**Exercise-1**

**Real Time Embedded Systems**

Submission by Kevin Tom, Dhiraj Bennadi

1. Total CPU Utilization

Total CPU utilization is given by the equation,

Where, Ci  - completion time (how much time is required for the completion of task)

Ti  - period (how often the task will come).

m – number of services

The CPU utilization for the services is,

= 0.93

The CPU utilization by 3 services is 93%.

Chart, bar chart

Description automatically generated

Chart, waterfall chart

Description automatically generated

Feasibility and safety issues:

The RM LUB is a pessimistic feasibility that will fail some proposed service sets that actually work but will never pass a set that doesn’t work. It a sufficient feasibility test but not necessary.

2. Apollo 11 system experienced a cpu overload while descending on the moon. It caused the decent services to miss deadlines, while using the radar system for descending the system give a 1202 alarm. This alarm indicated that the computer is behind work and is overloaded. The processor was already overloaded, the alarm processing for showing 1202 added to the workload.

Computer design:

* The computer had 36,864 15-bit word of fixed memory and 2048 words of RAM.
* Most of the executables were in fixed memory, erasable memory weas used for variable data, counters, etc.
* Since less RAM is available, the same memory had to be used for different purposes at different times.

Program execution:

* Each job was allocated 12 RAM space, if the job requires more space it asked for a VAC (which had 44 erasable words). There were 7 core sets and 5 VACs.
* Once a job is scheduled and the job requires a VAC area, the OS will scan through all the VAC areas, find one, then start scanning to find available core set.
* If no VAC space is there, it will return NOVAC which skips the scanning and sets up 1201 alarm.
* If no core sets are available, the program would branch and set alarm 1202.

What really happened?

* Repeated jobs to process rendezvous radar data being scheduled to the software because of misconfiguration of radar switches.
* Core set got filled up triggering 1202 alarm.
* Scheduling request that came after the 1202 was the one that caused the overflow and was the one requested for the VAC area.
* Each time a 1201/1202 alarm appeared, the computer rebooted and restarted important tasks. This did not restart any of the radar tasks that was erroneous.

The roots cause analysis

* The main cause of the problem was that the CPU was utilized more than 70%, which leaves a very little system space. The system should have been designed so that it will not utilize more than 70% as described by Liu and Layland.
* If it was to be scheduled for more than 70% the dynamic scheduling algorithm should have been implemented or a combination of both.
* Many tasks were non-periodic and was taking CPU time, priority was given to non-important tasks.
* The radar data was less important than other procedures and was running, this made it run out of core set and eventually VAC area.
* Even in the rebooting/restarting sequence of computer the radar task which was erroneous was also continued.

3 Key Assumptions

1. All services requested on periodic basis and period is constant.
2. Completion time is less than period.
3. Services requested are independent (does not depend on any other tasks)

Assumptions not understood:

(A5) Any nonperiodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

(A4) Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

Would RM analysis would have prevented Apollo 11 potential mission abort?

Yes, RM analysis would have prevented Apollo 11 potential abort. Some of the major reasons are,

* RM LUB policy suggests utilization of 70% of the CPU. In the case of Apollo 11 the computer was utilizing full CPU.
* A combination of both fixed and dynamic deadline driven scheduling algorithm should have provided the most benefits.
* The radar task was processing and was running at higher priority than other landing tasks, landing procedures or other important tasks should have given priority. This is where combination of fixed and dynamic deadline scheduling algorithm’s importance comes into picture.

Problem 3:

The code creates a thread called main\_thread which executes in the main loop. The main thread is assigned thread attributes such as PTHREAD\_INHERIT\_SCHED which defines the scheduler inheritance and SCHED\_FIFO as the scheduling algorithm.

The thread is also assigned the max priority for scheduling.

The main thread is created, and all the above-mentioned parameters are assigned to the main thread. The main thread is assigned a function delay\_test.

The resolution of the clock is checked in the delay\_test. The execution times are retrieved used the clock\_gettime function before the start and after the execution of the delay loop. The number of iterations of the delay loop is specified by a macro TEST\_ITERATIONS.

In the delay loop the thread is made to sleep by a high-resolution sleep function called nanosleep.

nanosleep() suspends the execution of the calling thread until either at least the time specified has elapsed or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process.

The function clock\_gettime retrieve the time specified by the clock (clk\_id)

The clk\_id argument is the identifier of the clock on which to act.

In the code 5 types of clocks have been used to measure the runtime of the code.

The following code was executed on Jetson Nano SBC.

|  |  |  |  |
| --- | --- | --- | --- |
| Clock Selected | Execution Time measured using the command time | Execution time measured using syslog | Number of Iterations |
| Clock Monotonic | 0m1.042s | 1s | 100 |
| Clock Monotonic | 0m10.169s | 10s | 1000 |
| Clock Monotonic | 0m50.745s | 51s | 5000 |
| Clock Monotonic Raw | 0m1.046s | 1s | 100 |
| Clock Monotonic Raw | 0m10.165s | 10s | 1000 |
| Clock Monotonic Raw | 0m50.771s | 51s | 5000 |
| Clock Monotonic Coarse | 0m1.046s | 1s | 100 |
| Clock Monotonic Coarse | 0m10.171s | 10s | 1000 |
| Clock Monotonic Coarse | 0m50.667s | 51s | 5000 |
| Clock Real Time | 0m1.045s | 1s | 100 |
| Clock Real Time | 0m10.160s | 10s | 1000 |
| Clock Real Time | 0m50.739s | 51s | 5000 |
| Clock Real Time Coarse | 0m1.049s | 1s | 100 |
| Clock Real Time Coarse |  |  | 1000 |
| Clock Real Time Coarse | 0m50.709s | 50s | 5000 |

The screenshots are available in the folder Screenshots

1. Low Interrupt Handler: Interrupt Latency for RTOS is the duration of time from the assertion of a device interrupt to the completion of the device’s requested operation.
2. Low Context Switch Time: A context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point
3. Stable Timer Services: The timers implemented in RTOS have high precision with availability to select from multiple clocks. The routines are implemented using these high precision clocks which makes them better suited for hard deadline driven systems.

All the above-mentioned features are a characteristic of RTOS, given the nature in which they deal with deadlines.

The response time is being limited by the sum of the IO Latency, context switch latency and execution time and potential interference.

Text

Description automatically generated

Argument:

Among all the clocks tested, the clock monotonic seemed to have better accuracy when compared syslog timers.

Problem 4:

1. Description of the codes
2. simplethread   
   Code Link:

The code used posix threads libraries to create multiple threads and to run tasks.

The number of threads is defined by the macro NUM\_THREADS. A structure is used to store the thread ID.

The main function creates the number of threads as specified by NUM\_THREADS and spawns them.

The threads are assigned to execute the function counterThread which calculates the sum as per the macro value and prints the thread index.

Code Result:

A picture containing graphical user interface

Description automatically generated

1. rt\_simplethread  
   Code Link:

This code builds on the previous simplethread, extending the functionality to changing the scheduling policy along with execution of the thread function for a larger amount of time.

The main function changes the default scheduling from SCHED\_OTHER to SCHED\_FIFO

The number of threads created by the is defined by the macro NUM\_THREADS.

Each of the thread is created with specific attributes. The thread attributes are created with the parameters PTHREAD\_EXPLICIT\_SCHED and the scheduling policy SCHED\_FIFO.

The threads are also assigned priority obtained from the return value of the function sched\_get\_priority\_max()  
Each of the threads are assigned to execute the function counterThread.

The thread is intended to carry out a sum task and calculate the Fibonacci Series through the parameters seqIterations and reqIterations

The timestamps are recorded before and after the execution of the tasks (sum and Fibonacci Series) using the function clock\_gettime

Another utility function to find the difference between the start and stop timestamps is invoked and the difference is printed.

Additional details such as the thread indices are also printed.

Code Result

Text

Description automatically generated

1. rt\_thread\_improved

Code Link:

This code builds on the code of rt\_simplethread, extending the functionality by setting specific cores of the CPU to execute the specific threads.

The code maintains the same functionality of changing the scheduling policy to SCHED\_FIFO and creation of threads with thread attributes PTHREAD\_EXPLICIT\_SCHED and the scheduling policy SCHED\_FIFO.

The threads are also assigned priority obtained from the return value of the function sched\_get\_priority\_max()

The specific core of the CPU is assigned using the function CPU\_SET

The threads are assigned to execute the function workerThread.

The function workerThread performs the same set of as explained earlier (sum and generation of Fibonacci series) using the parameters seqIterations and reqIterations  
Code Result

Text

Description automatically generated

Since all the above codes try to change the default scheduling policy, the code should be executed with the root privileges. As SCHED\_FIFO aggressively takes over the CPU resources.

Incdecthread

Text

Description automatically generated

Text

Description automatically generated with medium confidence

|  |  |
| --- | --- |
| Threading | Tasking |
| A thread is simply the trace of a CPU’s over time not including context switch code execution by an RTOS. State Information may or may not be associated with a thread of execution but the value of the PC before a context switch is the minimum state that must be maintained on a system that includes preemption | A thread with normal thread state, including stack, registers, PC but also including signal handlers, task variables, task ID and name, priority, entry point, and a number of state and inter-task communication data contained in a TCB |

Semaphores:

An RTOS mechanism that can be used for synchronization of otherwise asynchronous tasks in order to coordinate resource usage, such as shared memory, or to simply indicate a condition, such as data is available on an interface.

Semaphore give: A semaphore operation that allows a thread to check if a resource is available; if another thread is blocking on this resource, then this will unblock that thread.

The Linux Operating System terminology is sem\_post.

Semaphore take: A semaphore operation that allows a thread to check if a resource is available; if not, the RTOS can either block the calling thread until it is, or simply return an error code.

The Linux Operating System terminology is sem\_wait.

Synthetic Workload Generation: A SW is a set of parameterized synthetic or artificial programs which serve as the workload for a system under study. The parameterized nature of the programs allows the user to change their behavior to create different resource demands on the system. The SW is easy to use, flexible, and can be representative of a real-time workload.

Synthetic Workload Analysis:

The system provided consists of 3 tasks for which 3 threads are created.

Thread 0 is assigned to execute the function Sequencer

Thread 1 is assigned to execute a workload of 10ms. Thread 2 is assigned to execute a workload of 20ms. The synchronization between the threads is maintained by semaphores semF10 and semF20.

The sequencer thread is assigned the highest priority; hence this task executes first. In the sequencer thread, the thread is made to sleep using the functions usleep. Once the sequencer thread is in waiting state, the task with the next highest priority takes over the CPU and starts executing. This thread holds the CPU until either it completes execution, or a higher priority task preempts it. Currently the thread with a higher priority than Thread 1 is the sequencer which is in waiting state. Hence no preemption takes place. Once the Thread 1 completes execution Thread 2 takes over the CPU and starts its execution. This thread will be interrupted when either Thread 0 or Thread 1 have any execution.

Code Sources:

All the code has been sourced from the repository:

http://mercury.pr.erau.edu/~siewerts/cec450/code/  
Courtesy: Dr Sam Siewert

Professor Tim Scherr