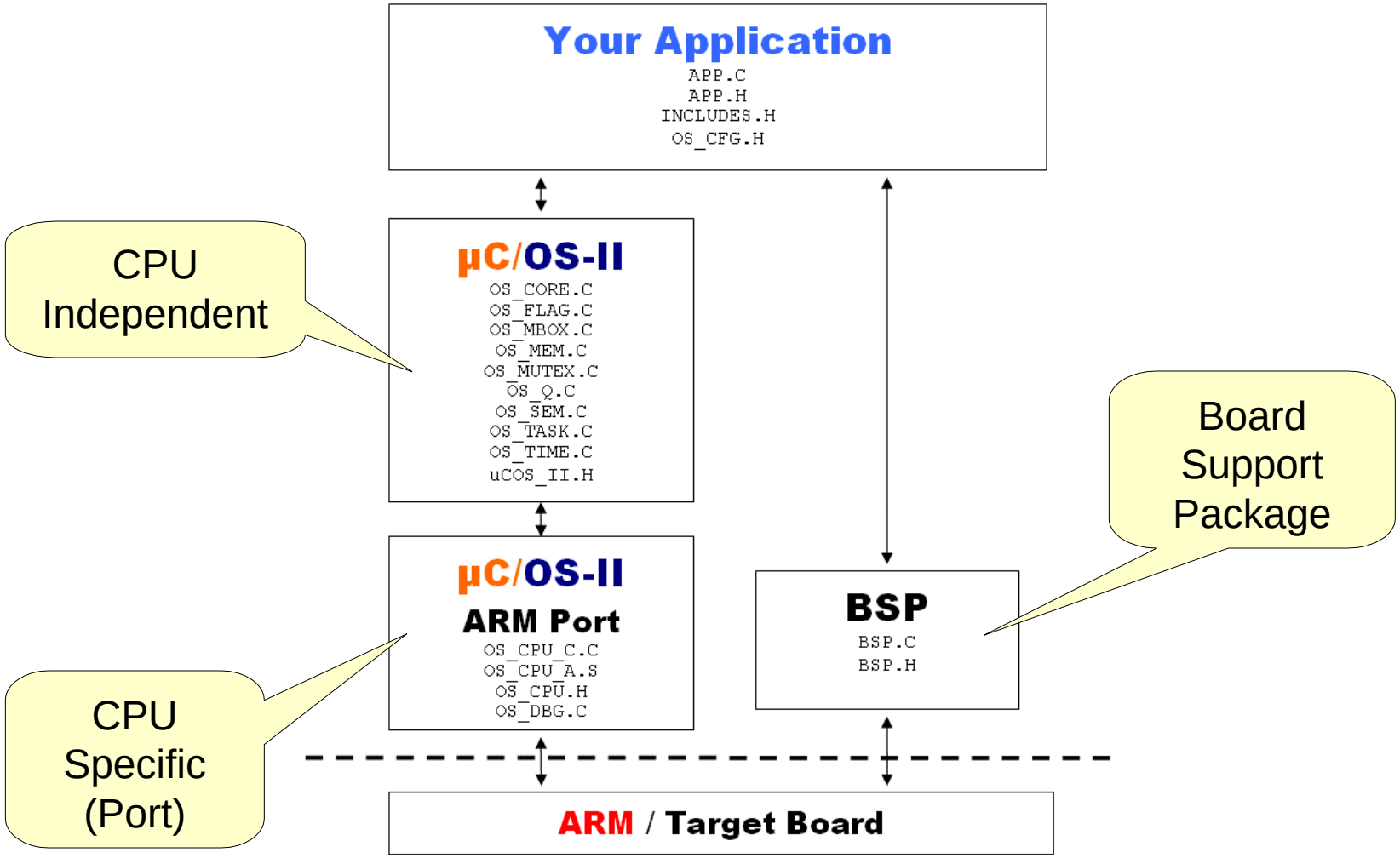
Message Mailbox



Note: POST deposites a pointer size variable in the queue

Message Queue

Micrium µC/OS-II and Related Files



http://en.wikipedia.org/wiki/UC/OS

RTOS Tasks

- A task is a simple program that thinks it has the CPU all to itself
- Each Task has
  - Its own stack space
  - A priority based on its importance
- A task contains YOUR application code

RTOS Tasks

- A task is a simple program that thinks it has the CPU all to itself
- Each Task has
  - Its own stack space
  - A priority based on its importance
- A task contains YOUR application code

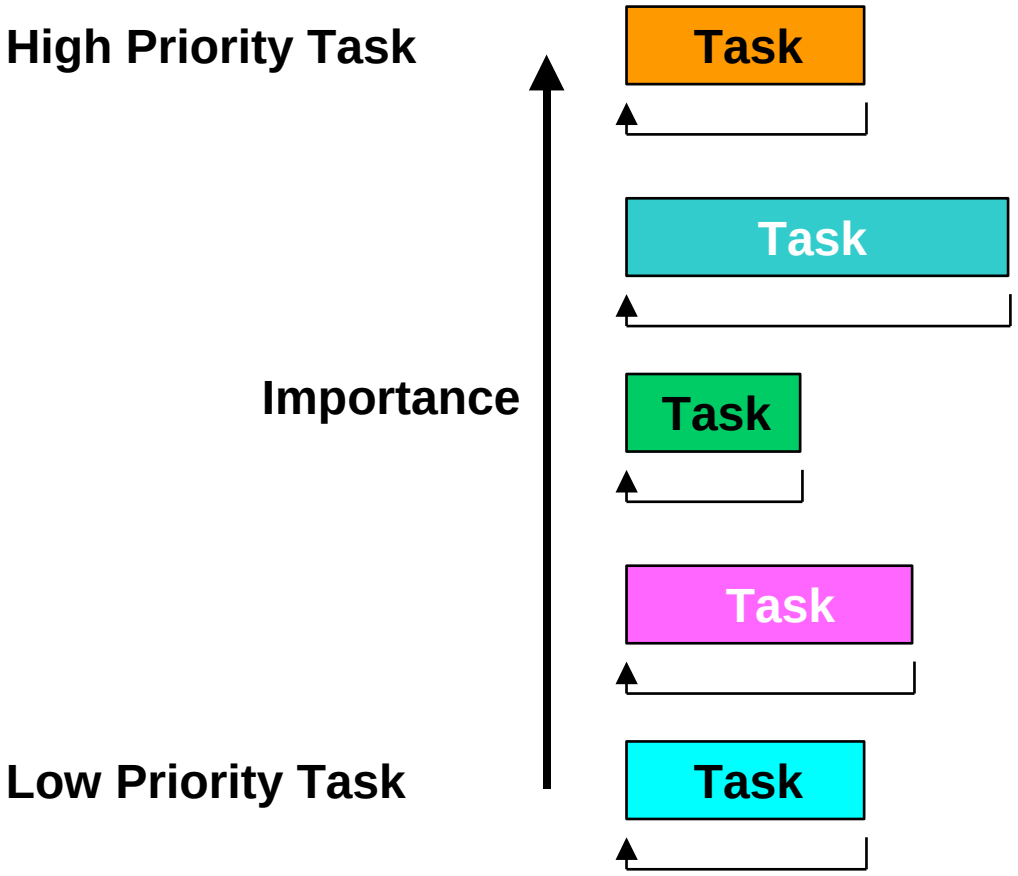
What is a Task?

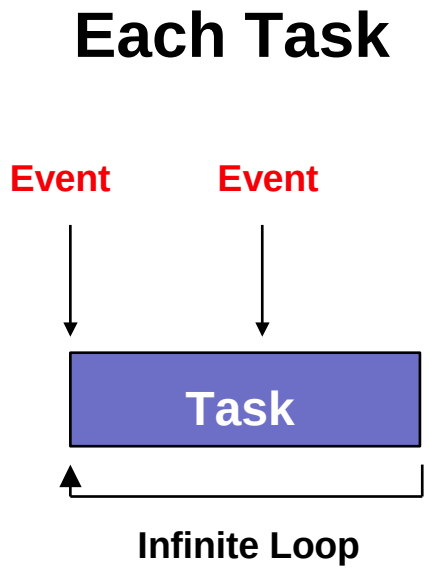
- A task is an infinite loop

```
void Task (void *p_arg)
{
   Do something with 'argument' p_arg;
   Task initialization;
   for (;;) {
      /* Processing (Your Code)                              */
      Wait for event;                          /* Time to expire ...     */
                                  /* Signal from ISR ...   */
                                  /* Signal from task ...  */
      /* Processing (Your Code)                              */
   }
}
```

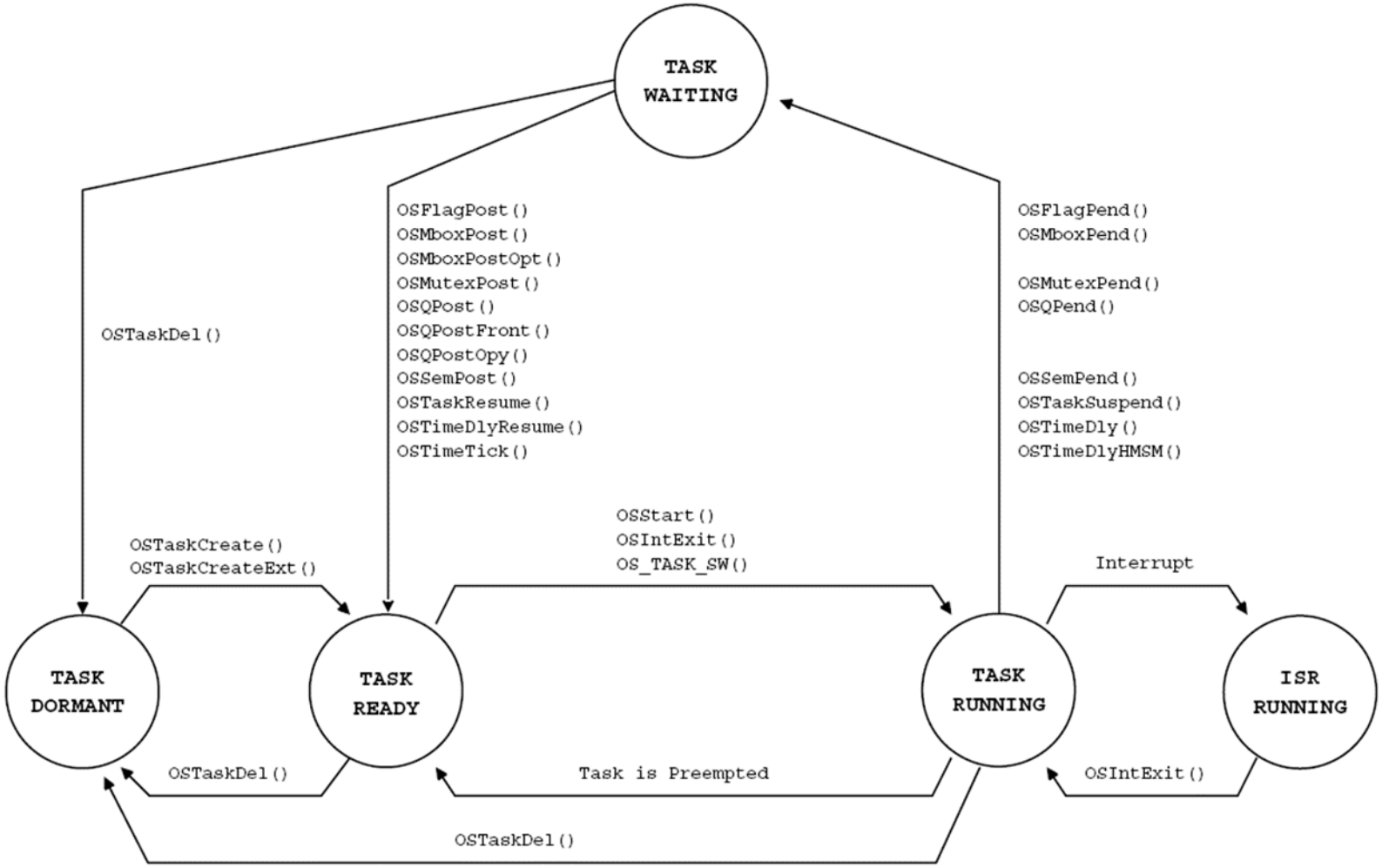
Designing with µC/OS-II

Splitting an application into Tasks





µC/OS-II Task States



Why Create a Task?

- To make it ready for multitasking
- The kernel needs to have information about your task
  - Its starting address
  - Its top-of-stack (TOS)
  - Its priority
  - Arguments passed to the task
  - Other information about your task

Creating a task with µC/OS-II

```
OSTaskCreateExt(void  (*task)(void *parg),
                void   *parg,
                OS_STK *pstk,
                INT8U   prio,
                INT16U  id,
                OS_STK *pbos,
                INT32U  stk_size,
                void   *pext,
                INT16U  opt);
```

Initializing µC/OS-II

Execution Path