SE350 RTX Project Documentation



**University of Waterloo**

**3A Software Engineering**

**3/10/2015**

Angad Kashyap (20466050), Mike Wang (20465276), Tong Tong (20460179) and Vishal Bollu (20474731)

[1 Introduction 4](#_Toc415138857)

[2 Software Design Description 5](#_Toc415138858)

[2.1 INITIALIZATION 5](#_Toc415138859)

[2.1.1 Memory Initialization 5](#_Toc415138860)

[2.1.2 Process Initialization 6](#_Toc415138861)

[2.2 MEMORY MANAGEMENT 7](#_Toc415138862)

[2.2.1 Requesting Memory 7](#_Toc415138863)

[2.2.2 Releasing Memory 8](#_Toc415138864)

[2.3 PROCESSOR MANAGEMENT 9](#_Toc415138865)

[2.3.1 Scheduling 9](#_Toc415138866)

[2.3.1 Context Switching 9](#_Toc415138867)

[2.3.2 Process Priority 10](#_Toc415138868)

[2.4 INTERPROCESS COMMUNICATION 11](#_Toc415138869)

[2.4.1 Sending Messages 11](#_Toc415138870)

[2.4.2 Receiving Messages 12](#_Toc415138871)

[2.4.3 Non-Blocking Message Receiving 12](#_Toc415138872)

[2.4.4 Delayed Messages 12](#_Toc415138873)

[2.5 INTERRUPT PROCESSES 13](#_Toc415138874)

[2.5.1 Timer I-Process 13](#_Toc415138875)

[2.5.2 UART I-Process 14](#_Toc415138876)

[2.5.2.1 Hotkeys 15](#_Toc415138877)

[2.6 SYSTEM PROCESSES 17](#_Toc415138878)

[2.6.1 Null Process 17](#_Toc415138879)

[2.6.2 KCD Process 17](#_Toc415138880)

[2.6.3 CRT Display Process 18](#_Toc415138881)

[2.7 USER PROCESSES 18](#_Toc415138882)

[2.7.1 24 Hour Wall Clock Display Process 18](#_Toc415138883)

[2.7.2 Set Priority Command Process 20](#_Toc415138884)

[2.7.3 Stress Tests 20](#_Toc415138885)

[3 Timing Analysis 20](#_Toc415138886)

[4 Lessons Learned 21](#_Toc415138887)

Table of Contents

1 Introduction

[Insert introduction here describing the purpose of the project, the objectives including an overview of the capabilities of the resulting RTX]

2 Software Design Description

# 2.1 INITIALIZATION

## 2.1.1 Memory Initialization

This method is responsible for allocating memory for PCBs and setting up memory management. It starts with placing 4 bytes of padding at the bottom of the usable RAM (adding 4 bytes to the p\_end pointer starting at the lower address). Memory is then allocated for the following:

* gp\_pcbs an array of pointers to pcbs (array size is equal to number of processes)
* PCBs (the memory address for each pcb is then stored stored in gp\_pcbs array)
* gp\_pcb\_nodes an array of pcb\_node pointers (array size is equal to number of processes)
* PCB\_NODEs (the memory for each pcb\_node is stored in the gp\_pcb\_nodes array)

After allocating memory for the processes, memory management is setup. The pointer beginMemMap is an array with the size equal to the number of memory blocks. This array represents whether a memory block has been taken are not (0 represents free and 1 represents taken). The pointer beginHeap contains the address of memory after space for beginMemMap has been allocated. This address stored in beginHeap represents the start of the heap.

A global stack pointer is created in preparation for the call to alloc\_stack. The global stack pointer contains the high address of the RAM and is used for allocating stack space for the processes.

void memory\_init(void)

{

lowAddrOfRAM ← &Image$$RW\_IRAM1$$ZI$$Limit;

// calculate beginning and end address of PCB pointers

globalPCBs ← lowAddrOfRAM;

lowAddrOfRAM ← lowAddrOfRAM + NUM\_PROCS \* sizeof(PCB \*);

allocate memory and assign pointers for each PCB;

// calculate beginning and end address of PCB node pointers

globalPCBNodes ← lowAddrOfRAM;

lowAddrOfRAM ← lowAddrOfRAM + NUM\_PROCS \* sizeof(PCB\_NODE \*);

allocate memory and assign pointers for each PCB node;

prepare for alloc\_stack() to allocate memory for stacks;

// Calculate beginning of heap pointers

beginningOfMemoryMap ← lowAddrOfRAM;

beginningOfHeap ← beginningOfMemoryMap + NUM\_OF\_MEMBLOCKS;

initialize each memory block flag with 0; // indicating free blocks

}

## 2.1.2 Process Initialization

This method begins with placing all of the information related to user processes as well as the null process into a global process table. This information includes the PID, the priority, the stack size and the starting PC value for the process. New PCBs are created for each of the processes in the global process table. Here, stack space is allocated for the PCBs using alloc\_stack. At the same time, PCB nodes are created for each process to be used in the ready priority queue and the blocked queue. The ready priority, blocked on receive and blocked on memory queues are both null-initialized. Then the PCB nodes of processes which are not interrupt handlers are enqueued in the ready priority queue according to their priorities.

void process\_init()

{

null process set to globalProcTable[0];

user process 1 set to globalProcTable[1];

user process 2 set to globalProcTable[2];

user process 3 set to globalProcTable[3];

user process 4 set to globalProcTable[4];

user process 5 set to globalProcTable[5];

user process 6 set to globalProcTable[6];

stress test A set to globalProcTable[7];

stress test B set to globalProcTable[8];

stress test C set to globalProcTable[9];

set priority command process set to globalProcTable[10];

wall clock process set to globalProcTable[11];

KCD process set to globalProcTable[12];

CRT process set to globalProcTable[13];

timer i-process set to globalProcTable[14];

UART i-process set to globalProcTable[15];

// initialize exception stack frame (i.e. initial context) for each process

for i = 0 to NUM\_PROCS – 1 do {

globalPCBs[i].pid ← globalProcTable[i].pid;

globalPCBs[i].state ← NEW;

globalPCBs[i].priority ← globalProcTable[i].priority;

globalPCBs[i].env\_q.head ← NULL; // Message queue init

globalPCBs[i].env\_q.tail ← NULL;

sp ← alloc\_stack(globalProcTable[i].stack\_size);

decrement sp;

globalPCBs[i].mp\_sp ← sp;

globalPCBNodes[i].pcb ← globalPCBs[i];

}

set all ready queues to be empty;

add all of the processes to the appropriate ready queue;

set the blocked queue to be empty;

set all blocked on memory queues to be empty;

initialize globalKeyboardCommands to be empty;

}

# 2.2 MEMORY MANAGEMENT

## 2.2.1 Requesting Memory

This method checks whether all of the memory blocks are in use. As long as all of the memory blocks are in use, the current process is pushed into the blocked queue and k\_release\_processor is called so that a process with ready state can run. If there are memory blocks available, a memory block is chosen, its status in the memory map is set to taken and the address of the memory block is returned.

void \*k\_request\_memory\_block (void) {

rVoid ← beginningOfHeap;

atomic (on);

while(memory is empty)

{

k\_block\_current\_processs();

atomic (off);

k\_release\_processor();

atomic (on);

}

for i = 0 to NUM\_OF\_MEMBLOCKS - 1 do {

if (beginningOfMemoryMap[i] == 0)

{

beginningOfMemoryMap[i] = 1;

break;

}

}

atomic (off);

return (void\*) (rVoid + i \* MEMORY\_BLOCK\_SIZE);

}

## 2.2.2 Releasing Memory

This method checks the validity of the memory block pointer. If it is a valid pointer that points to the beginning of a memory block, then the block is updated with an available status. The first process with the highest priority is removed from the blocked queue (if such a process exists) and is added to the appropriate ready queue. k\_release\_processor is called to see if the process needs to be switched.

int k\_release\_memory\_block(void\* memoryBlock) {

atomic (on);

if (memoryBlock is NULL or outside allocated memory space) {

return RTX\_ERR;

}

index ← (memoryBlock - beginningOfHeap) / MEMORY\_BLOCK\_SIZE;

beginningOfMemoryMap[index] = 0;

atomic (off);

k\_ready\_first\_blocked(); // unblock the first process blocked on memory

k\_release\_processor();

return RTX\_OK;

}

# 2.3 PROCESSOR MANAGEMENT

## 2.3.1 Scheduling

This method first checks if the uart\_preemption\_flag is set. If it is, then the scheduler returns the uart interrupt handler. If not then it iterates through the ready priority queue and dequeues the PCB node with the highest priority with a first-in first-out policy. The method then returns a pointer to this PCB node. Note that the null process is in the ready priority queue at the lowest level.

PCB\* scheduler(void) {

// check if there are any system processes ready to run

if(readyPriorityQueue[SYS\_PROC] is not empty){

return (dequeue readyPriorityQueue[SYS\_PROC]).pcb;

}

// check the user procs (last/default is the null process)

// return the PCB of the highest priority

for i = 0 to 4 do {

if(readyPriorityQueue[i] is not empty){

return (dequeue ready\_priority\_queue[i]).pcb;

}

}

return NULL;

}

## 2.3.1 Context Switching

This method keeps track of the current running process as the old process. If this process is not null and its state is RUN, then its state will be changed to RDY and the process' PCB will be enqueued in the appropriate ready queue. The method changes the current process pointer to the new process as returned by the scheduler method. If the previous process is NULL, which is the case when it is first run, the old process is set to the result of the scheduler. Then process\_switch is called on the old process. After the process switch, the new process' state is set to RUN.

int k\_release\_processor(void) {

oldPCB ← globalCurrentProcess;

if (globalCurrentProcess is not NULL)

{

if(globalCurrentProcess.state is RUN){

globalCurrentProcess.state ← RDY;

enqueue globalCurrentProcess' PCB node onto readyPriorityQueue;

}

}

globalCurrentProcess ← scheduler(); // get the next process to run

if ( globalCurrentProcess is NULL ) {

globalCurrentProcess ← oldPCB; // revert back to the old process

return RTX\_ERR;

}

if (oldPCB is NULL ) {

oldPCB ← globalCurrentProcess;

}

process\_switch(oldPCB); // context switch

globalCurrentProcess.state ← RUN;

return RTX\_OK;

}

## 2.3.2 Process Priority

This method takes in a PID and a new priority value, and changes the priority of the process with the associated PID. If the process does not have a blocked state, the method also dequeues the process from the ready priority queue and re-inserts it back into the appropriate queue. If the newly set priority has a higher priority than the currently running process then k\_release\_processor to preempt the process with the higher process.

int k\_set\_process\_priority(int pid, int priority) {

if (pid is invalid or priority is invalid) {

return RTX\_ERR;

}

node ← globalPCBNodes[pid];

if(node.pcb.priority is not priority) {

if (node.pcb.state is not BLOCKED\_ON\_MEMORY and node.pcb is not

globalCurrentProcess) {

remove node from ready readyPriorityQueue;

node.pcb.priority ← priority;

enqueue node onto readyPriorityQueue;

}

node.pcb.priority ← priority;

k\_release\_processor();

}

return RTX\_OK;

}

This method takes in a PID and returns the priority value for the node with that PID. The method returns RTX\_ERR if the process with the given PID is not found.

int k\_get\_process\_priority(int pid) {

node ← globalPCBNodes[pid];

if (node is NULL) {

return RTX\_ERR;

}

return node.pcb.priority;

}

# 2.4 INTERPROCESS COMMUNICATION

## 2.4.1 Sending Messages

This method accepts, a process Id and a pointer to an ENVELOPE. The envelope is enqueued to the message queue of the target process. If the target process is in the blocked on receive queue, it is removed and added to the ready queue. If preemption flag is on and target process priority is greater current process priority then k\_release\_processor causing the current process to switch to the target process. The preemption flag is set by the timer\_i\_proc (timer interrupt handler). When timer\_i\_proc is sending messages that are due, it should not cause preemption until all of the expired messages are sent therefore preemption flag is set to false.

int k\_send\_message (int targetPID, void\* msgEnvelope) {

atomic (on);

enqueue msgEnvelope onto targetProcess.msgQueue;

if (targetProcess.state is BLOCKED\_ON\_RECEIVE) {

remove targetProcess from blockedOnReceiveList;

targetProcess.state ← READY;

enqueue targetProcess onto readyPriorityQueue;

if (targetProcess.priority > currentProcess.priority){

k\_release\_processor ();

}

}

atomic (off);

return RTX\_OK;

}

## 2.4.2 Receiving Messages

This process accepts a pointer to an Integer which represents sender ID. If the current process (the process calling this function) has an empty message queue, its blocked. The state of the current process is changed to blocked on receive and it gets added to the blocked on receive list. Then k\_release\_processor is called so that a new process can start running. If the current process does not have an empty message queue, then dequeue the first message. Then set the sender ID parameter to point to the location of the senderId.

void\* k\_receive\_message (int\* senderID) {

atomic (on);

get currentProcess;

while (currentProcess.msgQueue is empty) {

currentProcess.state ← BLOCKED\_ON\_RECEIVE;

add targetProcess to blockedOnReceiveList;

k\_release\_processor ();

}

msgEnvelope ← dequeue currentProcess.msgQueue;

senderID ← msgEnvelope.senderPID;

atomic (off);

return (void\*) msgEnvelope;

}

## 2.4.3 Non-Blocking Message Receiving

void\* k\_non\_block\_receive\_message(int destinationID){

atomic (on);

get targetProcess using destinationID;

msgEnvelope ← dequeue currentProcess.msgQueue;

atomic (off);

return (void\*) msgEnvelope;

}

## 2.4.4 Delayed Messages

int k\_delayed\_send (int targetPID, void\* msgEnvelope, int delay) {

atomic (on);

msgEnvelope.delay ← globalTimerCount + delay;

k\_send\_message (TIMER\_PID, msgEnvelope);

atomic (off);

return RTX\_OK;

}

# 2.5 INTERRUPT PROCESSES

[Insert description]

## 2.5.1 Timer I-Process

void timer\_i\_proc (void) {

atomic (on);

LPC\_TIM0.IR ← (1 << 0);

envelope ← k\_non\_blocking\_receive\_message (TIMER\_PID);

while (envelope not NULL) {

insert envelope into timeoutQueue in sorted order by delay;

envelope ← k\_non\_blocking\_receive\_message (TIMER\_PID);

}

while (timeoutQueue.head is not NULL and

timeoutQueue.head.delay <= globalTimerCount) {

cur ← dequeue timeoutQueue;

k\_send\_message (cur.destination\_pid, cur);

}

globalTimerCount++;

atomic (off);

k\_release\_processor ();

}

## 2.5.2 UART I-Process

void uart\_i\_proc (void) {

atomic (on);

if (UART0.IIR is in read status) {

envelope ← k\_request\_memory\_block ();

envelope.message\_type ← MSG\_CRT\_DISPLAY;

envelope.message ← characterRead;

k\_send\_message (CRT\_PID, envelope);

if (characterRead is not '\r') {

put characterRead in globalInputBuffer;

} else {

put end of line character in globalInputBuffer;

envelope ← k\_request\_memory\_block ();

envelope.message\_type ← MSG\_CONSOLE\_INPUT;

envelope.message ← globalInputBuffer;

k\_send\_message(KCD\_PID, envelope);

}

} else if (UART0.IIR is in write status) {

if (globalCurrentEnvelope is NULL) {

globalCurrentEnvelope ← k\_non\_block\_receive\_message(UART\_IPROC\_PID);

}

if (globalCurrentEnvelope is not NULL) {

UART0.THR ← globalCurrentEnvelope.message [globalOutCharacterCounter];

globalOutCharacterCounter ++;

if (globalCurrentEnvelope.message [globalOutCharacterCounter - 1] is '\0') {

unmark UART0 for output;

k\_non\_block\_release\_memory\_block(globalCurrentEnvelope);

globalOutCharacterCounter ← 0;

}

}

}

atomic (off);

}

## 2.5.2.1 Hotkeys

void k\_print\_ready\_queue()

{

print("----- PROCESSES CURRENTLY IN READY QUEUE -----");

print("Current running process with PID " + globalCurrentProcess.pid);

for i ← 0 to 3 do {

if(readyPriorityQueue[i] is not empty) {

curPCBNode ← readyPriorityQueue[i].head;

print("Priority " + i + ": ");

while(curPCBNode is not NULL){

print("Process with PID " + curPCBNode.pcb.pid);

curPCBNode ← curPCBNode.next;

}

}

}

}

void k\_print\_blocked\_on\_memory\_queue()

{

print("----- PROCESSES CURRENTLY IN BLOCKED ON MEMORY QUEUE -----");

for i ← 0 to 3 do {

if(blockedOnMemoryQueue[i] is not empty) {

curPCDNode ← blockedOnMemoryQueue[i].head;

print("Priority " + i + ": ");

while(curPCDNode not NULL) {

print("Process with PID " + curPCDNode.pcb.pid);

curPCDNode ← curPCDNode.next;

}

}

}

}

void k\_print\_blocked\_on\_receive\_queue\_helper (int priority) {

curPCB ← blocked\_on\_receive\_list;

while (curPCB is not NULL)

{

if (curPCB.pcb.priority is priority){

break;

}

curPCB ← curPCB.next;

}

if (curPCB is NULL) return;

print ("Priority " + priority + ": ");

while(curPCB is not NULL) {

if(curPCB.pcb.priority is priority){

print("Process with PID " + curPCB.pcb.pid);

}

curPCB ← curPCB.next;

}

}

void k\_print\_blocked\_on\_receive\_queue()

{

print("----- PROCESSES CURRENTLY IN BLOCKED ON RECEIVE QUEUE -----");

for i ← 0 to 3 do {

k\_print\_blocked\_on\_receive\_queue\_helper(i);

}

}

# 2.6 SYSTEM PROCESSES

[Insert description]

2.6.1 Null Process

void null\_proc (void) {

while (true) {

k\_release\_processor ();

}

}

## 2.6.2 KCD Process

void kcd\_proc (void)

{

while (true) {

msgEnvelope ← receive\_message (sender);

if (msgEnvelope is not NULL) {

if (msgEnvelope.message\_type is MSG\_COMMAND\_REGISTRATION) {

for i ← 0 to KC\_MAX\_COMMANDS - 1 do {

if (globalKeyboardCommands[i] is available) {

globalKeyboardCommands[i].pid ← sender;

globalKeyboardCommands[i].command ← msgEnvelope.message;

break;

}

}

} else if (msgEnvelope.message\_type is MSG\_CONSOLE\_INPUT) {

command ← command portion from msgEnvelope.message

for j ← 0 to KC\_MAX\_COMMANDS – 1 do {

if (command equals globalKeyboardCommands[j].command) {

KCDMsgEnvelope ← request\_memory\_block();

KCDMsgEnvelope.message\_type ← MSG\_KCD\_DISPATCH;

KCDMsgEnvelope.message ← msgEnvelope.message;

send\_message(globalKeyboardCommands[j].pid, KCDMsgEnvelope);

break;

}

}

release\_memory\_block(msgEnvelope);

}

}

}

}

## 2.6.3 CRT Display Process

void crt\_proc (void)

{

while (true) {

envelope ← receive\_message (NULL);

pUart ← LPC\_UART0;

if (envelope.message\_type is MSG\_CRT\_DISPLAY) {

send\_message (UART\_IPROC\_PID, envelope);

mark pUart for output;

} else {

release\_memory\_block (envelope);

}

}

}

# 2.7 USER PROCESSES

## 2.7.1 24 Hour Wall Clock Display Process

void wall\_clock\_proc (void) {

register command "%WR" in globalKeyboardCommands;

register command "%WS" in globalKeyboardCommands;

register command "%WT" in globalKeyboardCommands;

while (true) {

receiveMsgEnvelope ← receive\_message (NULL);

if(receiveMsgEnvelope.message\_type is MSG\_WALL\_CLOCK and

receiveMsgEnvelope.sender\_pid is WALL\_CLOCK\_PID and showWallClock is true) {

send\_message to CRT to display current time;

delayed send message to self in 1000 ms;

} else {

if (charMessage == % WR) {

showWallClock ← true;

send message to self to reset current time;

}

if (charMessage == 'WS') {

showWallClock ← true;

send message to self to reset to time indicated in receiveMsgEnvelope.message;

} else if (charMessage[2] is 'T') {

showWallClock ← false;

}

}

release\_memory\_block(receiveMsgEnvelope);

}

}

## 2.7.2 Set Priority Command Process

void set\_priority\_proc(void) {

register command "%C" in globalKeyboardCommands;

while (true) {

recMsg ← receive\_message (NULL);

if (recMsg.message has the valid format) {

pid ← recMsg.message [3];

priority ← recMsg.message [5];

set\_process\_priority(pid, priority);

} else {

send invalid input error message to CRT for printing to console;

}

release\_memory\_block(recMsg);

}

}

## 2.7.3 Stress Tests

## 2.7.4 User Tests

3 Timing Analysis

4 Lessons Learned