Homework #3B
Alec Newport (acn55) and Eldor Bekpulatov (eb654)

1. An application in a real-time environment must be completed correctly and it must be within its timing requirement.

2. One innovation that may impact application execution in a real-time environment would be interrupts. Based on the typical definition, an event triggering an interrupt is handled as soon as it is detected, preempting the currently running process. This can't work in a real-time system, because processes are carefully scheduled to meet their timing requirements. Any addition of unexpected code, such as an interrupt handler, could cause a number of processes to miss their deadlines. To integrate interrupts into a real-time system, they must be treated as any other process (i.e. they cannot automatically preempt the running process), albeit perhaps with a shorter deadline to induce a more rapid response.
Another innovation affecting real-time guarantees is memory management, and swapping in particular. In a typical embedded system, the operating system can swap memory blocks between upper level caches and main memory. If this were to occur in a real-time system, not only would precious time need to be spent on the swap action itself, potentially causing timing issues with other processes, if a certain process is nearing its deadline but its data has been swapped to main memory, it can take a very long time to bring it back into an upper-level cache for access, likely causing the process to fail its timing requirement terribly. To adapt this for a real-time system, the memory management unit would need to take deadlines into account when deciding which blocks to swap out of the caches. The closer a process' deadline, the less likely it should be that that section of memory is chosen to be evicted.

3. Figure 1 below shows an example process with its arrival time t_a, which is the time the process becomes available to be scheduled; start time t_s, which is the first time the processor begins executing the process; finish time t_f which is the time the processor finishes executing the process; and deadline time t_d, which is the time by which the real-time system guarantees the process will be finished. The execution time is the difference between the start time and the finish time, representing the time it takes to actually execute the process code. This data can be retrieved from the perf stat command. The process latency is the difference between the arrival time and the start time, representing the time it takes for the processor to start executing the process code. This can be measured using the perf sched latency command.

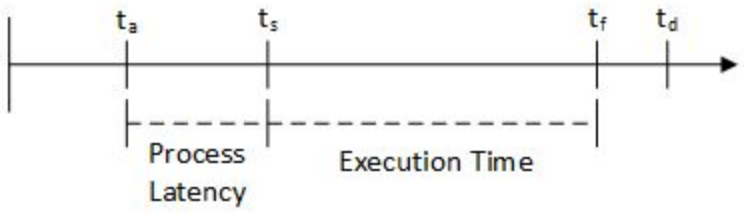

Figure 1: Single process schedule

4. One Linux kernel function we discussed was memory management. This means that the kernel is responsible for keeping track of the memory used by all running processes. There are eight L1 caches: one instruction cache and one data cache for each processor core. There is a single L2 cache which is used for resource sharing between cores. Finally, there is the main memory in the 1 GB RAM chip used for longer-term storage (Figure 2). The kernel is responsible for allocating space for all processes (both user and kernel), implementing virtual memory, and swapping blocks between caches and main memory.

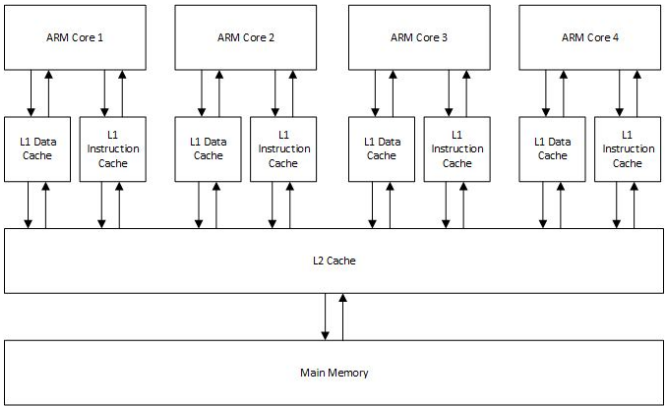

Figure 2: Memory structure

Another function discussed was resource management. This means the kernel decides how to allocate processes between the processor cores, when the processes can use the memory ports, etc. Essentially any time there's some limited hardware on the board that only a certain number of processes can access at once, the kernel decides how to allocate the available resources through various scheduling algorithms.

5. The first stage of the boot sequence runs from ROM within the SoC so the processors always know exactly where to look. This activates the memory and caches and moves the second stage of the boot sequence, bootcode.bin, from the SD card to the L2 cache where it is accessible. This code is run by the GPU (not CPU cores yet). The second stage of the boot sequence, bootcode.bin, runs from the cache. It initializes the SDRAM and moves the third stage, loader.bin, into RAM. This third stage runs from RAM. This is responsible for loading and running the GPU firmware stored in start.elf. The final setup stage is start.elf, which runs on the GPU as well. It is responsible for loading the kernel image file, kernel.img, into memory, loading the configuration state, and starting up the CPU. Finally, kernel.img starts running on the CPU and control is turned over to the user.

ELF stands for Executable and Linkable Format. It is an executable format that supports a broad variety of hardware configurations and ISAs.