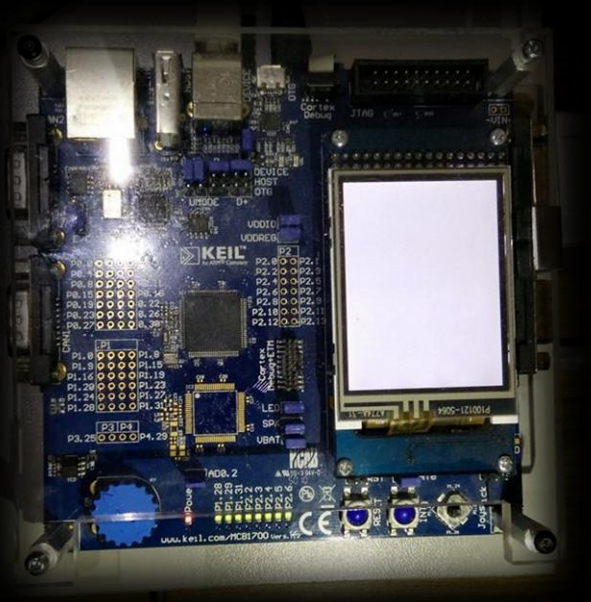

SE350 RTX Project Documentation

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

University of Waterloo

3A Software Engineering

2/3/2015

Data Structures, Designs, Algorithms

USER PROCESSES

`void k_rtx_init(void);`

This method initializes UART0, memory management and process management. Then it calls `k_release_processor` to start the processes.

`int release_processor(void);`

This method calls the corresponding method that releases the processor in the kernel.

`void request_memory_block(void);`

This method calls the corresponding method that requests memory blocks in the kernel.

`int release_memory_block(void *);`

This method calls the corresponding method that releases memory blocks in the kernel.

`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.

MEMORY MANAGEMENT

`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.

`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.

`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.

PROCESS MANAGEMENT

`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.

`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.

`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.

`void k_block_current_processs(void);`

This method places the PCB node corresponding to the current process into the blocked queue if the current process is not null. Before enqueuing, the process' state is changed to BLOCKED.

`void k_ready_first_blocked(void);`

This method checks whether the blocked queue is empty and if it isn't, the PCB node of the first process in the queue is dequeued and the process' state is changed to RDY. This PCB node is then added to the ready priority queue according to its priority.