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)

Table of Contents

[Table of Contents 2](#_Toc413695201)

[1 Introduction 4](#_Toc413695202)

[2 Software Design Description 5](#_Toc413695203)

[2.1 INITIALIZATION 5](#_Toc413695204)

[2.1.1 Memory Initialization 5](#_Toc413695205)

[2.1.2 Process Initialization 5](#_Toc413695206)

[2.2 MEMORY MANAGEMENT 5](#_Toc413695207)

[2.2.1 Requesting Memory 5](#_Toc413695208)

[2.2.2 Releasing Memory 6](#_Toc413695209)

[2.3 PROCESSOR MANAGEMENT 6](#_Toc413695210)

[2.3.1 Scheduling 6](#_Toc413695211)

[2.3.1 Context Switching 6](#_Toc413695212)

[2.3.2 Process Priority 6](#_Toc413695213)

[2.4 INTERPROCESS COMMUNICATION 7](#_Toc413695214)

[2.4.1 Sending Messages 7](#_Toc413695215)

[2.4.2 Receiving Messages 7](#_Toc413695216)

[2.4.3 Non-Blocking Message Receiving 8](#_Toc413695217)

[2.4.4 Delayed Messages 8](#_Toc413695218)

[2.5 INTERRUPT PROCESSES 8](#_Toc413695219)

[2.5.1 Timer I-Process 9](#_Toc413695220)

[2.5.2 UART I-Process 9](#_Toc413695221)

[2.5.2.1 Hotkeys 9](#_Toc413695222)

[2.6 SYSTEM PROCESSES 9](#_Toc413695223)

[2.6.1 Null Process 9](#_Toc413695224)

[2.6.2 KCD Process 9](#_Toc413695225)

[2.6.3 CRT Display Process 10](#_Toc413695226)

[2.7 USER PROCESSES 10](#_Toc413695227)

[2.7.1 24 Hour Wall Clock Display Process 11](#_Toc413695228)

[2.7.2 User Tests 11](#_Toc413695229)

[2.7.3 Stress Tests 11](#_Toc413695230)

[3 Timing Analysis 11](#_Toc413695231)

[4 Lessons Learned 12](#_Toc413695232)

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

[Insert description]

**void memory\_init(void);**

This method starts with placing 4 bytes of padding at the bottom of the RAM (at the lowest address). Then memory is allocated for PCB pointers and PCB node pointers. A global stack pointer is created in preparation for the call to alloc\_stack. Memory is allocated for the memory blocks map. The pointer to the start of the heap is also initialized. The memory map is zero-initialized to indicate that all of the memory blocks are free.

**U32 \*alloc\_stack(U32 size\_b);**

This method shifts the global stack pointer by the provided value (size\_b). The global stack pointer is 8-byte aligned.

## 2.1.2 Process Initialization

[Insert description]

**void process\_init(void);**

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. 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 queue and the blocked queue are both null-initialized. Then the processes' PCB nodes are enqueued in the ready priority queue according to their priorities.

# 2.2 MEMORY MANAGEMENT

## 2.2.1 Requesting Memory

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

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. The processor is released and moves onto the next process. If there are allocable memory blocks, we allocate them in the order that the processes were enqueued in the blocked queue.

## 2.2.2 Releasing Memory

**int k\_release\_memory\_block(void \*);**

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.

# 2.3 PROCESSOR MANAGEMENT

## 2.3.1 Scheduling

**PCB \*scheduler(void);**

This method 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.

## 2.3.1 Context Switching

**int k\_release\_processor(void);**

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. Then process\_switch is called on the old process. After the context switch, the new process' state is set to RUN.

## 2.3.2 Process Priority

**void set\_priority(int pid, int 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 in to the appropriate queue. The method returns RTX\_ERR if the process with the given PID is not found.

**int get\_priority(int pid);**

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.

# 2.4 INTERPROCESS COMMUNICATION

## 2.4.1 Sending Messages

[Insert description]

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

atomic (on);

newMsgEnvelope ← msgEnvelope;

get senderProcess using newMsgEnvelope.senderPID;

get targetProcess using targetPID;

enqueue newMsgEnvelope onto targetProcess.msgQueue;

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

remove targetProcess from blockedOnReceiveList;

targetProcess.state ← READY;

enqueue targetProcess onto readyPriorityQueue;

if ((preemption is on) and

(targetProcess.priority >

currentProcess.priority)){

k\_release\_processor ();

}

}

atomic (off);

return RTX\_OK;

}

## 2.4.2 Receiving Messages

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);

newMsgEnvelope ← msgEnvelope;

newMsgEnvelope.delay ← globalTimerCount + delay;

preemption (off);

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

preemption (on);

atomic (off);

return RTX\_OK;

}

# 2.5 INTERRUPT PROCESSES

[Insert description]

## 2.5.1 Timer I-Process

void timer\_i\_proc (void) {

preemption\_flag ← false;

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);

}

preemption (off);

while ((timeoutQueue.head not NULL) and

(timeoutQueue.head.delay <= globalTimerCount) {

cur ← dequeue timeoutQueue;

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

if (globalPCBs[cur.destination\_pid].m\_priority > globalCurrentProcess.m\_priority) {

preemption\_flag ← true;

}

}

preemption (on);

globalTimerCount++;

atomic (off);

if (preemption\_flag) {

k\_release\_processor ();

}

}

## 2.5.2 UART I-Process

void uart\_i\_proc (void) {

pUart ← (LPC\_UART\_TypeDef \*) LPC\_UART0;

atomic (on);

uart\_preemption (off);

IIRInterruptID ← (pUart.IIR) >> 1 ;

if (IIRInterruptID is in read status) {

characterRead ← pUart.RBR;

if (there is memory available) {

if (characterRead is '\r') {

display\_msg ← '\n' + characterRead + '\0';

} else {

display\_msg ← characterRead + '\0';

}

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

envelope.sender\_pid ← UART\_IPROC\_PID;

envelope.destination\_pid ← CRT\_PID;

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

envelope.message ← display\_msg;

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

uart\_preemption (on);

}

if (characterRead not '\r') {

globalInputBuffer[globalInputBufferIndex] ← characterRead;

globalInputBufferIndex ++;

} else {

globalInputBuffer[globalInputBufferIndex] ← '\0';

globalInputBufferIndex ++;

if (there is memory available) {

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

envelope.sender\_pid ← UART\_IPROC\_PID;

envelope.destination\_pid ← KCD\_PID;

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

envelope.message ← globalInputBuffer;

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

globalInputBufferIndex ← 0;

uart\_ preemption (on);

}

}

globalInputBuffer[globalInputBufferIndex] ← characterRead;

} else if (IIRInterruptID is in write status) {

if (globalCurrentEnvelope is NULL) {

globalCurrentEnvelope ←

(ENVELOPE\*) k\_non\_block\_receive\_message(UART\_IPROC\_PID);

}

if (globalCurrentEnvelope not NULL) {

globalCurrentEnvelopeMessage ← (char\*) globalCurrentEnvelope.message;

if (globalCurrentEnvelopeMessage [globalOutCharacterCounter] is not '\0') {

pUart.THR ← globalCurrentEnvelopeMessage [globalOutCharacterCounter];

globalOutCharacterCounter ++;

} else {

if (globalPCBNodes[UART\_IPROC\_PID].p\_pcb.msgQueue.head is NULL) {

unmark pUart for output;

}

pUart.THR ← globalCurrentEnvelopeMessage [globalOutCharacterCounter];

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

globalCurrentEnvelope ← NULL;

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.m\_pid);

for i ← 0 to 3 do {

if(readyPriorityQueue[i] is not empty) {

curPCBNode ← ready\_priority\_queue[i].head;

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

while(curPCBNode not NULL){

print("Process with PID " + curPCBNode.p\_pcb.m\_pid);

curPCBNode ← curPCBNode.next;

}

}

}

}

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

{

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

for i ← 0 to 3 {

if(blockedOnMemoryQueue[i] is not empty) {

curPCDNode ← blockedOnMemoryQueue[i].head;

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

while(curPCDNode not NULL) {

print("Process with PID " + curPCDNode.p\_pcb.m\_pid);

curPCDNode ← curPCDNode.next;

}

}

}

}

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

curPCB ← blocked\_on\_receive\_list;

while (curPCB not NULL)

{

if (curPCB.p\_pcb.m\_priority is priority){

break;

}

curPCB ← curPCB.next;

}

if (curPCB is NULL) return;

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

while(curPCB not NULL) {

if(curPCB.p\_pcb.m\_priority is priority){

print("Process with PID " + curPCB.p\_pcb.m\_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)

{

declare sender;

while (true) {

msgEnvelope ← receive\_message (sender);

if (msgEnvelope 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) {

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

KCDMsgEnvelope ← request\_memory\_block();

KCDMsgEnvelope.sender\_pid ← KCD\_PID;

KCDMsgEnvelope.destination\_pid ← globalKeyboardCommands[j].pid;

KCDMsgEnvelope.nextMsg ← NULL;

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

KCDMsgEnvelope.delay ← 0;

KCDMsgEnvelope.message ← msgEnvelope.message;

send\_message(KCD\_PID, KCDMsgEnvelope);

break;

}

}

release\_memory\_block(msgEnvelope);

}

}

}

}

## 2.6.3 CRT Display Process

void crt\_proc (void)

{

while (true) {

envelope ← (ENVELOPE\*) receive\_message (NULL);

pUart ← (LPC\_UART\_TypeDef \*) 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 ← (ENVELOPE\*) receive\_message (NULL);

charMessage ← (char \*) receiveMsgEnvelope.message;

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

(receiveMsgEnvelope.sender\_pid is WALL\_CLOCK\_PID) and (show\_wclock is true)) {

delayedMsgEnvelope ← (ENVELOPE\*) request\_memory\_block ();

w\_clock ← (ENVELOPE\*) request\_memory\_block ();

w\_clock.sender\_pid ← WALL\_CLOCK\_PID;

w\_clock.destination\_pid ← CRT\_PID;

w\_clock.message\_type ← MSG\_CRT\_DISPLAY;

curr\_time ← g\_timer\_count - elapsed + base;

w\_clock.message ← curr\_time.hh + ":" + curr\_time.mm + ":" + curr\_time.ss;

send\_message(CRT\_PID, w\_clock);

delayedMsgEnvelope.message\_type ← MSG\_WALL\_CLOCK;

delayedMsgEnvelope.sender\_pid ← WALL\_CLOCK\_PID;

delayedMsgEnvelope.destination\_pid ← WALL\_CLOCK\_PID;

delayedMsgEnvelope.message ← NULL;

delayed\_send(WALL\_CLOCK\_PID, delayedMsgEnvelope, 1000);

} else {

if (charMessage[2] is 'R') {

msgEnvelope ← (ENVELOPE \*) request\_memory\_block ();

msgEnvelope.message\_type ← MSG\_WALL\_CLOCK;

msgEnvelope.sender\_pid ← WALL\_CLOCK\_PID;

msgEnvelope.destination\_pid ← WALL\_CLOCK\_PID;

msgEnvelope.message ← NULL;

reset current time;

if (show\_wclock is false) {

show\_wclock ← true;

send\_message(WALL\_CLOCK\_PID, msgEnvelope);

}

}

if (charMessage[2] is 'S') {

msgEnvelope ← (ENVELOPE \*) request\_memory\_block ();

msgEnvelope.message\_type ← MSG\_WALL\_CLOCK;

msgEnvelope.sender\_pid ← WALL\_CLOCK\_PID;

msgEnvelope.destination\_pid ← WALL\_CLOCK\_PID;

msgEnvelope.message ← NULL;

set current time to time indicated in receiveMsgEnvelope.message;

if (show\_wclock is false) {

show\_wclock ← true;

send\_message(WALL\_CLOCK\_PID, msgEnvelope);

}

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

show\_wclock ← false;

}

}

release\_memory\_block(receiveMsgEnvelope);

}

}

## 2.7.2 User Tests

## 2.7.3 Stress Tests

3 Timing Analysis

4 Lessons Learned