FEMOS (Fast Embedded Operating System) Reference

Real-Time Operating System

University of Waterloo, ON, Canada

INSOP SONG (insop@susleo.com)

Contents

INSOP SONG (insop@susleo.com)

1. Introduction

1.1. Main features of FEMOS

FEMOS (Fast Embedded OS) is a real-time operating system. This is mainly for embedded application systems or real-time application systems, such as robotic systems, nuclear power plants and mobile systems.

FEMOS utilizes multitasking, priority-driven preemptive scheduling with dynamic priority scheduling, inter-process communication using priority message queue and fast context switching features, which are all ingredients of a real-time operating system. [QNX]

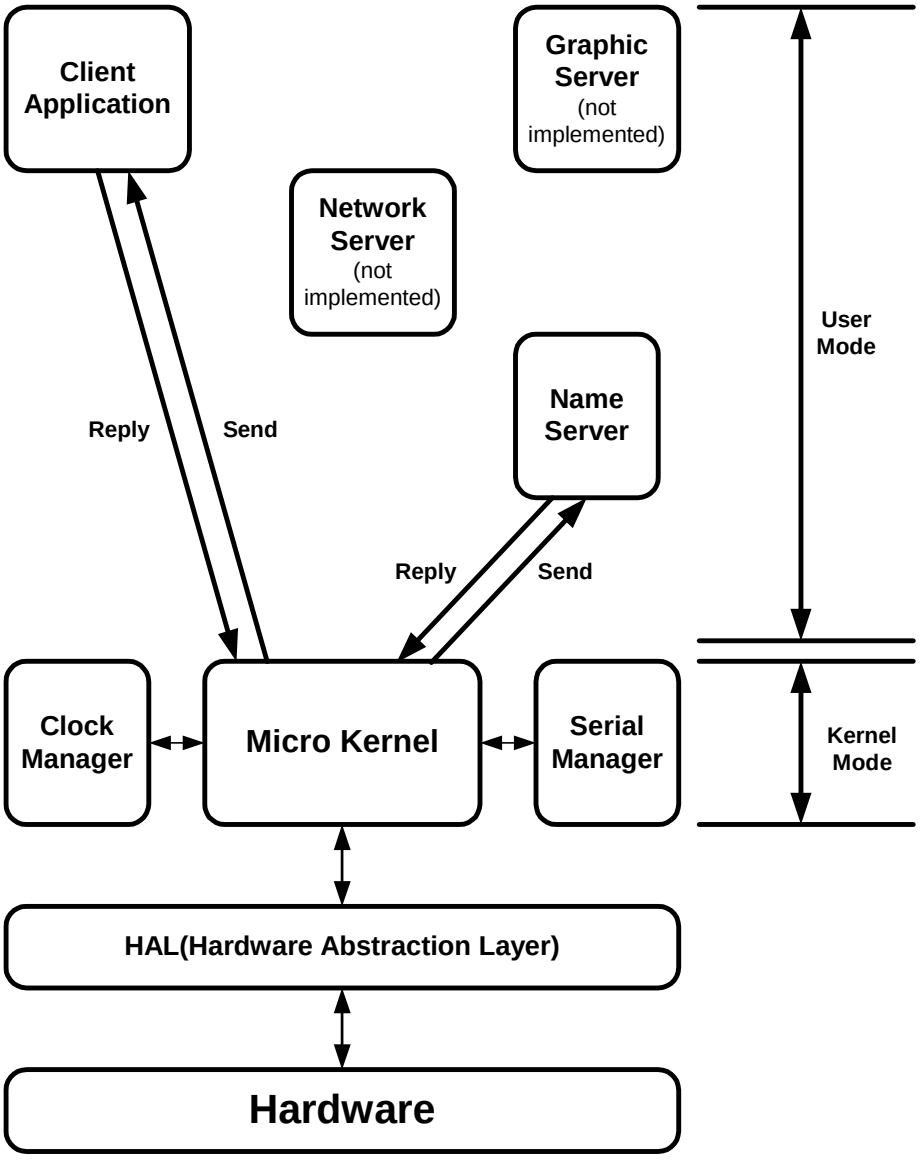
Two main *design features* of FEMOS are:

- Semi-micro kernel architecture
- Message-based inter-process communication

In addition to design features, FEMOS has *many implementation features*,

- Fully programmed in **C++** (all data structures are written using **object-oriented design** concepts)
- 64 priority-levels with preemptive multitasking,
- **Constant time, *O(1)* "Ready Queue search"** by priority to bit mapping method (Jean J. Labrosse)
- **Fast "*switch*" implementation** using function pointers (Michael Barr)
- **Multi-level prioritized message queue** (same as system's total priority)
- **Priority inheritance** in inter-process communication for avoiding priority inversion
- Shared memory implementation
- CPU usage task using shared memory
- Clock manager and Serial manager are included at Kernel level, and all other system services are run at user level (Semi-Micro kernel)

- Use of C++ features, such as *Inline functions* which speeds operations



FEMOS Arichitecture: semi-micro kernel, IPC based client-server

Figure 1. Femos Architecture

The Femos kernel is a semi-micro kernel, based on a QNX style micro kernel. Typical Micro kernels only have very minimal functionality, including support for message passing and scheduling functions.

Unlike QNX, Femos included a clock manager, the system clock (tick) manager, and a serial port manager (RS-232C port driver) see Figure 1. These two managers are placed in kernel itself, to improve system speed and performance, but it is easy to change these two managers to work outside of the kernel, like other user applications, so that the two servers use message passing.

1.2. Inter-Process Communication

When several processes run concurrently, as in typical real-time multitasking environments, the operating system must provide mechanisms to allow processes to communicate and synchronize with each other.

Send / Receive / Reply based IPC (Inter Process Communication) is the key to designing an application as a set of cooperating processes in which each process handles one well-defined part of the whole job.

Femos provides a simple but powerful set of IPC primitives (based on QNX IPC) capabilities that greatly simplify the job of developing applications made up of cooperating processes.

In Femos, a message is a packet of bytes passed from one process to another. Femos attaches no special meaning to the content of a message - the data in a message has meaning for the sender of the message and for its receiver, but for no one else.

Message passing not only allows processes to pass data to each other, but also provides a means of *synchronizing the execution of several processes*. As a process sends, receives, and replies to a message, the process undergoes various "changes of state" that affect when, and for how long, it may run. Knowing a processes state and priority, the Micro kernel can schedule a processes as efficiently as possible to fully utilize available CPU resources. Thus message passing is used heavily throughout the entire system.

Real-time and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The discipline imposed by FEMOS's message-passing design helps bring order and greater reliability to applications.

Yunho Jeon, CTO of Zestel, Co. Ltd (a Mobile Internet solution company based at Korea) said that, "the QNX synchronization primitives [Send-Receive-Reply, adapted by FEMOS] are the best tools to write *multiprocess /threaded applications*." After debugging numerous synchronization-related bugs in the VoiceXML interpreter, he implemented the QNX-style sync object in Java and is a heavy user of this object. Jeon thinks that there should be a law to force programmers to use QNX-style sync primitives for more than 90% of their synchronization needs, except for simple mutexe. Jeon has even been quoted as saying, "*Who's the one who first designed the primitive? He must be a genius*".

1.3. Development environment (Eos system)

FEMOS was developed on the Eos system for CS452/652, Real-Time Programming, a course offered by the department of Computer Science, at the University of Waterloo). FEMOS and a related application system is composed of different tasks (each of task has its own main() function and own *a.out header*).

Using the “*Bind486*” program and “*Bindfile*” file to bind all these programs (tasks and kernel). The Eos system uses Intel’s Protected mode, which entails the use of the GDT (General Descriptor Table) and IDT (Interrupt Descriptor Table). Several bind programs are loaded using the “*etherboot*” program which uses the tftp protocol for loading programs from server on to the Eos system (an IBM PC).

Femos find every program’s (task’s) memory position using “*Bindfile*” information, when Femos find the position of the each program, it read “*a.out header*” and set up the memory space for data and stack.

Every task in the FEMOS system has its own memory space. It is not possible to exchange data asynchronously, unlike IPC, which is synchronous. Thus FEMOS has a shared memory area, within each process, for asynchronous communications.

This memory sharing is implemented by letting one task access another task’s memory space with interrupts off. Therefore, shared memory access should be limited and processed as quickly as possible.

Real-time and other mission-critical applications generally require a dependable form of IPC, because the processes that make up such applications are so strongly interrelated. The discipline imposed by FEMOS's message-passing design helps bring order and greater reliability to applications.

1.4. Overview (File structure)

Followings are directory structure of FEMOS.

- Task: task descriptor, priority queue and doubly liked list
- Kernel: kernel main function and kernel class
- HAL: Hardware Abstraction Layer, GDT, *Bindfile, a.out header* IDT related programs

- Serial: serial port setup, bounded buffer for serial port
- BSP: Board Specific Part, Intel chip specific programs
- Syscall; system function call definition and interface from tasks and kernel using software interrupt.

1.5. Stereotypical Task Structures

In this section, the author introduces some task structures such as Notifier, server, buffer and more. Because, FEMOS is semi-micro kernel, most of the things will be handled at outside of kernel and as individual task. These will be the building block of application tasks and program.

Task is independent autonomous agent and basic application structuring unit. Task is composed of object and concurrency.

Multi-task (process) structuring is using stereotyped task team structuring.

Several producers, several consumers, with buffering, The buffer becomes a task

```
procedure producer()
    forever {
  produce( data );
  Send( bufferId, request, sizeof(request), ack, sizeof(ack) );
    }

procedure consumer()
    forever {
  Send( bufferId, request, sizeof(request), data, sizeof(data) );
  consume( data );
    }

procedure buffer()
    forever {
        id = Receive( request, sizeof(request) );
        if request.type == CONSUME {
```

```
                    if producerQueueEmpty {
                        enqueueConsumer( id );
                    } else {
                        dequeueProducer( data );
                        Reply( id, data, sizeof(data) );
                    }
                }
                elsif request.type == PRODUCER {
                    if ~consumerQueueEmpty {
                        consumerId = dequeueConsumer( );
                        Reply( consumerId, request.data, sizeof(request.data) );
                    } else {
                        enqueueProducer( request.data );
                    }
                    Reply( id, ack, sizeof(ack) );
                }
```

Notifiers and Servers

procedure clockNotifier

```
    forever {
        AwaitEvent( clockTick );
        time++;
        Send( clockServer, time, sizeof(time), ack, sizeof(ack) );
    }
```

procedure clockServer

```
    forever {
        requester = Receive( request, sizeof( request ) );
        switch (request.type) {
        case NOTIFIER:
            Reply( requester, ack, sizeof(ack) );
            time = request.data;
            while( nextWaiterTime( ) <= time ) {
                dequeueNextWaiter( );
```

```
                                        Reply( nextWaiter, ack, sizeof(ack) )
                        }
                        break;
                case DELAY:
                        if( request.data > 0 ) {
                                enqueueWaiter( requester, request.data );
                        } else {
                                Reply( requester, ack, sizeof(ack) );
                        }
                        break;
                case WAIT_UNTIL:
                .                       .
                }
        }
```

Stereotypical Task Structures

Tasks are divided into three groups: ***Servers***, ***Workers*** and ***Clients***.

In *server* group, there are *proprietors*, who provide service using resources that must be synchronized and *distributors*, who acquire things from producers, store them, and provide them to consumers, and *administrators*, who assign and monitor work done by others.

In *worker group*, there are *notifiers*, *couriers*, who take messages from server to server, *agents*, who accept work on behalf of other tasks.

Lastly, there is *Clients* who realize goals by sending requests to servers and getting back results.

Proprietor: A proprietor manages a resource that requires synchronized access. The basic code structure is

```
procedure Proprietor
        initialize resource;
        forever {
                id = Receive( request );
```

```
        service( id, request, reply );
        Reply( id, reply );
        }
```

- Note that the resource is initialized before any requests are accepted
- The Reply guarantees that -- at this time -- the service is complete
- Only a single request is processed at a time.
- The proprietor does the work on behalf of the client, at the priority of the proprietor.
- Request servicing can be split into a time-critical and a non-time-critical part, with the Reply between the two parts.

Distributors receive two kinds of requests, delivery requests and order requests. Thus, they may need to re-order service.

procedure Distributor

```
    initialize warehouse;
    forever {
        id = Receive( request );
        switch (request.type) {
        case DELIVERY:
            if ( backOrdered ) {
                dequeue( order );
                Reply ( orderer, request.data );
                Reply( id, ack );
            } elsif ( warehouseNotFull ) {
                store( request.data );
                Reply( id, ack );
            } else {
                Reply( id, error );
            }
            break;
        case ORDER:
            if ( warehouseEmpty ) {
```

```
                    queue( order );
                } else {
                    data = withdraw( );
                    Reply( id, data );
                }
                break;
            }
        }
```

Administrator: A weakness of the proprietor is that high priority, quick requests can be held in line behind low priority, slow requests. The administrator can take care of this problem.

procedure Administrator

```
    initialize resource;
    create pool of workers;
    forever {
        id = Receive( request );
        switch (request.type) {
        case SERVICE:
            if ( workerAvailable ) {
                worker = getFromPool( );
                Reply( worker, workRequest );
            } else {
                queue( workRequest );
            }
            break;
        case WORKER:
            if ( request.data # NULL ) {
                Reply( request.data.id, request.data.result );
            }
            if ( requestQueued ) {
                workRequest = dequeue( );
                Reply( id, workReuest );
```

```
                        } else {
                                addToPool( worker );
                        }
                        break;
                }
        }
```

procedure Worker

```
        initializeWorkspace;
        Send( administrator, NULL, workRequest );
        forever {
                workResult = doWork( workRequest );
                Send( administrator, workResult, workRequest );
        }
```

Guard: A guard can be work as guard, who manage the passage so that only one message can pass through to server.

```
Procedure Guard{
        RegisterAs(Name)
        While(1) {
        (id, request)= Receive();
        Send(MyParentTid(), request, response)
        Reply(id, response)
        }
}
```

2. Process Scheduling

2.1. Task Abstraction

Real-time OS manages user application process/tasks by using task descriptor block, following are the elements should be stored for task managements

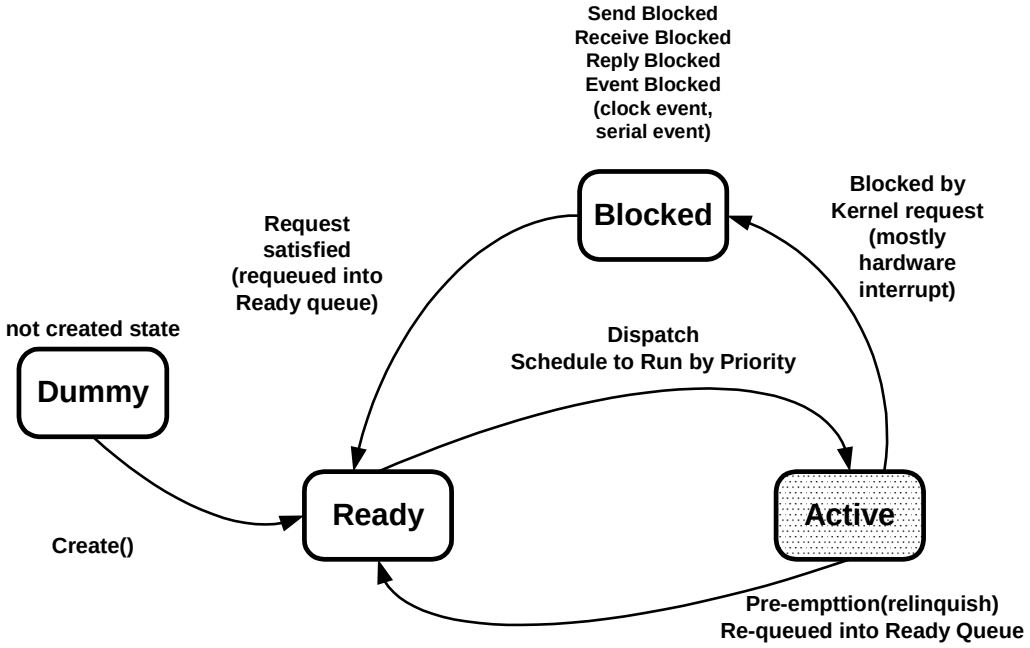
- Each task has a unique task identifier (TID, Task Identifier)
- Each task has a unique priority
- Each task has it own task state
- Each task has it own data space (include stack)

2.2. Task States

Every tasks in Femos has it's own task state and Femos schedules by these states.

These States are managed by tasks' priority and other hardware interrupts (clock event blocked, serial event blocked) and IPC states (Send blocked, Receive Blocked, Reply blocked).

Each state is follows, *Ready state* is inactive state managed by *Ready Queue (priority queue)* and *Active state* is task's running state, using process power state. *Dummy state* is the state, which is not yet created by any other tasks. In addition, Femos has *Blocked state* for IPC and hardware interrupts.



Femos Task State

Figure 2. Femos Task State

2.3. Task Descriptor Implementation

Task descriptor has all the information for context switching for Femos kernel. It has its TID (Task Identifier), priority, task state, and data space (include stack). Task descriptor is implemented at *'\femos\task\task.h\ and \femos\task\task.cc' class*. Following is the excerpt of Task Class.

```
// --
//              Task Descriptor Block class
// --


//              Task State define
enum TaskState          { Dummy, Active, Ready, Dead, SendBlocked, ReplyBlocked,
                  ReceiveBlocked, EventBlocked, PriorityInherited };
class Task
```

```
{
   private:
//  assign unique Tid for each TCB Objects
        static int      nextTid;
        int             tid;
        TaskState       state;
        // task priority
        int             priority;
// for priority inheritance
        int             priorityOriginal;
//maybe this is not necessary.. if not, delete this later.
        int             priorityInherited;
        Boolean         isPriorityInheritanceFlag;
// timer for clock event
        int             timer;


   public:
        int             parentTid;
// pointer for queue pNext for next, pPrev for previous Task
        Task *          pNext;
        Task *          pPrev;
// context     aContext
        int             pData;          // pointer to data address
        int             userCS;
        int             userDS;
        int             userSP;
        int             userSS;
        int             requestedSysCall;
// maybe set up one class for bindfile class
        char *          taskName;   /* The name of the process */
        int             codeSize;    /* Self explanitory variable! */
        int             entryPoint;  /* Start executing here */
        int             dataTotalSize;      /* Real memory pointer for data */
        int             stackSize;    /* Size of the stack - no data */
```

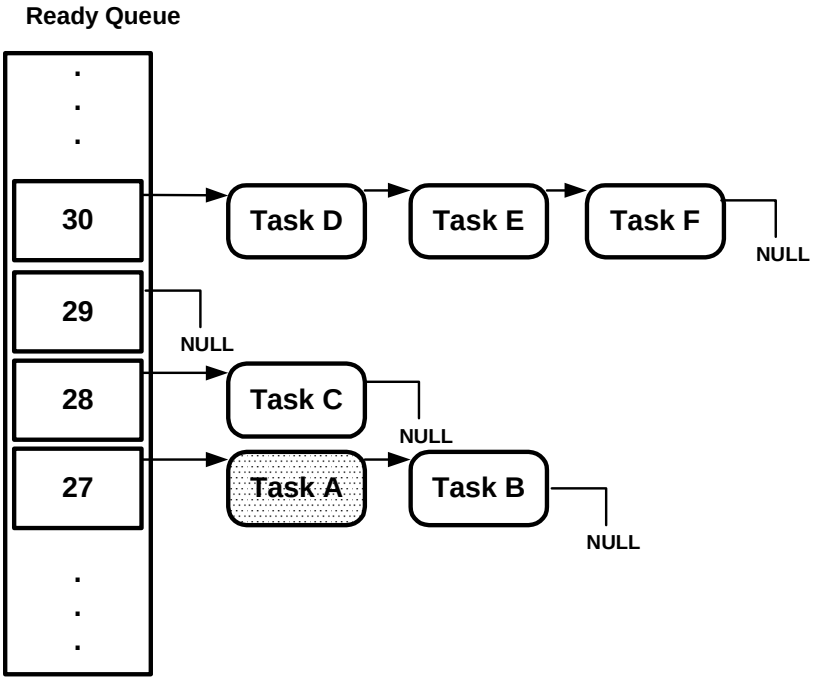
```
// for message copy and handling, IPC stuffs
        Task *          pTaskSend;
        Task *          pTaskRecv;
        int             pMsgSend;
        int             msgSendLen;
        int             pMsgRecv;
        int             msgRecvLen;
// member functions
        Task();
        ~Task();
        void initTask();
        ErrorCode isValidPriority(int _priority);
        void printTask();
        int     getTid();
        void    setParentTid(int _parentTid);
        int     getParentTid();
        // get task state in string form
        void    setState(TaskState _state);
        char *  getState();
        int     getStateNum();
        ErrorCode setPriority(int _priority);
        int       getPriority();
// timer related member functions
        inline int      getTimer( void );
        inline void     setTimer( int timer );
        inline Boolean          isTimeOut( int upTime );
        inline Boolean          isPriorityInherited( void );
        inline void   setPriorityInheritance( void );
        inline int    getOriginalPriority( void );
        inline void   resetPriorityInheritance( void );
        inline void   changeInheritePriority( int _inheritedPriority);
        inline void   revertInheritePriority( void );
};
```

2.4. Ready Queue (Priority Queue)

The Femos's scheduler makes scheduling decisions when:

- A task becomes unblocked
- The time slice for a running task expires
- A running task is preempted
- A running task call system functions call (syscall)

In Femos, every task is assigned a priority (1-64). The scheduler selects the next task to run by looking at the priority assigned to every process that is READY (a READY process is one capable of using the CPU). The task with the highest priority is selected to run.

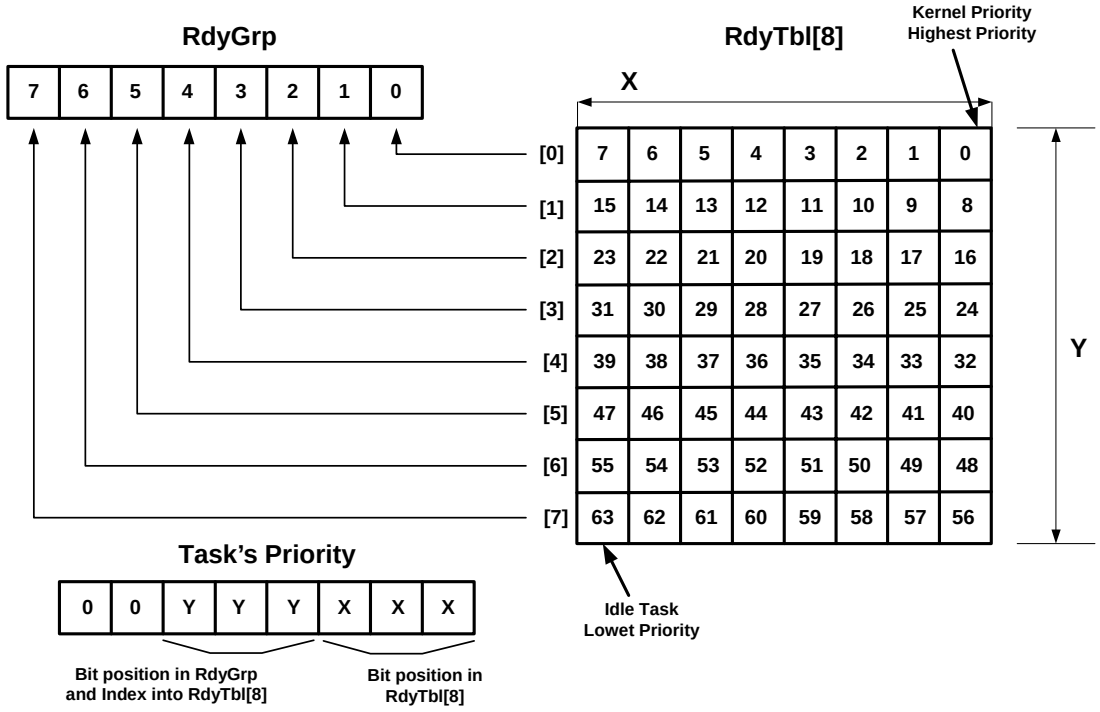


Femos Ready Queue

Figure 3 Femos Ready Queue

In Figure 3, scheduler de-queue Task A from the priority 27 queue and Task A stat is change from Ready to Active. When Task A is schedule to Ready state again it en-queue at the end of the priority 27 queue.

Scheduler search Ready queue and can find out highest priority ready task in *Constant time, O(1),* using Bitmap search method (Jean J. Labrosse). Followings are source code excerpt of Ready queue, which are located at "*\femos\task\priQueue.h"* and*" \femos\task\priQueue.cc*".



Constant time Ready queue search

Figure 4 Constant time Ready queue search by priority bit grouping

```
//
//          Priority Queue for Task Descriptor Block
//
class       PriQueue {
  private:
```

```
        UBYTE       rdyGrp;
        UBYTE       rdyTbl[8];
        int         numItem;
    public:
        TaskQueue  taskQ[PriorityRange];    // priority queue
        PriQueue();
        inline ErrorCode addPriQueue(Task * _pTask);
        inline ErrorCode addQueue(Task * _pTask, int _priority);
        inline Task *           getNextTask();
        inline Boolean   isEmpty();
        void printQueue(int _priority);         // print specific priority
        void printPriQueue();     // print priority queue
        inline void    setPriorityMask(int _priority);
        inline void    clearPriorityMask(int _priority);
        inline int      getHighestPriority(void);
        inline Task *           removeQueue(int _priority);
        inline Boolean          removePriQueue( Task * _pTask );
};

static UBYTE const MapTbl[]   = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

static UBYTE const UnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

```
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};


inline void   PriQueue::setPriorityMask(int p) {
    rdyGrp        |= MapTbl[ p >> 3 ];
    rdyTbl[p >>3 ]  |= MapTbl[ p & 0x07 ];
}
inline void   PriQueue::clearPriorityMask(int p) {
    if((rdyTbl[ p >> 3 ] &= ~MapTbl[ p & 0x07 ]) == 0 )
                rdyGrp         &= ~MapTbl[ p >> 3 ];
}
//              get highest priority
//              by bit mapping
inline int     PriQueue::getHighestPriority( void ) {
    register x,y,priority;
    y = UnMapTbl[ rdyGrp ];
    x = UnMapTbl[ rdyTbl[ y ] ];
    priority = ( y << 3) + x;
    return priority;
}
```

2.5. Scheduling

To meet the needs of various applications, Femos provides two scheduling methods:

- FIFO scheduling as static scheduling
- Priority inheritance, adaptive scheduling, dynamic scheduling

Each task on the system may run using any one of these methods. They

are effective on a per-task basis, not on a global basis for all processes on a node.

Remember that these scheduling methods apply only when two or more tasks that share the same priority are READY (i.e. the tasks are directly competing with each other). If a higher-priority task becomes READY, it immediately preempts all lower-priority processes.

In *FIFO scheduling*, a task selected to run continues executing until it:

- Voluntarily relinquishes control (e.g. it makes *any* kernel call)
- Is preempted by a higher-priority task

In *priority inheritance, dynamic scheduling*, a task selected to run continues executing until it:

- A task get message from higher task

Priority inheritance is implemented for avoid the unwanted priority inversion, which was happened at Mars Path Finder (*http://www.time-rover.com/Priority.html*). This will be covered more detail at Chapter 3. IPC.

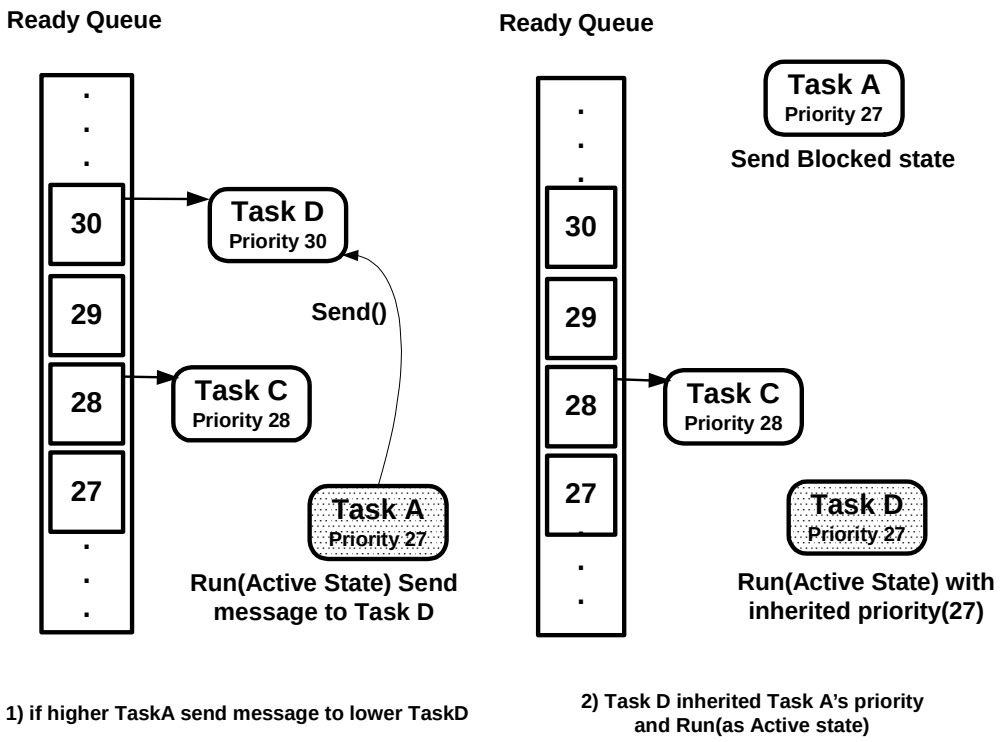


Figure 5 Femos Priority Inheritance

2.6. Priority inversion and Mars Path Finder

The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. Successes included its unconventional "landing" -- bouncing onto the Martian surface surrounded by airbags, deploying the Sojourner rover, and gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web. But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer

was trying to do too many things at once".

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data-gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue.

The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the(medium priority) communications task to be scheduled during the short interval while the (high priority) information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

2.7. Context Switching

Femos uses fast *switch()* statement for context switching and system function call inside of the Kernel using function pointer array.

Define function pointer array at the beginning of the function like follows

```
//--
//  Estabilishement of a table of pointers to functions
//              for  fast switch implementations
//--
int (*switchFunctions[])(void)  = \
    {       dispatcher_sysCall,
            handler_clockEvent,
            handler_serialEvent_0,
            handler_serialEvent_1
            };
```

In the main function, *switch() statement* will be implemented like this with faster operation than real switch().

```
status= switchFunctions[getNextRequest()]();
// followings are the same as the above
//request = getNextRequest();
//status= switchFunctions[request]();
```

3. IPC (Inter-Process Communication)

The Femos semi-micro kernel supports message type of IPC:

- *Messages* - the fundamental form of IPC in Femos. They provide synchronous communication between cooperating processes where the process sending the message requires proof of receipt and potentially a reply to the message.

3.1. IPC basic

IPC fulfills two roles

- Transfer of data from one task to another
- Synchronization, which produces control-flow between tasks by manipulating task state between ready and blocked

In Femos, a message is a packet of bytes that's synchronously transmitted from one task to another. Femos attaches no meaning to the content of a message. The data in a message has meaning for the sender and for the recipient, but for no one else.

To communicate directly with one another, cooperating processes use these C language functions:

- status = ***Send(tid, *request, reqLen, *replyBuf, replyBufLen)***
  - Send message to specific task(tid) and block awaiting a reply
  - status >= 0 : length of reply
  - status < 0 : error
- status = ***Receive(* request, reqLen)***
  - Block awaiting a message from any process

- Send message to specific task(tid) and block awaiting a reply
- status >= 0 : tid of sender
- status < 0 : error

- ***Reply (tid, *reply, replyLen)***

- Reply to a received message, never blocks

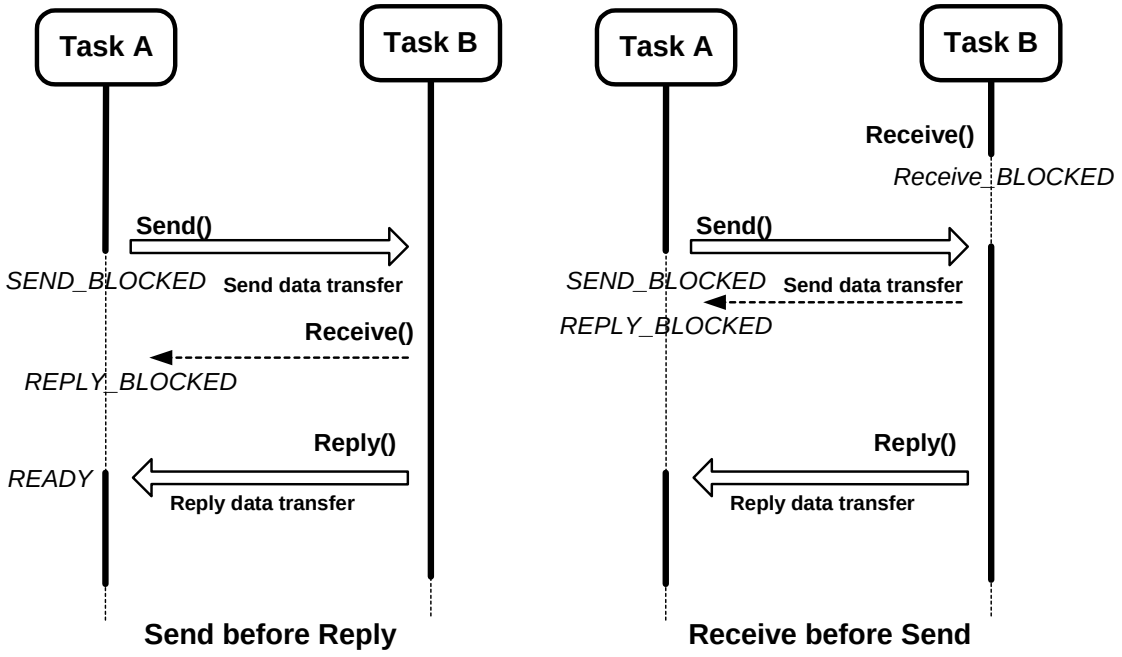


Figure 6 Femos IPC

The above illustration outlines a simple sequence of events in which two Tasks, Task A and Task B, use *Send()*, *Receive()*, and *Reply()* to communicate with each other:

1. Task A sends a message to Task B by issuing a *Send()* request to the kernel. At this point, Task A becomes SEND-blocked until Task B issues a *Receive()* to receive the message.
2. Task B issues a *Receive()* and receives Task A's waiting message. Task A changes to a REPLY-blocked state. Since a message was

waiting, Task B doesn't block. (Note that if Task B had issued the *Receive()* before a message was sent, it would become RECEIVE-blocked until a message arrived. In this case, the sender would immediately go into the REPLY-blocked state when it sent its message.)

3. Task B completes the processing associated with the message it received from Task A and issues a *Reply()*. The reply message is copied to Task A, which is made ready to run. A *Reply()* doesn't block, so Task B is also ready to run. Who runs depends on the relative priorities of Task A and Task B.

Typical Application Code

High priority server who receives requests and provides service in response to them

Server:

```
forever {
        rqst.id := Receive( rqst.args, rqst.len );
        ParseArgsAndProvideService( rqst, service );
        Reply( rqst.id, service.result, service.len );
        }
```

Lower priority clients who send requests and receive service

Client:

```
forever {
        DoStuffThatProducesNeedForService( rqst );
        error = Send( rqst.serverId, rqst, rqst.len, service, service.len );
        MakeUseOfService( service );
    }
```

- Client is, in effect, executing at the priority of the server while send-blocked.
- Server is, in effect, executing at priority of highest-priority sender

while receive-blocked.

- Client has to know the task Id of server before transaction starts.
- Server learns task Id of sender during transaction; doesn't have to know it in advance.
- Type checking is the responsibility of the application programmer.

3.2. Task Synchronizations

Message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several cooperating processes.

Let's look at the Figure 6 again. Once Task A issues a *Send()* request, it's unable to resume execution until it has received the reply to the message it sent. This ensures that the processing performed by Task B for Task A is complete before Task A can resume executing. Moreover, once Task B has issued its *Receive()* request, it can't continue processing until it receives another message

When a process isn't allowed to continue executing - because it must wait for some part of the message protocol to end - the process is said to be *blocked*

The followings summarizes the blocked states of processes:

- *SEND_BLOCKED: Send()* request, and the message it has sent hasn't yet been received by the recipient process
- *RECEIVE_BLOCKED: Receive()* request, but hasn't yet received a message
- *REPLY_BLOCKED*: *Send()* request, and the message has been

received by the recipient process, but that process hasn't yet replied

A server Task is normally RECEIVE-blocked for a request from a client in order to perform some task. This is called *send-driven messaging*: the client task initiates the action by sending a message, and the server replying to the message finishes the action.

Although not as common as send-driven messaging, another form of messaging is also possible - and often desirable - to use: *reply-driven messaging*, in which the action is initiated with a *Reply()* instead. Under this method, a "worker" process sends a message to the server indicating that it's available for work. The server doesn't reply immediately, but rather "remembers" that the worker has sent an arming message. At some future time, the server may decide to initiate some action by replying to the available worker process. The worker process will do the work, and then finish the action by sending a message containing the results to the server.

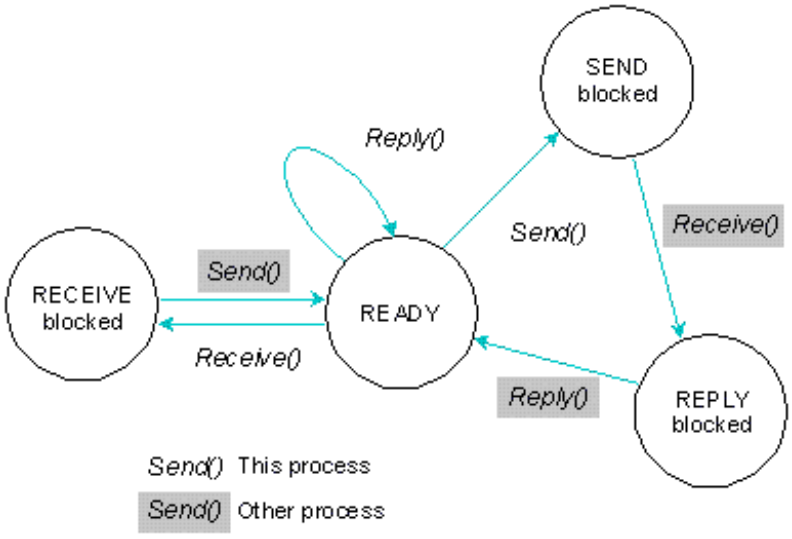
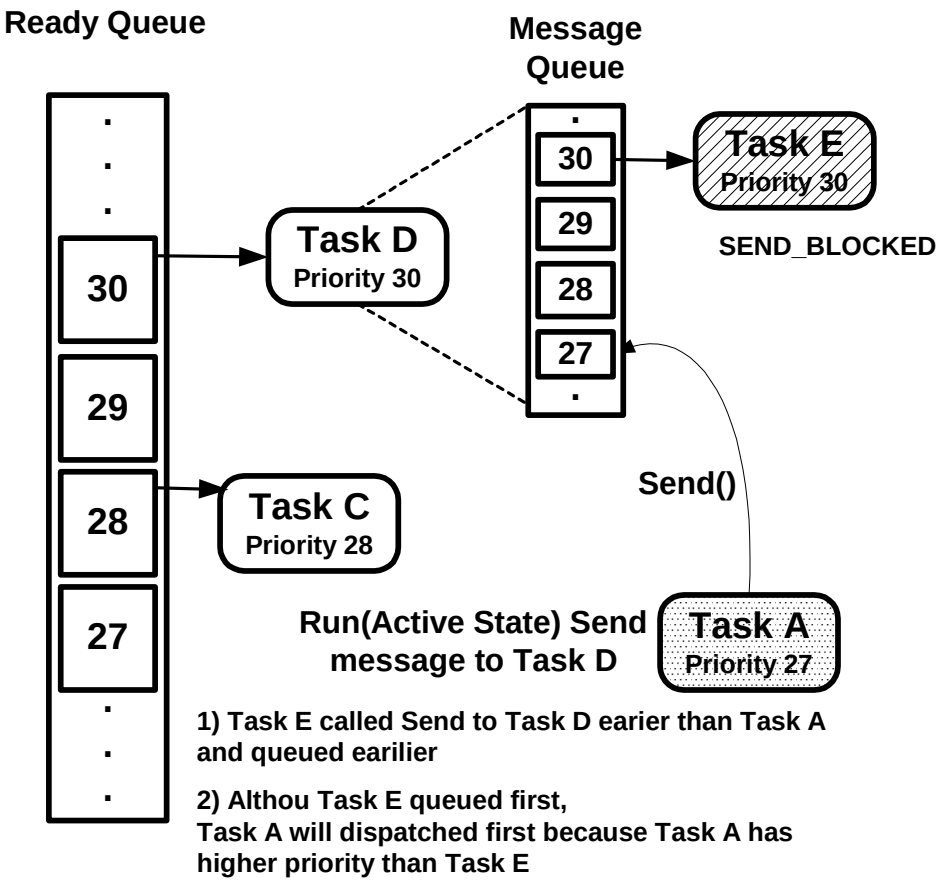


Figure 7 Femos Task State when IPC(from QNX system Arch.)

3.3. Prioritized Message Queue

One of the features of Femos is that Femos has prioritized message queue with the same number of system priority (64 or 32).

Femos is designed by Objected-oriented method; every data structure in Femos is made by C++ language. Femos is using prioritized Ready queue for scheduler. At the same concept, Femos uses the same priority queue for message queue without any change of data structure.



Femos Message Queue with Priority

Figure 8 Femos Prioritized Message Queue

3.4. Priority Inheritance

Since, Femos has prioritized message queue, it is quite simple to implement the priority inheritance.

Priority inheritances will occur when a higher priority task send a message to a lower task as in Figure 8. A high priority will be inherited to lower priority task to avoid priority inversion.

3.5. IPC via Semaphores and other things

Semaphores are another common form of synchronization that allows tasks to "post" and "wait" on a semaphore to control when tasks wake or sleep. The post operation increments the semaphore; the wait operation decrements it.

If you wait on a semaphore that’s positive, you won’t block. Waiting on a non-positive semaphore will block until some other process executes a post. It’s valid to post one or more times before a wait - this will allow one or more processes to execute the wait without blocking.

A significant difference between semaphores and other synchronization primitives is that semaphores are "async safe" and can be manipulated by signal handlers. If the desired effect is to have a signal handler wake a process, semaphores are the right choice.

Here are some more things to keep in mind about message passing:

- The message data is maintained in the sending task until the

receiver is ready to process the message. There is *no* copying of the message into the kernel. This is safe since the sending task is SEND-blocked and is unable to inadvertently modify the message data

- The message reply data is copied from the replying task to the REPLY-blocked task as an atomic operation when the *Reply()* request is issued. The *Reply()* doesn't block the replying task - the REPLY-blocked task becomes unblocked after the data is copied into its space.
- The sending task doesn't need to know anything about the state of the receiving task before sending a message. If the receiving task isn't prepared to receive a message when the sending task issues it, the sending task simply becomes SEND-blocked.
- If necessary, a task can send a zero-length message, a zero-length reply, or both.
- From the developer's point of view, issuing a *Send()* to a server task to get a service is virtually identical to calling a library subroutine to get the same service. In either case, you set up some data structures, then make the *Send()* or the library call. All of the service code between two well-defined points - *Receive()* and *Reply()* for a server task, function entry and return statement for a library call - then executes while your code waits. When the service call returns, your code "knows" where results are stored and can proceed to check for error conditions, process results, or whatever.
- Despite this apparent simplicity, the code does much more than a simple library call. The *Send()* may transparently go across the network to another machine where the service code actually executes. It can also exploit parallel processing without the overhead of creating a new task. The server task can issue a *Reply()*, allowing the caller to resume execution as soon as it is safe to do so, and meanwhile continue its own execution.
- There may be messages outstanding from many processes for a single receiving task. The receiving task can specify that messages

be received in an order based on the priority of the sending task.(using multiple message prioritized queue)

3.6. IPC implementation

Followings are source code of Send, Receive and Reply system function call implementation (which is located at *"\femos\kernel\kernelMain.cc"* and system function call interface source is *"\femos\syscall\sysCall.c"* and *"\femos\syscall\sysCall.h"*)

```
/*
 *********************************************************************
 * function : send(void)
 *********************************************************************
 */
void sysCall_send(void) {
    ErrorCode errorCode = OK;
    Task * pTask = aKernel.pCurrentTask;
    int  rcvTid;
    Task * pTask_Rcv;

// get tid and convert this into pointer
    rcvTid = getParameter(1);
    pTask_Rcv = pTask->pTaskRecv   = aTDTable.getPtTask(rcvTid);
    pTask->pMsgSend    = getParameter(2);
    pTask->msgSendLen  = getParameter(3);

// when this replied, that means received
    pTask->pMsgRecv   = getParameter(4);
    pTask->msgRecvLen = getParameter(5);

// receiver is not ready
    if(pTask->pTaskRecv->getStateNum() != ReceiveBlocked ) {
            pTask->setState( SendBlocked );
    // if receiver priority is less than equal to sender priority
```

```
    // i.e. no need for priority inheritance
                if(pTask->getPriority() >= pTask_Rcv->getPriority()) {
    // get the pointer of msg queue of each task
                    pMsgQueue = aTDTable.getPtMsgQueue( rcvTid );
                    pMsgQueue->addPriQueue( pTask );
                } // if
                // if receiver priority is greater than sender priority
                // i.e. priority inheritance
                else {
    // get the pointer of msg queue of each task
                    pMsgQueue = aTDTable.getPtMsgQueue( rcvTid );
                    pMsgQueue->addPriQueue( pTask );
                // remove receiver task from original priority queue
                    if(pTask_Rcv->getStateNum() != EventBlocked) {
                            pReadyQueue->removePriQueue( pTask_Rcv );
                            pTask_Rcv->changeInheritePriority( pTask->getPriority() );
                    // put add queue againe
                            pTask_Rcv->setState(Ready);
                            pReadyQueue->addPriQueue( pTask_Rcv );
                    }
                    // Because some task is blocked as Event
                    // if event blocked , just change priority
                    else {
                            pTask_Rcv->changeInheritePriority( pTask->getPriority() );
                    }
                } // else
    } // if
// if receiver is waiting, blocked
    else if(pTask->pTaskRecv->getStateNum() == ReceiveBlocked ) {
// if receiver priority is less than equal to sender priority
    // i.e. no need for priority inheritance
                if(pTask->getPriority() >= pTask_Rcv->getPriority()) {
                    msgCopy( pTask, pTask_Rcv);
                // set Return value for receiver sender TID
```

```
                setReturnValueTo(pTask->getTid(),
                                        pTask_Rcv);
                                        //aKernel.pCurrentTask->pTaskRecv);
            // set sender pointer for receiver
                //pTask->pTaskRecv->pTaskSend = pTask;
                pTask_Rcv->pTaskSend = pTask;
            // send task is going to reply blocked , waiting reply
                pTask->setState( ReplyBlocked );
            // set receiver ready and put into ready queue
                pTask_Rcv->setState( Ready );
                pReadyQueue->addPriQueue( pTask_Rcv );
            } // if
            // if receiver priority is greater than sender priority
            // i.e. priority inheritance
            else {
                msgCopy( pTask, pTask_Rcv);
            // set Return value for receiver sender TID
                setReturnValueTo(pTask->getTid(),
                                        pTask_Rcv);
                                        //aKernel.pCurrentTask->pTaskRecv);
            // set sender pointer for receiver
                //pTask->pTaskRecv->pTaskSend = pTask;
                pTask_Rcv->pTaskSend = pTask;
            // send task is going to reply blocked , waiting reply
                pTask->setState( ReplyBlocked );
            // change the priority with inherited one
                pTask_Rcv->changeInheritePriority( pTask->getPriority() );
            // set receiver ready and put into ready queue
                pTask_Rcv->setState( Ready );
                pReadyQueue->addPriQueue( pTask_Rcv );
            } // else
    } // else if
    //return errorCode;
}
```

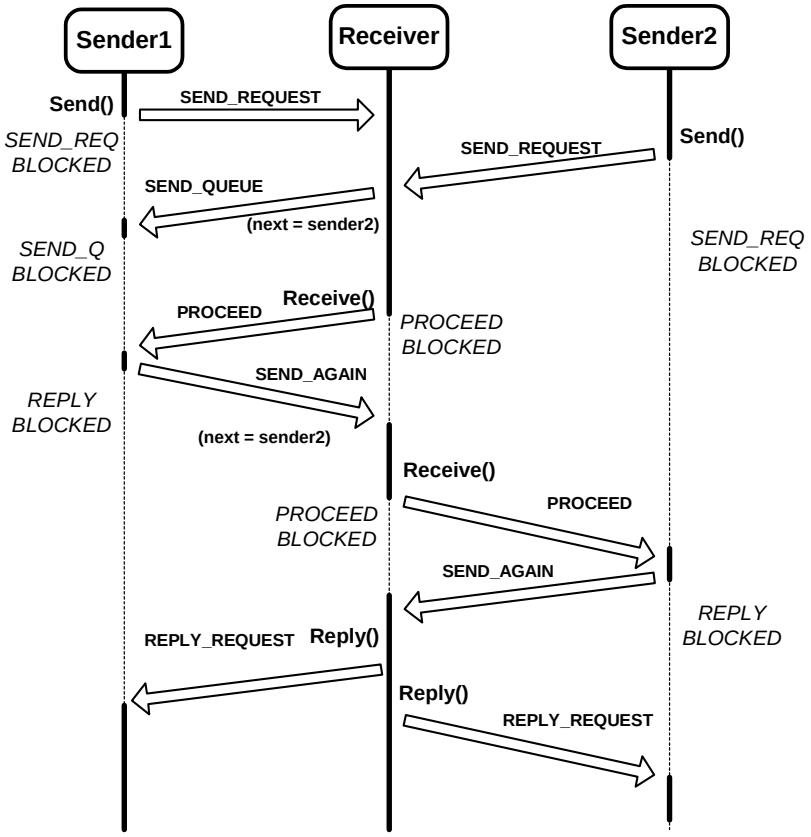
```
/*********************************************************************
 * function : receive(void)
 *********************************************************************
 */
void sysCall_receive(void) {
   ErrorCode errorCode = OK;
   Task * pTask = aKernel.pCurrentTask;
   Task * pSenderTask;
   pTask->pMsgRecv    = getParameter(1);
   pTask->msgRecvLen  = getParameter(2);

   pMsgQueue = aTDTable.getPtMsgQueue( pTask );
// sender is not ready, msgQueue is empty
   if( pMsgQueue->isEmpty() ) {
            pTask->setState(ReceiveBlocked);
   } // if
// if sender is waiting, msg queue is not empty
   else {
            pSenderTask = pMsgQueue->getNextTask();
            msgCopy(pSenderTask, pTask);
   // send task is going to reply blocked , waiting reply
            pSenderTask->setState( ReplyBlocked );
   // set Return value for receiver for sender TID
            setReturnValue(pSenderTask->getTid());
   // set myself, receiver ready and put into ready queue
      pTask->setState( Ready );
            pReadyQueue->addPriQueue( pTask );
   } // else
   //return errorCode;
}

/*
```

```
 ************************************************************************
 * function : reply(void)
 ************************************************************************
 */
void sysCall_reply(void) {
    ErrorCode errorCode = OK;
    Task * pTask = aKernel.pCurrentTask;
    Task * pSenderTask;
    Task * pTask_Rcv;
    int    highestPrioMsgQueue;
//  for return value , real copy data size
    int copyLen = 0;
    pTask_Rcv = pTask->pTaskRecv   = aTDTable.getPtTask(getParameter(1));
    // task should be replied blocked
    if(pTask_Rcv->getStateNum() == ReplyBlocked) {
                pTask->pMsgSend    = getParameter(2);
                pTask->msgSendLen  = getParameter(3);
                copyLen = msgCopy( pTask, pTask->pTaskRecv);
    // set Return value for sender copyied msg size
                setReturnValueTo(copyLen, pTask_Rcv);
    // set repliee ready and put into ready queue
                pTask_Rcv->setState( Ready );
                pReadyQueue->addPriQueue( pTask_Rcv );
    } // if
    else {
                Error(ErrorWarning,"reply to invalid task ");
    } // else
    if( !pTask->isPriorityInherited() ) {
                // set myself, replier ready and put into ready queue
                pTask->setState( Ready );
                pReadyQueue->addPriQueue( pTask );
    }  // if
// else priority is not inherited
    else {
```

```
            // if original priority is higher than queued msg task
            // then just revert priority with original priority
            pMsgQueue = aTDTable.getPtMsgQueue( pTask );
  // if msg queue is not empty
            if( !pMsgQueue->isEmpty() ) {
               highestPrioMsgQueue = pMsgQueue->getHighestPriority();
               // if original priority is higher than the highest one at queue
               if( pTask->getOriginalPriority() <= highestPrioMsgQueue ) {
               // revert priority with the original one
                        pTask->revertInheritePriority();
               // set repliee ready and put into ready queue
                        pTask->setState( Ready );
                        pReadyQueue->addPriQueue( pTask );
               } // if
               // if original priority is lower than queued msg task
               // then change priorith with that priority
               else {
                        pTask->changeInheritePriority( highestPrioMsgQueue );
               // set repliee ready and put into ready queue
                        pTask->setState( Ready );
                        pReadyQueue->addPriQueue( pTask );
               } // else
            } // if
            // if msg queue is empty
            else {
            // revert priority with the original one
               pTask->revertInheritePriority();
            // set repliee ready and put into ready queue
               pTask->setState( Ready );
               pReadyQueue->addPriQueue( pTask );
            } // else
  } // else
}
```

3.7. Distributed environment IPC

Multiple IPC will be discussed for the reference of distributed computing environment. In fact, Femos is not implemented this distributed IPC, but QNX is implemented this IPC. In Figure 9 shows multiple sends before receive, notice that there are SEND_QUEUE, PROCEED, SEND_Q_BLOCKED, PROCEED_BLOCKED.



Multiple SENDs before RECEIVE

Figure 9 Multiple Sends before Receive

4. Clock Manager

System clock, system tick manager, is using a 8254 timer interrupt every 50 mSec. Presently, Femos has system clock manager inside kernel. Basically, main job of clock manager is managing doubly linked list of client tasks, when clients call the clock related calls (Delay(), DelayUntil(), UpTime(), WakeUp()) then it adds item or removes item or compares delay time in queued task descriptor blocks with current uptime or just return uptime.

Followings are clock timer interrupt handler implementations.

```
//---------------------------------------------------------------------
// inner kernel interrupt dispatch functions (servers)
//              clock server
//---------------------------------------------------------------------
int handler_clockEvent() {
   int i=0, itemSize;
   Task * pTask;
   Task * pRmvTask;
   aKernel.incUpTime();
   aKernel.pCurrentTask->setState(Ready);
   pReadyQueue->addPriQueue( aKernel.pCurrentTask );
   pTask = pClkEvtQueue->getHead();
   while( pTask != (Task *)Empty) {
            if (!pTask->isTimeOut( aKernel.getUpTime() )) {
               pTask = pTask->pNext;
            } //if
            else {
            // if time out then remove
               pRmvTask = pTask;
               pTask = pTask->pNext;
               pClkEvtQueue->removeElement(pRmvTask);
            // set removed task to Ready and put into ready queue
               pRmvTask->setState( Ready );
               pReadyQueue->addPriQueue( pRmvTask );
            } // else
```

```
    } // while
    // notify that End Of Interrupt to ICU
    SEND_EOI;
    return i;
}
```

Followings are system function call implementations

```
/*********************************************************************
 * function : upTime(void)
 ********************************************************************* */
void sysCall_upTime( void ) {
    int i=0;
    if (aKernel.pCurrentTask == NULL) {
                Error(ErrorWarning, "NULL TaskToRun");
                asm("hlt;");
                while(1);
    } // if
// put add queue againe
    aKernel.pCurrentTask->setState( Ready );
    pReadyQueue->addPriQueue( aKernel.pCurrentTask );
    setReturnValue( aKernel.getUpTime() );
}
/*********************************************************************
 * function : delay(void)
 ********************************************************************* */
void sysCall_delay(void) {
    int errorCode = 0;
    int duration;
    Task *pTask = aKernel.pCurrentTask;
    duration = getParameter(1);
    pTask->setTimer( duration + aKernel.getUpTime() );
    pTask->setState( EventBlocked );
    //pTask->printTask();
    pClkEvtQueue->addQueue( pTask );
}
```

```
/*********************************************************************
 * function : delayUntil(void)
 ********************************************************************* */
void sysCall_delayUntil(void) {
    int errorCode = 0;
    int directDuration;
    Task *pTask = aKernel.pCurrentTask;
    directDuration= getParameter(1);
    pTask->setTimer( directDuration );
    pTask->setState( EventBlocked );
    //pTask->printTask();
    pClkEvtQueue->addQueue( pTask );
}
/*********************************************************************
 * function : WakeUp(void)
 ********************************************************************* */
 // if task is event blocked then wake it up
 // unless just return
void sysCall_wakeUp(void) {
    int errorCode = 0;
    int tid;
    Task *pRmvTask;
    tid = getParameter(1);
    pRmvTask = aTDTable.getPtTask(tid);
    if( pRmvTask->getStateNum() == EventBlocked ) {
                pClkEvtQueue->removeElement(pRmvTask);
    // set removed task to Ready and put into ready queue
                pRmvTask->setState( Ready );
                pReadyQueue->addPriQueue( pRmvTask );
    }
//  put current task into ready queue
    aKernel.pCurrentTask->setState(Ready);
    pReadyQueue->addPriQueue( aKernel.pCurrentTask );
}
```

5. Serial Manager

Serial manager is managing serial port interrupt and character sized bounded buffer for input and output. Femos has Get() and Put() for one character input/output, in addition it has Read()(with echo or without echo) and Write() for string-type input/output.

Followings are serial interrupt handler implementations.

```
//--
//  WYSE terminal
//--
int handler_serialEvent_1() {
   int i=2;
   BYTE iir, lsr;
   BYTE dummy, data, item;
   BYTE loop = 3;
   Task * pTask;
   Boolean CR_inputted = False;
// check this is for receive or transmit
   iir = inb( USART_1_BASE + USART_IIR) & 0xf;
   // first check that this is read event
   // check recieved data first
   if(iir == 0x04) { /* bit 2-1 : 0 10 0 : received data  */
//  if this is not read mode then normal operation
            if(!aSerial.isReadMode1()) {
               do {
               lsr = inb( USART_1_BASE + USART_LSR);
               if( lsr & 0x1e)          {
                        dummy = inb( USART_1_BASE + USART_LSR );
               }  //if
               else if ( LSR_Data_Ready & lsr) {
                        data = inb( USART_1_DATA );
```

```
                        if( !pBDBuf_W_I->isFull() ) {
                        // put the inputted data into bound buffer
                           pBDBuf_W_I->addItem( data );
                           //  check serial event queue
                        } // if
                    } //  else if
                    iir = inb(USART_1_BASE + USART_IIR) & 0xf;
                    } while( iir && (--loop > 0));
   // check blocked queue
                    while( !pQueue_W_I->isEmpty() &&
                                    !pBDBuf_W_I->isEmpty()) {
                        pTask = pQueue_W_I->getItem();
                        // set ready state again
                        pTask->setState( Ready );
                        pReadyQueue->addPriQueue( pTask );
                        //get the buffered data
                        item = pBDBuf_W_I->getItem();
                        setReturnValueTo( data, pTask );
                    }
                }// if read mode
//  if read mode then just read the data and put into buffer and
// check the data. if CR(=13) comes the call syscall_read_return with
// port number
                else {
                   do {
                        lsr = inb( USART_1_BASE + USART_LSR);
                        if( lsr & 0x1e)              {
                           dummy = inb( USART_1_BASE + USART_LSR );
                        }  //if
                        else if ( LSR_Data_Ready & lsr) {
                           data = inb( USART_1_DATA );
                           // if the data is not CR
                           if( data != 13)  {
                                if( !pBDBuf_W_I->isFull() ) {
```

```
                                        // put the inputted data into bound buffer
                                            pBDBuf_W_I->addItem( data );
                                            //  check serial event queue


// for read syscall echo
if( aSerial.isReady1() && pBDBuf_W_O->isEmpty() ) {
    outb( USART_1_DATA , data);
    aSerial.resetReady1();
} // if
else if(!pBDBuf_W_O->isFull()) {
    //cprintf(" in buf ");
    pBDBuf_W_O->addItem(data);
} // else if
                                        } // if
                                    }
                                    // if CR is inputted
                                    else {
                                        CR_inputted = True;
                                    }
                                } //  else if
                                iir = inb(USART_1_BASE + USART_IIR) & 0xf;
                        } while( iir && (--loop > 0) && !CR_inputted);
                        // if cr is inputted the call clearence function
                        // of Read syscall
                        if(CR_inputted) {
                                sysCall_read_return( WYSE );
                        }
                } // processing Read syscall
    } // if
///
/// bit 2-1 : 0 01 0 : THRE
/// means output
///
    else if(iir == 0x02) {
```

```
            if(!pBDBuf_W_O->isEmpty()) {
               data = pBDBuf_W_O->getItem();
               outb( USART_1_DATA , data);
            }
            else {
               // in case of empty set ready flag
               aSerial.setReady1();
            }
            //  NEED TO DO waiting list processing
      } // else if
      aKernel.pCurrentTask->setState(Ready);
      pReadyQueue->addPriQueue( aKernel.pCurrentTask );
      SEND_EOI;
      return i;
   }
```

6. Reference

- **Prof. Charlie Clarke**, *CS 452/652 Real-Time Programming Course work*, Summer 2001, University of Waterloo, ON, Canada

- **Douglas Comer**, *Operating System Design – The XINU approach*, 1984, Prentice-Hall, Inc

- **Ian Bull, John Casey, Peter D. Gray, etc**. *CS452/652 Real-Time Programming Lab Manual*, Summer 2001, University of Waterloo, ON, Canada

- **Jean J. Labrosse**, *uC/OS The Real-Time Kernel*, 1992, R & D Publications

- **Michael Barr**, *Programming Embedded Systems in C and C++*, 1999, O'Reilly & Associates

- *QNX System Architecture Reference* (http://www.qnx.com/literature/qnx_sysarch/index.html)

- **Dr. Yunho Jeon**, C.T.O. of Zestel Co, Ltd. Korea, Personal correspondence about QNX-style IPC.