ECE3411 – Fall 2015

Lecture 6c.

Real Time Operating System: Inter-Process Communication (IPC)

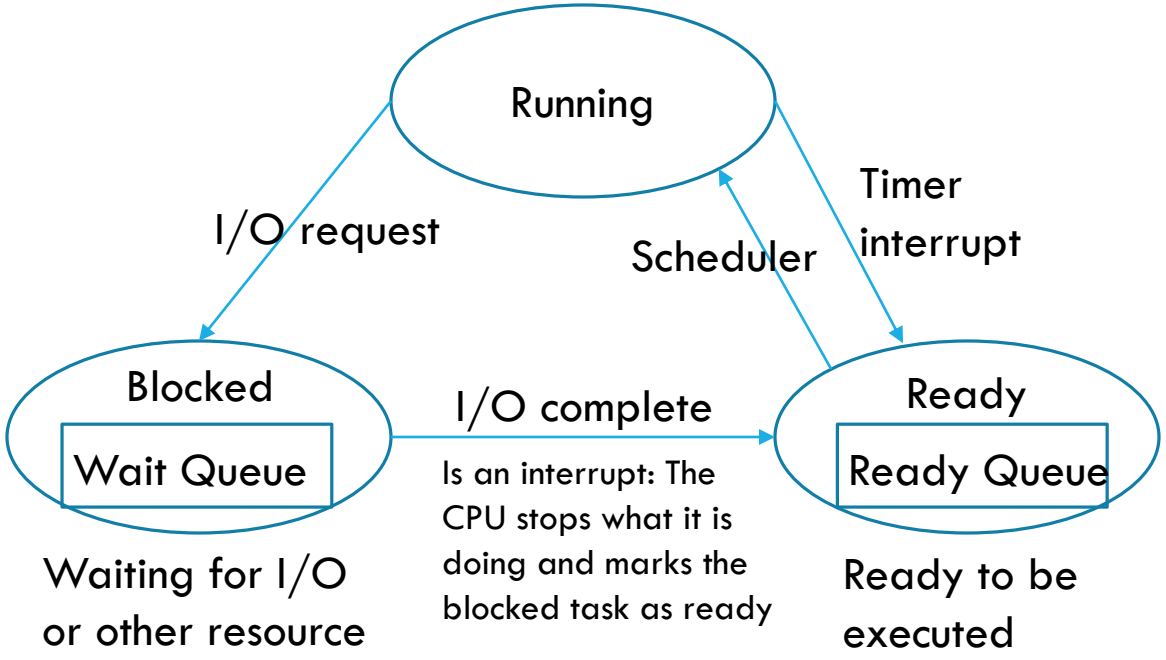
Marten van Dijk, Syed Kamran Haider
Department of Electrical & Computer Engineering
University of Connecticut
Email: {vandijk, syed.haider}@engr.uconn.edu

UCONN

These slides are extracted or copied from MIT 16.07 (Smith) and "Embedded Software Architecture", Cook & Freudenberg, 2008.

Task State Diagram

- A task/process goes through several states during its life in a multitasking system
- Tasks are moved from one state to another in response to the stimuli marked on the arrows



- Any tasks that are ready to run sit on the *ready queue*. This queue may be prioritized so the most important task runs next.
- When the scheduler decides the current task has had enough time on the CPU, either because it finished or its time slice is up, the *Running* task is moved to the ready queue. Then the first task on the ready queue is selected for Running.
- If the Running task needs I/O or needs a resource that is currently unavailable, it is put on the *blocked queue*. When its resource becomes available, it goes back to Ready.

Inter-Task Communication

- Tasks don't work in isolation from each other. They often need to share data or modify it in series.
- Since only one task can be running at one time, there must be mechanisms for tasks to communicate with one another
  - A task is reading data from a sensor at 15 hz. It stores 1024 bytes of data and then needs to signal a processing task to take and process the data so it has room to write more.
  - A task is determining the state of a system- i.e. Normal Mode, Urgent Mode, Sleeping, Disabled. It needs to inform all other tasks in the system of a change in status.
  - A user is communicating to another user across a network. The network receive task has to deliver messages to the terminal program, and the terminal program has to deliver messages to the network transmit task.

Inter-Task Communication

- Regular operating systems have many options for passing messages between processes, but most involve significant *overhead* and aren't deterministic.
  - *Pipes* (is a connection between two processes, such that the standard output from one process becomes the standard input of another process; the system temporarily holds the piped information until it is read by the receiving process),
  - *message queues* (asynchronous communications protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time; messages placed onto the queue are stored until the recipient retrieves them),
  - *Semaphores* (is a variable or abstract data type that is used for controlling access, by multiple processes, to a common resource in a concurrent system such as a multiprogramming operating system),
  - *Remote Procedure Calls* (is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details),
  - *Sockets* (is one endpoint of a two-way communication link between two programs running on the network, a socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to, an endpoint is a combination of an IP address and a port number),
  - *Datagrams* (a self-contained, independent entity of data carrying sufficient information to be routed from the source to the destination computer without reliance on earlier exchanges between this source and destination computer and the transporting network), *etc.*
- In a RTOS, tasks generally have direct access to a common memory space, and the fastest way to share data is by sharing memory.
  - In ordinary OS's, tasks are usually prevented from accessing another task's memory, and for good reason.

Shared Memory: Global Variables used as Flags

- Different tasks are spawned and each, when finished, increments a variable called *finished*; once *finished* is equal to the total number of tasks spawned, the computation is done

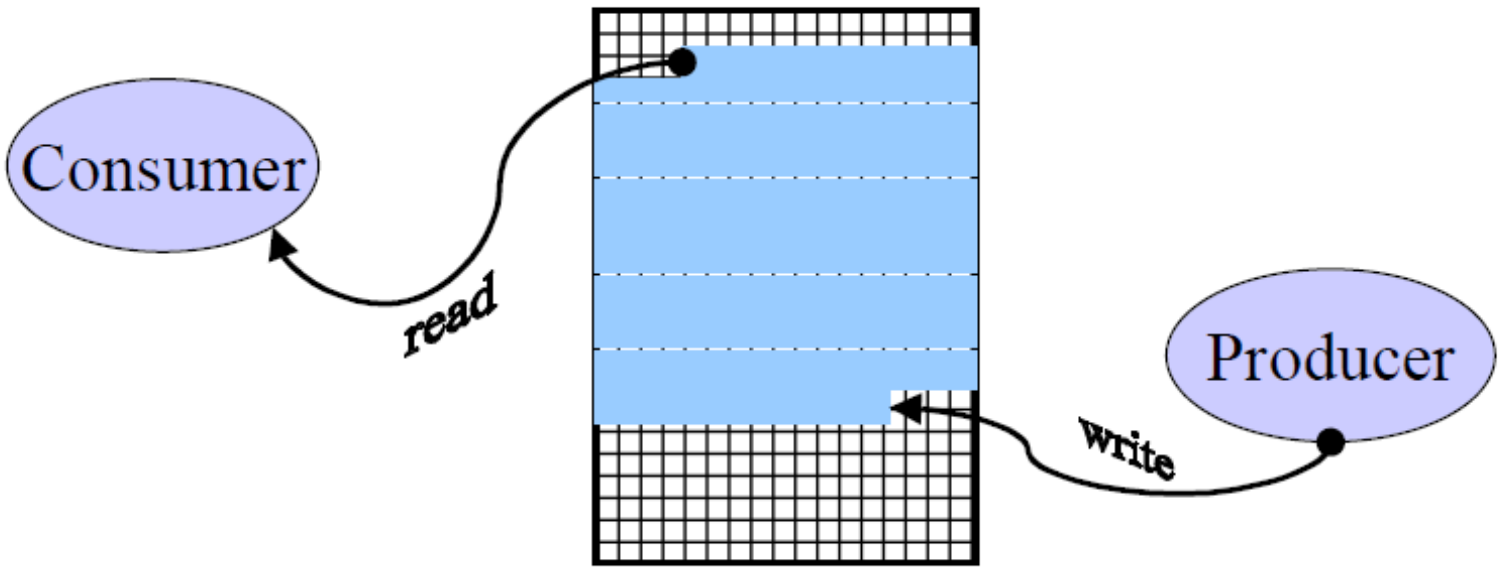
```
int main(void)
{
        initialize_all();

        Initialize_kernel(3, SCHEDULING_QUANTUM);

        /* Register Tasks */
        /* void RegisterTask(double task_period, void* task_function) */
        /* Arguments:
                task_period: Task Period (in secs). 0 for non-periodic tasks
                task_function: Pointer to the task's function
        */
        finished = 0;
        RegisterTask(0, (void*) &task1); //task1 increments finished when done
        RegisterTask(0, (void*) &task2); //task2 increments finished when done
        RegisterTask(0, (void*) &task3); //task3 increments finished when done

        /* Function to starts the tasks. This gives control to the scheduler. */
        Run_tasks(); //In our kernel this means that all tasks are being executed and the processor will not get beyond this instruction.
                     //In more advanced kernels, the tasks are spawned at the background and the next instruction in the main loop is executed.
        while (finished != 3) ;
        printf("Done");
}
```

Shared Memory: Mailboxes

- Post() - write operation- puts data in mailbox
- Pend() - read operation- gets data from mailbox
- Just like using a buffer or shared memory, except:
  - If no data is available, pend() task is suspended
  - Mutual exclusion built in: if somebody is posting, pend() has to wait.
- No processor time is wasted on polling the mailbox, to see if anything is there yet.
- Pend might have a timeout, just in case

Shared Memory: Buffering Data

- If you have a producer and a consumer that work at different rates, a buffer can keep things running smoothly
  - As long as buffer isn't full, producer can write
  - As long as buffer isn't empty, consumer can read



Shared Memory: Corruption

- Shared memory can be as simple as a global variable in a C program, or an OS-supplied block of common memory.
- In a single-task program, you know only one function will try to access the variable at a time.
- With two tasks updating the same piece of memory, conflicts arise:
  - E.g., consider the instruction x = x – a;
    - Get stored variable x
    - Subtract a (assume it is already stored in one of the 32 registers)
    - Replace x with the result x-a
- One task may interrupt another at any arbitrary point:
  - One C instruction is represented by several assembly instructions; an interrupt may happen in the middle
  - A hardware event causing an interrupt will finish 4 assembly instructions (enough to finish a C instruction such as x = x-a; but not sufficient to finish a more complex composed line of C code)
  - A RTOS orders time slices at start and end points at arbitrary spots if not explicitly instructed

Shared Memory: Corruption

- We need to be careful in multi-tasked systems, especially when modifying shared data.
- We want to make sure that in certain *critical sections* of the code, no two processes have access to data at the same time.
- If we set a flag (memory is busy, please hold), we can run into the same problem as the previous example:
  - Suppose flag = 0
  - Task1 executes while (flag !=0);
  - RTOS switches context to Task 2
  - Task2 executes while (flag !=0);
  - Task2 executes its next instruction which sets flag = 1;
  - RTOS switches context to Task1
  - Task1 executes its next instruction (after the while loop) which sets flag = 1;
  - Both tasks think they have exclusive ownership over the memory corresponding to flag … and start accessing the memory at the "same time" (the RTOS context switches Task1 and Task2 in and out).

Mutual Exclusion: Atomic Operations

- An operating system that supports multiple tasks will also support atomic *semaphores*.
- The names of the functions that implement semaphores vary from system to system
  - **test-set**/**release**
  - **lock**/**unlock**
  - **wait**/**signal**
  - **P**()/**V**()
- The idea: You check a "lock" before entering a critical section.
  - If it is set, you wait.
  - If it isn't, you go through the lock and unset it on your way out.
- The word *atomic* means checking the lock and setting it only takes one logical operation which cannot be interrupted
  - See previous lecture on how a pin interrupt can be used to write an atomic procedure

Semaphores

```
…

void SensedDataUpdate()
{
        lock( sensed_data );
        update( curr_sensed_data );
        unlock( sensed_data );
}


…
```

```
void SensedDataTransmit()
{
        lock( transmitter );



        …



        lock( sensed_data );
        transmit( curr_sensed_data );
        unlock( sensed_data );
        unlock( transmitter );
}
```

Semaphores: Deadlock

```
void ProcessSensedData()
{
        lock( sensed_data );
        edit( curr_sensed_data );

        lock( transmitter );
        … deadlock …
        transmit( edited_sensed_data );
        unlock( transmitter );
        unlock( sensed_data );
}
```

Waiting on sensed_data →

← Waiting on transmitter

```
…

void SensedDataTransmit()
{
        lock( transmitter );
        lock( sensed_data );
        …
        … deadlock …
        transmit( curr_sensed_data );
        unlock( sensed_data );
        unlock( transmitter );
}
```

Deadlock: Detection and Avoidance

- Cannot always be found in testing
- Four conditions necessary
  - Area of mutual exclusion
  - Circular wait
  - Hold and wait
  - No preemption
- Some well-known solutions exist
  - Make all resources sharable
  - Impose ordering on resources, and enforce it (if a task holds a lock on a resource x and y<x, the task cannot ask for a lock on resource y)
  - Force a task to get all of its resources at the same time or wait on all of them (a global lock approach)
  - Allow priority preemption
- Not recommended:
  - Avoidance: Only write single-task programs or programs that don't use shared memory
  - Ostrich method: Ignore the problem completely, assuming it won't happen often, or at least not often enough for your customers to sue you
  - Brute force: Disable interrupts completely during "critical section" operations

Lessons

- Buffering data can smooth out the interaction of a producer that generates data at one rate and a consumer that eats at another.
- Inter-task communication can be tricky- if your operating system supports high-level communication protocols, and they are appropriate for your task, use them!
- If you use a flag to indicate a resource is being used, understand why checking and setting the flag needs to be atomic.

Another Example

```
int ADC_channel[3];

// The following is called in an actual ISR
void ISR_ReadData(void)
{
        Read ADC_channel[0];
        Read ADC_channel[1];
        Read ADC_channel[2];
}

int delta, offset;

int main(void)
{
        …
        while(1)
        {
                …
                delta = ADC_channel[0]-ADC_channel[1];
                offset = delta*ADC_channel[2];
                …
        }
}
```

What if an interrupt happens between the calculation of delta and offset?

Even the calculation of delta can be interrupted!

Assembly

Label: Memory address where the code is located (optional).

Op-Code: Mnemonic for the instruction to be executed.

Operands: Registers, addresses or data operated on by the instruction.

The following are examples of assembly instructions for the Freescale MPC 5553 microprocessor:

```
add     r7, r8, r9;    Add the contents of registers 8 and 9, place the result
                       in register 7.
and     r2, r5, r3;    Bitwise AND the contents of registers 5 and 3,
                       place the result in register 2.
lwz     r6, 0x4(r5);   Load the word located at the memory address formed by
                       the sum of 0x4 and the contents of register 5 into register 6.
lwzx    r9, r5, r8;    Load the word located at the memory location formed by
                       the sum of the contents of registers 5 and 8 into register 9.
stwx    r13, r8, r9;   Store the value in register 13 in the memory location formed
                       by the sum of the contents of registers 8 and 9.
```

Assembly

- Assembly code instructions are atomic
- temp = temp – offset; is translated as

```
lwz  r5, 0(r10);    Read temp stored at 0(r10) and put it in r5
li   r6, offset;    Put offset value into r6
sub  r4, r5, r6;    Subtract the offset and put the result into r4
stwz r4, 0(r10);    Store the result back in memory
```

A First Solution

- Just disable/enable all interrupts …
- But interrupts are there for a reason and have the highest priority …
- Use disable/enable sparingly … good programming avoids them at all cost

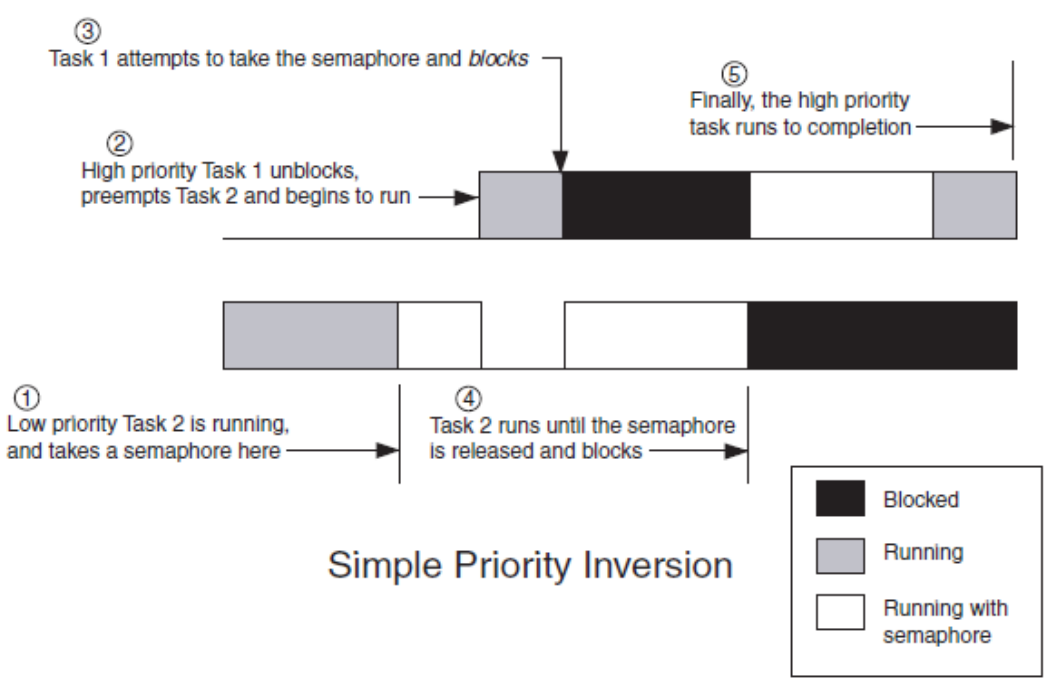
```
while(1)
{
        …
        disable();
        delta = ADC_channel[0]-ADC_channel[1];
        offset = delta*ADC_channel[2];
        enable();
        …
}
```

Another Solution?

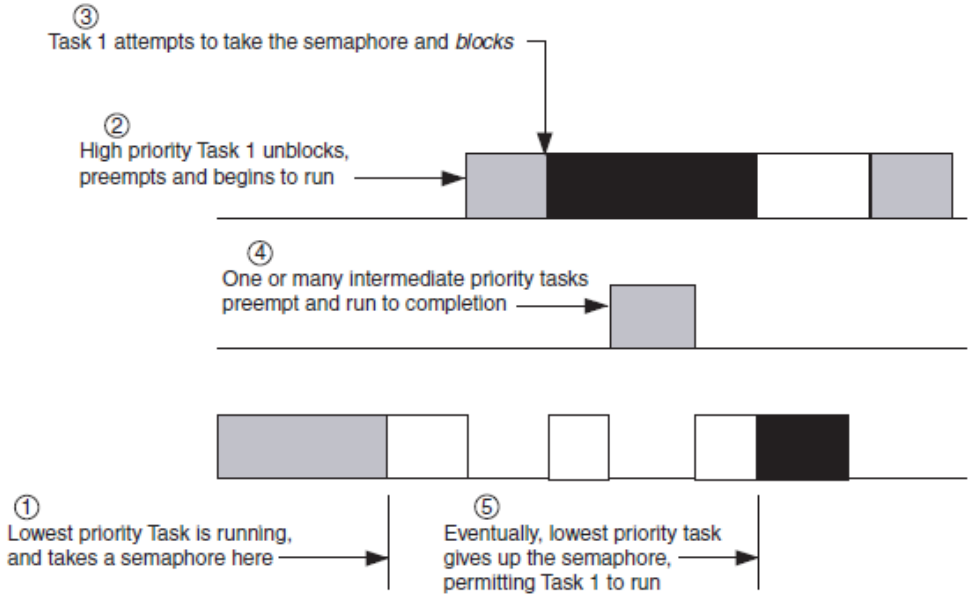
```
int ADC_channel[3];

// The following is called in an actual ISR
void ISR_ReadData(void)
{
        GlobalLock();
        Read ADC_channel[0];
        Read ADC_channel[1];
        Read ADC_channel[2];
        GlobalUnlock();
}

int delta, offset;

int main(void)
{
        while(1)
        {
                GlobalLock();
                delta = ADC_channel[0]-ADC_channel[1];
                offset = delta*ADC_channel[2];
                GlobalUnlock();
        }
}
```

Leads to deadlock! HW events that cause the ISR to execute can happen anywhere!

Semaphores: Priority Inversion

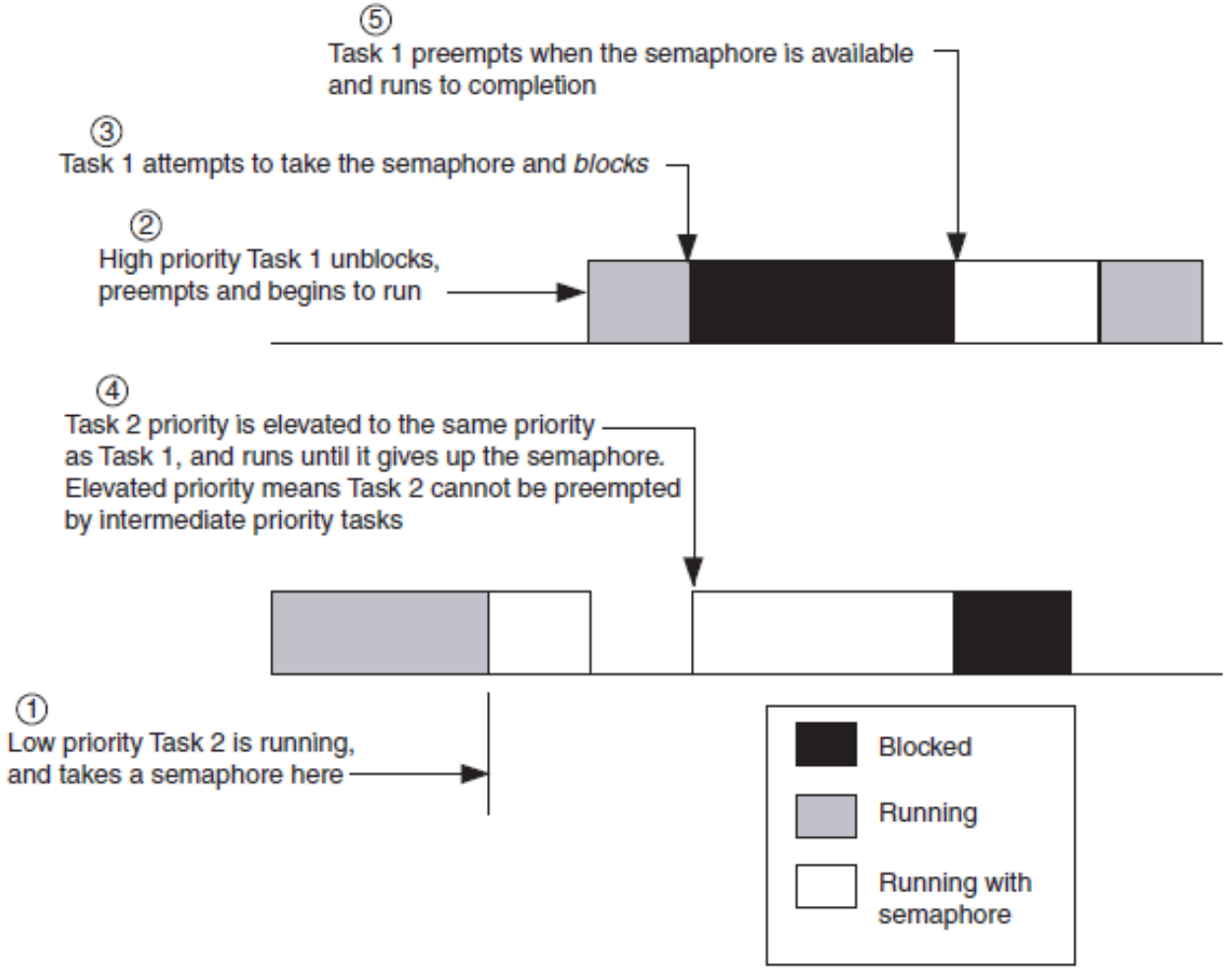


Simple Priority Inversion



Unbounded Priority Inversion

Priority Inheritance



Priority Inheritance

Priority Ceiling Protocol

- Each task is assigned a static priority, and each semaphore, or “resource” is assigned a “ceiling” priority greater than or equal to the maximum priority of all the tasks that use it.
- At run time, a task assumes a priority equal to the static priority or the ceiling value of its resource, whichever is larger:
  - if a task requires a resource, the priority of the task will be raised to the ceiling priority of the resource;
  - when the task releases the resource, the priority is reset.
- It can be shown that this scheme minimizes the time that the highest priority task will be blocked, and eliminates the potential of deadlock.

Priority Ceiling Protocol