ECED 4402 Real Time System

Assignment 2 Design document

Victor Gao B00677182

Ray Su B00634512

Introduction:

In this assignment, we are asking to build a simple kernel for control and switch between the processes based on their priorities. There are also some support functions for kernel and processes.

For Kernel:

The Kernel data structures is an array (queue) which have 5 priorities that handles each priority's function pointer. Therefore, the pointer in array will only point to one of the process which has same priority.

Kernel also has several functions to support with, such as get id, terminate, switch process and nice, etc.

For process:

All processes are initialized and registered at first time, and then kernel kicks in to run the process.

There are some functions support for processes, such as send message, receive message and binding message to implement the inter-process communications.

For Devices:

The best way to know if the switching between processes works is to let them output something. And in this assignment, the UART IO will be used to implements that and show on screen.

Furthermore, the SYSTICK will send message to Time sever in order to record the time.

Data Dictionary

Stack	= *The structure of stack, which contains R4 to R11, R1, R2, R3, R12, LR, PC and PSR*
R4 to R11	= *Member of Stack, also is a CPU register. Registers R4 to R11, which are software pushed to stack*
R1, R2, R3, R12	=* Member of Stack, also is a CPU register. Register R1, R2, R3, R12, which are hardware pushed to stack*
LR	=* Member of Stack, also is a CPU register, also is a CPU register. Link register*
PC	=* Member of Stack, also is a CPU register. Program counter*
PSR	=* Member of Stack, also is a CPU register. Program status register*
PCB	= * The struct of process control block, which contains Next, Previous, PSP, Priority, MBP, PID, Msg_Wait and Mbx_Wait*
Next	= * A member of struct PCB. The PCB pointer points to next linked process's PCB *
Prev	= * A member of struct PCB. The PCB pointer points to previous linked process's PCB *
PSP	= * A member of struct PCB. Process stack pointer *
Priority	= * A member of struct PCB. The priority of process *
MBP	= * * A member of struct PCB. Mail box pointer, stores the address of one bound mail box, NULL if no mail box is bound to this process *
PID	= * A member of struct PCB. Stores process's ID value *
Msg_Wait	= * A member of struct PCB. Stores the RecvMsgArgs pointer of the message that process is waiting to receive. Used for unblock. If value is NULL, the process is unblocked, otherwise is blocked*
Mbx_Wait	= * A member of struct PCB, stores the mailbox number of the mailbox waiting to receive. Used for unblock.*
PRIORITY_LIST	= * An array of PCB pointers with size of 6, which stores last running process's PCB pointer in each priority. PRIORITY_LIST[0] will be the idle process. PCB pointers of priority 1 to 5 will be stored in PRIORITY_LIST [1] to PRIORITY_LIST [5] correspondingly. If there is no process exist in certain priority, the NULL value will be stored in corresponding PRIORITY_LIST position. *
RUNNING	= * A global PCB pointer which point to current running process's PCB*
KcallArgs	= * The structure of kernel call arguments, which includes Code, RtnValue, Arg1 and Arg2*
Code	= * A member of struct KcallArgs, stores the enum value of KcallCode of action to do in kernel call *

RtnValue = * A member of struct KcallArgs, stores the return value return from kernel call*

Arg1 = * A member of struct KcallArgs, stores argument 1*

Arg2 = * A member of struct KcallArgs, stores argument 2*

KcallCode = * enum of kernel code, includes GETID, NICE, TERMINATE, SEND, RECV*

GETID = * A member of enum KcallCode, to do get ID action in kernel *

NICE = * A member of enum KcallCode, to do nice action in kernel*

TERMINATE = * A member of enum KcallCode, to do terminate action in kernel *

SEND = * A member of enum KcallCode, to do send message action in kernel *

RECV = * A member of enum KcallCode, to do receive message action in kernel *

FIRST_PROCESS = * The flag indicated the first time run to kernel *

SendMsgArgs = * The structure of send message argument, has the member of Recver, Sender, Msg_addr and Sz*

RecvMsgArgs = * The structure of send message argument, has the member of Recver, Sender, Msg_addr and Sz *

Recver = * A member of struck SendMsgArgs and RecvMsgArgs. Mailbox number of receiver*

Sender = * A member of struck SendMsgArgs and RecvMsgArgs. Mailbox number of the sender, value type in SendMsgArgs, pointer type in RecvMsgArgs*

Msg_addr = * A member of struck SendMsgArgs and RecvMsgArgs. The address of message*

Sz = * A member of struck SendMsgArgs and RecvMsgArgs. Size of the message*

Send_Err_Msg = * A enum of error return from send function. Includes RECV_ERROR, RECV_FULL, SEND_ERROR*

Recv_Err_Msg = * A enum of error return from receive function. Includes UNBINDED, SEND_ERROR *

RECV_ERROR = * A member of enum Send_Err_Msg, receiver's mailbox not exist*

RECV_FULL = * A member of enum Send_Err_Msg, receiver's mailbox is full*

SEND_ERROR = * A member of enum Send_Err_Msg and Recv_Err_Msg, the send mailbox is invalid*

UNBINDED = * A member of enum Recv_Err_Msg, the mailbox to check receive does not belong to this process*

MAILBOX_LIST = * A global array of Mailbox, each Mailbox has a number*

Mailbox = * The structure of Mailbox, includes MBX_Prev, MBX_Next, Message_ptr, Pcb_Ptr and Msg_Tail*

MBX_Prev = * A member of structure Mailbox, used to point the Mailbox that belong to same PCB*

MBX_Next = * A member of structure Mailbox, used to point the Mailbox that belong to same PCB*

Message_ptr = * A member of structure Mailbox, used to store the address of the first Message*

Pcb_Prt = * A member of structure Mailbox, the PCB pointer of bind process*

Msg_Tail = * A member of structure Mailbox, the pointer of Message points to the tail of message queue *

Message = * A member of structure Mailbox, the structure of message, includes the Msg_Next, Message_Addr, Size and Sender*

Message_Addr = * A member of structure Message, the address of message*

Size = * A member of structure Message, the size of message*

Sender = * A member of structure Message, the mailbox number of sender's mailbox*

Msg_Next = * A member of structure Message, the pointer of Message points to the next of message in the queue *

UNBLOCK_PRIORITY = * A global variable of the priority of unblocked process. Value will be cleared to 0 when process unblocking has been handled.*

CPU state diagram:(on system level)

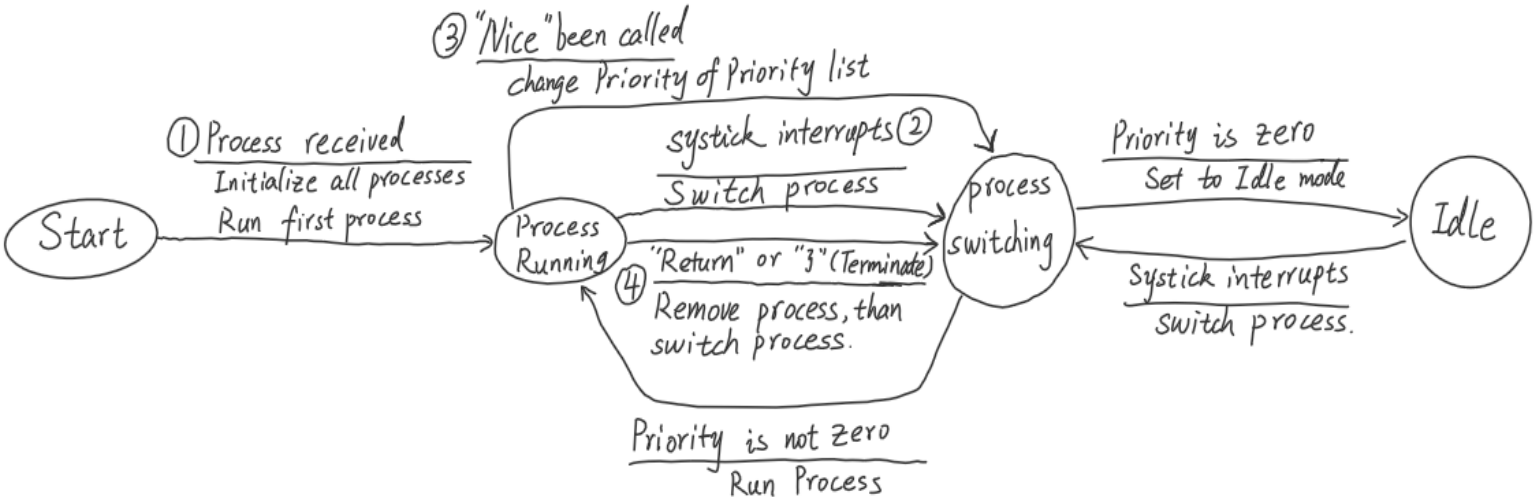


Figure 1

Notes:

1> Since both the NICE and Terminate are checking priority list and might change it, we draw it on graph. The GetID() function does not change anything, therefore, it is not on graph.

2> The processes send and receive messages is on process level, it will be explained another section.

3> The label of each action (1-4) will be explained with more details.

Process Initialization Structure English (part ① in CPU state diagram)

**** In kernel space****

ALLOCATE memory of PRIORITY_LIST

DO WHILE have process need to be initialized

    ALLOCATE memory of Stack

    ALLOCATE memory of PCB

    STORE the address of Stack to PSP

    STORE the address of process's function to PC in Stack

    STORE the address terminate function to LR in Stack

    ASSIGN process ID to PID in this PCB

    ASSIGN priority to Priority in this PCB

    ENQUEUE process to correct PROCESS QUEUE

    * Call EnqueueProcess function, see structure English of EnqueueProcess function*

END DO-WHILE

ASSIGN HIGHEST priority process to RUNNING

ASSIGN TRUE to FIRST_PROCESS

INITIALIZE KERNEL

CALL KERNAL

Structure English of Switch Process Kernel Call (part ② in CPU state diagram)

IF UNBLOCK_PRIORITY GREATER than RUNNING PRIORITY THEN

if a process been unblocked and has a greater priority then current process

ASSIGN PRIORITY_LIST [UNBLOCK_PRIORITY] to RUNNING

both PRIORITY_LIST [#] and running are the pointers point to PCBs

ELSE * no process is unblocked or unblocked process's priority is smaller than current priority*

ASSIGN Next of RUNNING PCB to RUNNING * running = running->next*

END IF

ASSIGN 0 to UNBLOCK_PRIORITY *reset the unblock priority*

GET PSP *load new PSP form new PCB to CPU*

PULL R4 to R11 From Stack of RUNNING PCB

MOVE RUNNING PCB's PSP to the R0 of RUNNING PCB's Stack *TOP of PCB stack now is R0*

ASSIGN RUNNING PCB's PSP to CPU PSP

Structure English of Nice (part ③ in CPU State Diagram)

**** Take new priority value as input****

****in process space****

ASSIGN the value of new priority to Arg1 of KcallArgs

ASSGN Code of KcallArgs to NICE

ASSIGN the address of KcallArgs to CPU R7

CALL KERNEL

****in kernel space****

GET address of KcallArgs from R7

GET new priority value from Arg1 of KcallArgs

STORE Priority of RUNNING to old priority

* old priority is a local variable that save the process priority before get changed*

Dequeue RUNNING from old process queue *See structure English of EnqueueProcess function*

ASSIGN new priority value to Priority of RUNNING

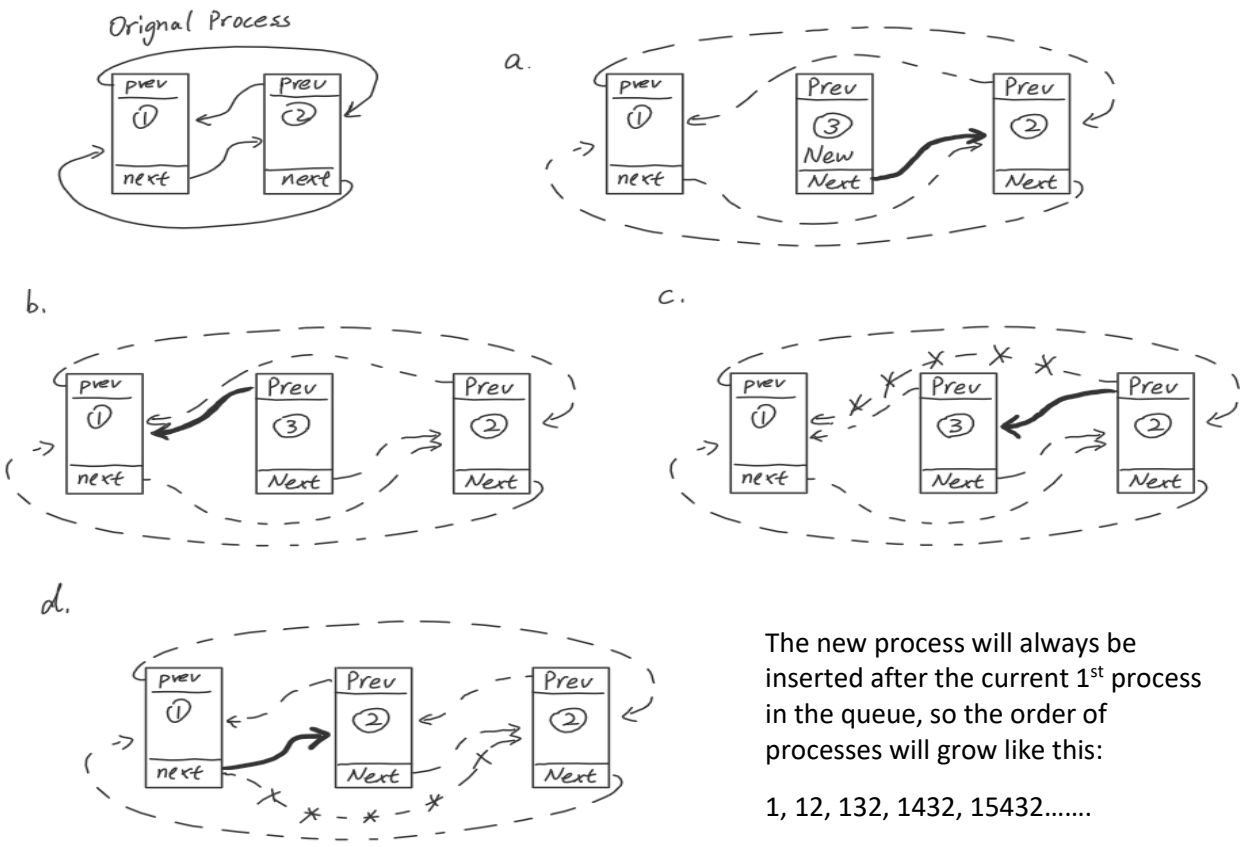
Enqueue RUNNING to new process queue *See structure English of DequeueProcess function*

```
        IF new priority LARGER than old priority THEN
                RUN PRIORITY_LIST[new priority]
        ELSE
                IF old priority is not empty
                        ASSIGN PRIORITY_LIST[old priority] to RUNNING
                ELSE
                        DO check lower priority process until process is found
                        ASSIGN PRIORITY_LIST[lower priority] to RUNNING
                END-IF
        END IF
```

Structure English of EnqueueProcess function

```
****Take PCB pointer as input****
        IF IS the FIRST process in the priority queue THEN * PRIORITY_LIST [PCB->Priority]==NULL*
                ASSIGN address of new PCB to PREV of new PCB  *PCB->Prev=PCB*
                ASSIGN address of new PCB to NEXT of new PCB  *PCB->Next=PCB*
                ASSIGN address of new PCB to PRIORITY_LIST * PRIORITY_LIST [PCB->Priority]=PCB*
        ELSE *Enqueue to existing queue*
                ASSIGN address of Next of PRIORITY_LIST [PCB->Priority] to Next of new PCB
                *PCB->Next= PRIORITY_LIST [PCB->Priority]->NEXT, see part a in Figure 2*
                ASSIGN address of PRIORITY_LIST [PCB->Priority] to Prev of new PCB
                *PCB->Prev= PRIORITY_LIST [PCB->Priority], see part b in Figure 2*
                ASSIGN address of new PCB to Next of Prev of PRIORITY_LIST [PCB->Priority]
                * PRIORITY_LIST [PCB->Priority]->Next->Prev=PCB, see part c in Figure 2*
                ASSIGN address of new PCB to Next of PRIORITY_LIST [PCB->Priority]
                * PRIORITY_LIST [PCB->Priority]->next=PCB, see part d in Figure 2*
```

Data structure of adding process to existed process queue



The new process will always be inserted after the current 1st process in the queue, so the order of processes will grow like this:

1, 12, 132, 1432, 15432…….

Figure 2

Structure English of DequeueProcess function

****Take PCB pointer of the process to remove as input****

IF Next of this PCB is this PCB THEN * PCB->Next == PCB*

    ASSIGN NULL to PRIORITY_LIST[current priority] * PRIORITY_LIST [priority] = NULL*

ELSE

    ASSIGN Next of RUNNING to Next of Prev of RUNNING

    * RUNNING->Prev->Next = RUNNING->Next*

    ASSIGN Prev of RUNNING to Prev of Next of RUNNING

    RUNNING->Next->Prev=RUNNING->Prev

END IF

Structure English of Terminate (part ④ in CPU State Diagram)

RUN CheckLowerPriorityProcess function * see structure English of CheckLowerPriorityProcess *

ASSIGN CheckLowerPriorityProcess return value to next_to_run

next_to_run is a local variable to store next process to run

DEQUEUE RUNNING from process queue *See structure English of DequeueProcess function*

FREE the memory of Stack of RUNNING

FREE the memory of PCB of RUNNING

ASSIGN next_to_run to RUNNING

Structure English of CheckLowerPriorityProcess function

IF Next of RUNNING is NOT RUNNING THEN *not the only process in current process queue*

    ASSIGN RUNNIGN to next_to_run *next_to_run is a local variable to store next process to run*

ELSE

    DO check lower priority process until process is found

    idle process will be the lowest priority process

    ASSIGN PRIORITY_LIST[lower priority] to next_to_run

END IF

RETURN next_to_run

State Diagram of process:

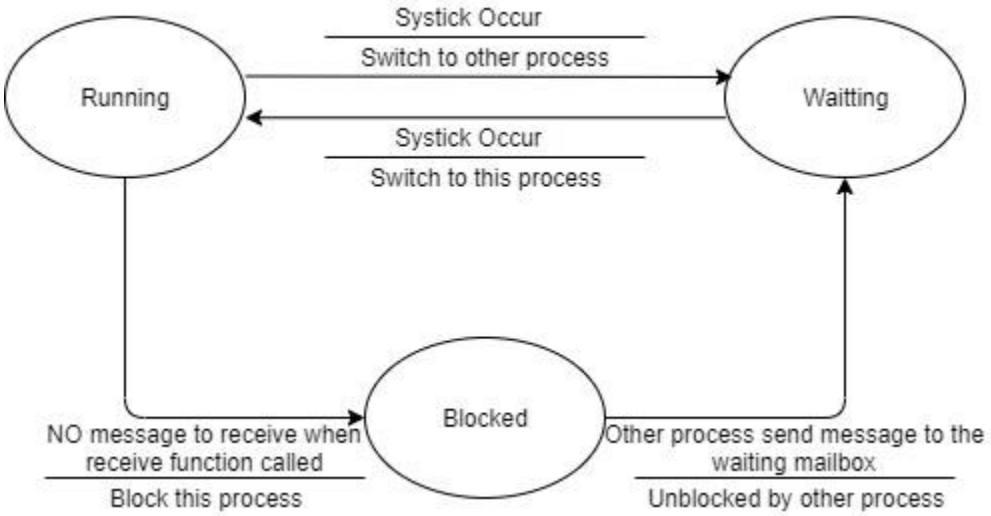


Figure 3

Diagram of data structure of the Mailbox:

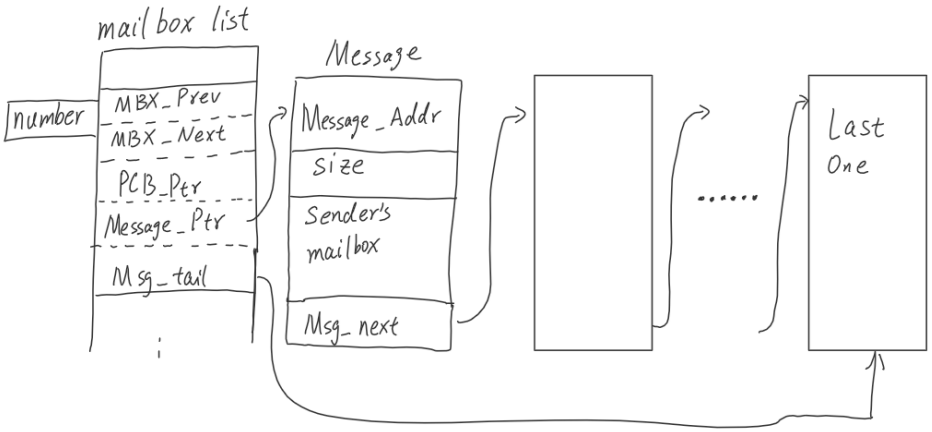


Figure 4

Structure English of Send function:

****Take receiver mailbox number, sender mailbox number, message's address and message's size****

****process space****

CREATE SendMsgArgs by using receiver mailbox number, sender mailbox number, message's address and message's size

ASSIGN the address of SendMsgArgs to Arg1 of KcallArgs

ASSGN Code of KcallArgs to SEND

ASSIGN the address of KcallArgs to CPU R7

CALL KERNEL

****kernel space****

GET KcallArgs from R7

GET SendMsgArgs from Arg1 in KcallArgs

IF Sender's mailbox belong to RUNNING THEN

* MAILBOX_LIST[SendMsgArgs->Sender].PCB_Ptr == RUNNING *

IF PCB pointer in receiver's Mailbox is NOT NULL THEN* Mailbox is valid, owned by receiver*

IF receiver's process is blocked and waiting on this Mailbox or any mailbox THEN

* MAILBOX_LIST[SendMsgArgs->Recver].PCB_Ptr-> Msg_Wait != NULL, so receiver is blocked*

* MAILBOX_LIST[SendMsgArgs->Recver].PCB_Ptr-> Mbx_Wait ==SendMsgArgs->Recver, so receiver is waiting on this mailbox*

* MAILBOX_LIST[SendMsgArgs->Recver].PCB_Ptr-> Mbx_Wait ==ANY (-1), so receiver is waiting on any mailbox belong to it*

GET receiver's Msg_Wait pointer from receiver's PCB

ASSIGN SMALLER value between Sz in SendMsgArgs and Size in Msg_Wait to CopySz

Take smaller size as copy size, CopySz is a local variable

COPY CopySz bytes from Msg_addr in SendMsgArgs to Message_Addr in Msg_Wait

ASSIGN CopySz to Size in Msg_Wait

ASSIGN CopySz to RtnValue in KcallArgs *For return*

SET NULL to Msg_Wait in receiver's PCB

* MAILBOX_LIST[SendMsgArgs->Recver].PCB_Ptr-> Mbx_Wait = NULL*

UNBLOCK the process pointed by receiver's Mailbox's PCB

* See structure English Unblock function*

ELSE *is unblocked or not waiting on this mailbox*

CREATE a Message and save memory at Message_Addr with Sz in SendMsgArgs

COPY bytes from Msg_addr in SendMsgArgs to the Message_Addr of Message

ASSIGN the value of Sz in SendMsgArgs to Size of Message

ASSIGN the Sender in SendMsgArgs to Sender of Message

* Save sender's mailbox number to message*

ENQUEUE Message to Message queue in receiver's Mailbox

add message to the end of message queue. Mailbox.tail->Next=&Message, tail=&Message

END-IF

ELSE

ASSIGN RECV_ERROR to RtnValue in KcallArgs

Assign error code receiver mailbox is invalid

END-IF

ELSE

ASSIGN SEND_ERROR to RtnValue in KcallArgs *Assign error code sender mailbox is invalid*

END IF

****process space***

RETURN RtnValue in KcallArgs

Structure English of Receive:

****Take receiver mailbox number, sender mailbox address, message's address and message's size****

****process space****

CREATE SendMsgArgs by using receiver mailbox number, sender mailbox address, message's address and message's size

ASSIGN the address of RecvMsgArgs to Arg1 of KcallArgs

```
ASSGN Code of KcallArgs to RECV
ASSIGN the address of KcallArgs to CPU R7
CALL KERNEL
****kernel space****
GET KcallArgs from R7
GET RecvMsgArgs from Arg1 in KcallArgs
IF receiver's Mailbox's PCB pointer NOT EQUAL to NULL THEN*Mailbox is bound to receiver*
        IF Message_ptr in Mailbox NOT EQUAL to NULL THEN*Mailbox has received message*
                ASSIGN SMALLER value between Sz in RecvMsgArgs and Size in the first Message in
                Mailbox to CopySz
                *Take smaller size as copy size, CopySz is a local variable*
                COPY CopySz bytes from the Msg_addr in the first Message in the Mailbox to the
                Msg_addr in RecvMsgArgs
                * Copy the message from mailbox to destination*
                ASSIGN CopySz to RtnValue in KcallArgs
                ASSIGN Msg_Next to Message_ptr
                FREE first Message
        ELSE *Mailbox has no message*
                ASSIGN RecvMsgArgs to Msg_Wait in RUNNING PCB
                * RUNNUNG-> Msg_Wait  = RecvMsgArgs *
                ASSIGN Recver in RecvMsgArgs to Mbx_Wait in RUNNING PCB
                * RUNNUNG-> Mbx_Wait  = RecvMsgArgs->Recver *
                BLOCK RUNNING *See structure English of Block function*
ELSE *Mailbox has no owner *
        ASSIGN UNBINDED to RtnValue in KcallArgs *Assign error code*
END-IF
****process space***
RETURN RtnValue in KcallArgs
```

Structured English of Block

RUN FindNextProcessToRun function * see structure English of FindNextProcessToRun *

ASSIGN FindNextProcessToRun return value to next_to_run

next_to_run is a local variable to store next process to run

DEQUEUE RUNNING from process queue *See structure English of DequeueProcess function*

ASSIGN next_to_run to RUNNING

Structured English of Unblock

****Take PCB pointer of process to unblock****

Enqueue process to unblock to new process queue *See structure English of DequeueProcess function*