001

Multiprogramming

Operating Systems

Multiprogramming

Building a Bare-Metal ARM Operating System

General Description

This project aims to implement a minimal multiprogramming system on a BeagleBone Black. The system will manage three distinct programs resident in memory: an OS block responsible for initialization, interrupt handling, and process scheduling, and two user processes—one printing lowercase letters from 'a' to 'z' in a loop, and another printing digits from 0 to 9 in a loop. Due to the absence of a memory management unit (MMU) in this bare-metal setup, all programs will be loaded at fixed memory addresses determined at link time. The OS will use the AM335x's DMTimer2 to generate periodic interrupts, triggering a Round-Robin scheduler that context-switches between the two user processes. Each process will execute for a fixed time slice (e.g., 1s, as per prior timer configurations), producing interleaved output that alternates between letters and numbers, demonstrating cooperative multitasking.

The architecture builds on a layered approach, inspired by the provided example code:

- **Low-Level Code**: os.c and root.s handle hardware initialization (UART, timer, interrupts), context switching, and basic I/O. (Later we need to implement the syscalls)
- **Language Library**: stdio.c and string.c provide abstractions like PRINT and string manipulation.
- **User Programs**: Two processes (process1.c, process2.c) implement the letter and number printing logic.
- **Linker Script**: linker.ld assigns fixed memory regions to prevent overlap, with separate stacks for each process.

The expected output will show interleaved execution, e.g., numbers (0-4), then letters (a-e), then numbers (5-9), and so on, driven by timer interrupts.

Project Objective

The primary objective is to design and implement a bare-metal multiprogramming system on the BeagleBone Black that:

1. Loads three programs (OS and two user processes) into fixed memory locations.
2. Initializes the AM335x hardware, including UART for I/O, DMTimer2 for interrupts, and the interrupt controller (INTC).
3. Implements a Round-Robin scheduler using timer interrupts to switch between two user processes.
4. Ensures each process has its own stack to avoid overlap and maintains process state (registers) during context switches.
5. Produces interleaved output from the two user processes, demonstrating successful multitasking.

The system should be robust, with clear separation of concerns between OS and user code, and maintain predictable timing (e.g., 1 s per time slice).

Assignment Tasks

1. **Hardware Initialization (OS Block)**:
    - Modify root.s to set up the initial stack for the OS and jump to a C-based main function.
    - In os.c, initialize:
        - UART0 (0x44E09000, as per AM335x) for serial I/O
        - DMTimer2 (0x48040000) for periodic interrupts at 1 s.
        - INTC (0x48200000) to enable IRQ 68 for DMTimer2.
    - Set up the interrupt vector table and handler for timer interrupts.

2. **User Process Implementation**:
    - Create process1.c to print lowercase letters (a to z) in a loop, using PRINT from stdio.c.
    - Create process2.c to print digits (0 to 9) in a loop, using PRINT.

- Each process should run indefinitely, yielding control via timer interrupts.

3. **Memory Layout and Linking**:
   - Update linker.ld to allocate fixed memory regions:
     - OS code/data: Starting at 0x80000000 (typical for AM335x RAM).
     - Process 1 code/data: 0x80010000.
     - Process 2 code/data: 0x80020000.
     - OS stack: 0x80008000.
     - Process 1 stack: 0x80018000.
     - Process 2 stack: 0x80028000.
   - Ensure .bss is cleared for all programs to avoid uninitialized variables.
   - Note: This memory layout is a suggestion, you need to give your own design.

4. **Context Switching**:
   - Implement a context switch in the timer interrupt handler (os.c):
     - Save the current process's registers (R0-R12, LR, PC, CPSR) to its stack.
     - Store the stack pointer in a process control block (PCB).
     - Switch to the next process by restoring its registers and stack pointer.
     - Use a simple PCB array to track two processes (state, stack pointer).
   - Alternate between processes in Round-Robin fashion every 1 s.

5. **Scheduler**:
   - In the OS main, initialize two PCBs with entry points to process1_main and process2_main.
   - Start the first process after setup.
   - On each timer interrupt, invoke the context switch to toggle processes.

6. **Build and Test**:
    - Update build_and_run.sh to compile process1.c, process2.c, and link all objects.
    - Test on BeagleBone Black via a bootloader (e.g., U-Boot).
    - Verify output shows interleaved letters and numbers, e.g.:
    - 0
    - 1
    - 2
    - 3
    - 4
    - a
    - b
    - c
    - d
    - e
    - 5
    - 6
    - ...

Tips

- **Timer Setup**:
    - Reuse your proven DMTimer2 code. Ensure TCLR enables auto-reload (0x3) and TIER enables overflow interrupts (0x2).
    - Verify IRQ 68 is unmasked (INTC_MIR_CLEAR2, 1 << 4).

- **Context Switching**:
    - Save/restore registers in assembly within the interrupt handler for speed.

- Example:

```
void timer_irq_handler(void) {
    asm volatile (
        "stmfd sp!, {r0-r12, lr}" // Save registers
        "mrs r0, spsr"
        "stmfd sp!, {r0}" // Save CPSR
        // Save SP to PCB, switch process
    );
    PUT32(TISR, 0x2);
    PUT32(INTC_CONTROL, 0x1);
    // Restore next process's registers
}
```

- Store PC indirectly via LR to resume correctly.

- **Memory Layout**:
    - Check AM335x memory map: DDR3 RAM starts at 0x80000000. Avoid reserved regions (e.g., 0x40000000-0x5FFFFFFF for peripherals).
    - Allocate ~64KB per process to be safe (code + stack).

- **Stack Separation**:
    - Initialize each process's stack pointer in its PCB during OS setup.
    - Test for overlap by filling stacks with known patterns (e.g., 0xDEADBEEF) and checking for corruption.

- **Debugging**:
    - Add debug prints in the handler to log context switches (e.g., PRINT("Switch to process %d\n", current_pid)).
    - On BeagleBone, use a serial console to capture output.

- **Performance**:
  - Minimize interrupt handler overhead—avoid heavy PRINT calls inside it.

- **Safety**:
  - Ensure .bss is zeroed in root.s before calling main:

```
ldr r0, =__bss_start
ldr r1, =__bss_end
mov r2, #0
bss_clear:
    cmp r0, r1
    strlt r2, [r0], #4
    blt bss_clear
```

  - Disable interrupts during context switch to prevent reentrancy (cpsid i).

- **Testing**:
  - Start with one process to verify timer interrupts and UART.
  - Add the second process only after the first runs stably.
  - Check for missed interrupts by logging tick counts.

- **Extra Points**:
  - Add a process state (RUNNING, READY) in the PCB for future scalability.
  - Implement a yield function for voluntary context switches if processes finish a cycle early.