Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

A real-time Implementation of Gang Scheduling on embedded platforms

Supervisor

Prof. Luigi Palopoli

Student

Valeria Grotto

Academic year 2021/2022

Acknowledgments

Firstly, I would like to thank my thesis supervisor Prof. Luigi Palopoli from the Department of Computer Science at The University of Trento. He helped me to set up and correct this thesis, clearing all the doubts that emerged along the way. Moreover, I would like to thank Prof. Alberto Montresor for his availability during the entirety of the internship, the interest shown and the useful suggestions on the thesis format.

Secondly, I would like to thank Paolo Gai and all the staff at Huawei's Research Center in Pisa; I am grateful for the opportunity given me and the time I spent there. A special tanks goes to all the members of the POSIX Team: Alessandro Biasci, Bruno Morelli, Francesco Bagagli, Valerio Di Gregorio and Andrea Serafini. Your expertise and guidance were invaluable to successfully complete this research work. Furthermore, I am grateful for your time and patience in answering my questions and providing feedback on my work.

Finally, I would like to show my gratefulness to my friends and family for their continued support during the years spent at University.

Contents

Abstract

This thesis describes the implementation and testing of a process scheduler that enforces a gang scheduling policy for *PSE53*, a novel real-time operative system developed by Huawei. This work was carried on during an internship at the Huawei's Research Center in Pisa, Evidence S.r.l. The internship lasted 4 months, from February to June 2022.

Gang scheduling is a scheduling algorithm that schedules a set of threads to run in parallel on different cores. Usually, a real-time operating system implements single-core scheduling policies to avoid task interference and deadline misses. However, the gang policy allows the user to benefit from the multicore infrastructure while still avoiding interference. In fact, one of the main advantages of gang scheduling is the possibility to form a gang with only tasks that cooperate and do not hinder the others.

Moreover, the gang algorithm allows synchronizing tasks with *busy-waiting*, reducing the delay provoked by context switching.

Keywords: gang scheduling, real-time, parallel, scheduling, interference, busy-waiting

1 Introduction

The increasing demand for high-performance applications, such as autonomous driving and computer vision, has drawn attention to new solutions in the field of *parallel processing*, i.e. the simultaneous use of multiple cores. To handle processes and tasks, and execute them on different cores at the same time, it was necessary to develop different *scheduling algorithms* and one of them is *gang scheduling*, an algorithm that groups tasks and executes them simultaneously.

Despite the rising popularity, parallel processing is typically not used in safety-critical systems, like the ones in the automotive and the aviation industry. These type of systems usually support *real-time* applications that have to respect specific time constraints, i.e. *deadlines*, therefore having multiple tasks executing simultaneously can lead to interference between them and deadline misses.

In fact, the main challenge in the scheduling of parallel real-time tasks is how to allocate shared hardware resources, such as cache and memory controllers, among competing tasks. The inaccurate allocation and management of these resources can lead to an unpredictable *Worst Case Execution Time (WCET)*, a time measure used to estimate the maximum execution time of a task . In safety-critical systems, where having a predictable WCET is a necessity, parallelism is generally avoided and sometimes cores are even disabled to obtain a single-core architecture [2].

However, nowadays, it has become important to fully take advantage of the multiple cores even on real-time systems, to efficiently process computationally demanding real-time workloads (e.g., AI and vision). This is possible with a *gang scheduling* algorithm. In fact, the gang policy can be adapted for a real-time operating system so that the tasks in a gang do not interfere with each other and they can be executed in parallel without deadline misses.

1.1 Gang Scheduling

Gang scheduling is an algorithm that schedules multiple tasks to run simultaneously on different cores; the tasks that form a *gang* are interconnected and can cooperate in a constructive manner by sharing common resources without interfering with each other.

Today, the Gang model is already supported by many parallel computing standards, such as MPI, OpenMP, Open ACC or GPU computing. Moreover, the use of this algorithm also enables tight and accurate WCET in a real-time operating system, and it allows the implementation of well-known single-core scheduling strategies, that can be adapted to work with multiple gangs.

In this thesis, I present an implementation of a Gang Scheduling algorithm on a novel real-time operating system, developed for the automotive industry.

1.2 Gang Synchronization

Another advantage of gang scheduling, besides its predictability, is the ability to use *busy-waiting* to synchronize the tasks of a gang [8]. Busy-waiting, busy-looping, or spinning, is a technique in which a process repeatedly checks whether a condition is met, such as whether a keyboard input or a lock is available [5]. This technique can lead to task starvation if the tasks are not executed in a specific order. Yet, with gang scheduling, the tasks that use busy-waiting are grouped into a gang and are always executed simultaneously, so there is no downside of using it to synchronize the gang.

In addition, spinning, using the gang policy to schedule the tasks, is a better way to synchronize them compared to *blocking*, in fact, it saves time by not *context switching*. Blocking is a

synchronization method that uses a *lock* to protect a portion of code of a thread. For example, a thread can be blocked waiting for a resource to become available. Blocking generally happens when a thread tries to acquire a lock that is already held by another thread, in this case, the first one becomes blocked, there is a context switch, and it does not execute until the second releases the lock.

2 Background: Operating Systems

An *operating system (OS)* is a software that manages the computer hardware and software resources, and provides common services for computer programs [1] [12]. Some examples of well known OS are Linux, Unix, macOS and Microsoft Windows.

The central core of an operating system is the *Kernel* program. This software has complete control over everything that occurs in the system [10]. The kernel is the first part of the operating system to load in the memory during booting (i.e. system startup), and it remains there for the entire duration of the computer session, because its services are required continuously. It is important for it to be as small as possible, while still providing all the essential services needed by the other parts of the operating system and by the various application programs, such as memory management, process management, file management and I/O (input/output) management. These services are usually requested by application programs through a specified set of program interfaces, referred to as *system calls*.

System memory can be divided into two distinct regions: *kernel space* and *user space*. Kernel space is where the kernel executes and provides its services, whereas user space is a set of memory locations, in which user processes (such as generic software applications) run. User processes can access the Kernel space only through the use of system calls.

The kernel also has the important task of managing processes. It has to ensure that each process obtains its turn to run on the processor and that the individual processes do not interfere with each other by writing in the same areas of the memory. A *process* can be defined as an executing (i.e. running) instance of a program; it is represented in the operating system by a data structure called *Process Control Block* (PCB) or Process Descriptor. This structure contains all the information of the process, its identification number, its status (i.e. READY, RUNNING, SUSPENDED or BLOCKED) and other information about the current state of the process [17].

The different components of a kernel vary considerably according to the operating system, but they typically include: a *scheduler*, which determines how the various processes share the kernel's processing time; an *interrupt handler*, which manages all requests from the various hardware devices that compete for the kernel's services and a *memory manager*, which allocates the system's address spaces (i.e. locations in memory) among all users of the kernel's services.

2.0.1 Real-time Operating Systems

A particular type of operating system is a *real-time operating system*, RTOS. This kind of system allows the implementation of applications that have critically defined time constraints [19]. Real-time programs must guarantee response within specified time constraints, often referred to as *deadlines* [18]. In this context, a delay in the response may bring the system in an undesirable state. Moreover, in *hard* real-time systems, if the deadline is not met, could result in critical errors and catastrophic failures that, in some cases, may bring people's lives at risk. These applications are usually employed in the automotive and aviation industries, both of which demand immediate and accurate mechanical response.

2.0.2 Process Scheduler

The type of operating system is defined by the *scheduler*, a component that determines which program (or task) executes next. The scheduler in a RTOS is designed to provide a predictable (i.e. deterministic) execution pattern to avoid deadline misses [9]. In a multi-user OS (such as Unix), it will ensure each user a fair amount of the processing time. Furthermore, in a desktop operating system (such as Windows) will guarantee that the computer remains responsive to its user.

2.0.3 POSIX API

POSIX is a core concept in the world of operating systems. POSIX, which stands for *Portable Operating System Interface*, represents a set of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems [13] [14]. An operating system is said to be *POSIX compliant* if it implements the core standard (POSIX.1), some examples are Linux and macOS.

Theoretically, POSIX compliant source code should be seamlessly portable, so it should be easy to run the code on different machines. Yet, in the real world, application transition often runs into system specific issues, in this scenario POSIX compliance makes it simpler to port applications which can result in time savings [15].

POSIX does not define the operating system or its implementation, it only defines the interface between an application and an operating system, the *POSIX API* (Application Program Interface). In addition to the core standard, it also defines several extensions to the standards, including real time extensions and an extension for a thread library.

2.0.4 Schedulers in the POSIX API

A scheduler is the kernel component that decides which runnable thread will be executed by the CPU next. In the standard POSIX API, each thread - the smallest sequence of instructions that can be managed independently by a scheduler [23] - has associated a scheduling policy and a static scheduling priority, the scheduler makes its decision based on knowledge of the scheduling policy and static priority of all threads on the system [20]. Another assumption of the standard, is that all real time scheduling is *preemptive*, if an RT-thread with a higher priority becomes ready to run, the currently running thread will be *preempted* and returned to the wait list or ready queue.

The POSIX standard specifies at least two real time scheduling strategies which can be used, SCHED_FIFO (First-In-First-Out policy) and SCHED_RR (Round-Robin policy). The FIFO scheduling algorithm is a simple algorithm that executes tasks in the exact order of arrival, using a queue. The highest priority real-time task at the front of the queue will be granted the CPU until it terminates or blocks. Instead, Round Robin is similar to FIFO but provides time-slicing among threads of equal priority, so each thread is allowed to run only for a maximum time quantum and after that is inserted at the end of the queue for its priority.

3 Gang Scheduling

Over the years, multicore system have become increasingly commonplace, they are now into desktop computers, laptops, and even in embedded systems [3]. To take advantage of this technology, applications should be written to run in *parallel*, for example using *threads*. Moreover, to handle multiple threads and to take advantage of the parallelism have been developed many scheduling algorithms, such as *co-scheduling* and *gang scheduling*. The first one is the principle of scheduling related tasks to run on different CPUs concurrently [22], meanwhile the second is a stricter form of co-scheduling. In fact, the gang algorithm requires all threads of the gang

to run simultaneously, whereas co-scheduling allows for *fragments*, sets of threads that do not run at the same time as the others [8]. The term *gang*, in this scenario, identifies a group of tasks that run in parallel, as a unit, on different time-shared CPUs. In many parallel and distributed computing systems, a gang scheduler is used to improve communication and data sharing between processes.

There are different versions of the gang policy targeted at specific applications. For example, the *allocation* of the tasks before and after the *preemption* - the act of interrupting an executing task, with the intention of resuming it at a later time [16] - may differ. The simplest version of gang scheduling always allocates a gang in the same set of cores, so the tasks are pinned to the cores and the allocation does not change after the preemption. However, more flexible versions have been proposed. One of them is *migratable preemptions*, where jobs that are preempted in a set of processors can be resumed in a different set of cores. A more complex version is *malleable gang scheduling*, where jobs can be resumed in a set of cores of different size [6].

Other types of gang scheduling are *static* and *dynamic* gang scheduling, the difference between the two is the formation of a gang. The most common form of gang scheduling is *static gang scheduling*, in which the set of tasks to be scheduled together is fixed in advance. This requires that the tasks to be scheduled together are known prior than the scheduling, which is not always the case. *Dynamic gang scheduling* is a more versatile form of gang scheduling, in which the set of processes to be scheduled together can be changed at run time. This is more difficult to implement, but can be more effective in practice.

Even the order in which the tasks are scheduled may differ. The order usually depends on the *priority* of the gang, this parameter can be computed in various ways. For instance, in a FIFO gang scheduler the gang has a static priority, and it executes until all the tasks have terminated or a gang with an higher priority becomes active. Instead, with *Gang EDF (Earliest Deadline First)* the priority depends on the deadline. Another algorithm similar to FIFO is *Round-Robin*, which has a static priority, however the gang executes periodically and is limited by a time quantum, this improves the fairness of the scheduler.

3.1 Gang scheduling in RTOS

Gang scheduling is also used in real-time systems to improve the predictability of process execution (WCET) and to avoid process interference from shared resources, which can lead to missed deadlines. With gang scheduling, interference can be avoided scheduling only tasks that need to run together and share resources in a constructive way, such as exchanging messages. This allows to have tight and accurate WCET, and to take advantage of the parallelization while still avoiding conflicts between tasks.

Gang scheduling has been shown to be effective in reducing the execution time of parallel applications. However, in RTOS it can also lead to increased core idleness and it may reduce processor utilization. To avoid under-utilization have been proposed various solutions, for example the creation of *virtual gangs* or the scheduling of *best-effort* tasks on idle cores [2]. However, running multiple gangs together to form a virtual gang, while it reduces process idleness, it defies the purpose of using the gang scheduling in real-time OS, that is to reduce task interference. The second solution is to schedule task best-effort on idle cores, this type of task does not have time constraints but it can still hinder real-time tasks. A solution to further avoid interference from them is to implement a memory bandwidth throttling mechanism to reduce the memory

usage of these tasks and to benefit the gang.

4 Design of the scheduler

4.1 Background: PSE53

PSE53 is a novel real time operating system developed at the Huawei's Research Center in Pisa. It offers a PSE53 API, which is a subset of the POSIX API, it includes process and thread support, as well as a minimal file system. The distinctive feature of PSE53 is a *modular scheduling framework* that allows the addiction of multicore scheduling policies, such as gang.

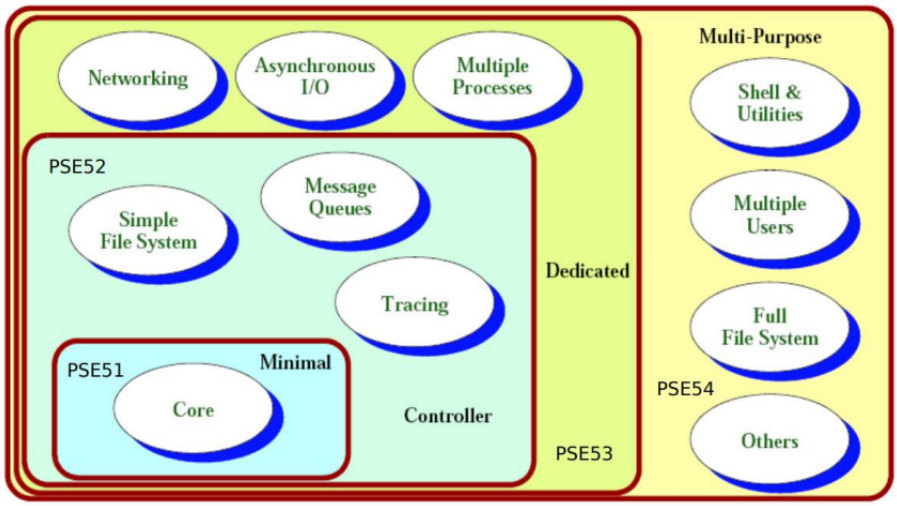


Figure 4.1: POSIX API

The kernel itself is somehow independent of a specific scheduling policy, in fact, the implementation of a modular scheduler interface allows freedom in the configuration of the scheduling modules at startup. I tested the gang scheduler alongside a FIFO, Round-Robin and EDF scheduling modules.

4.1.1 Scheduling module in PSE53

The PSE53 OS allows the implementation and the configuration of many scheduling modules. A scheduling module implements one or more scheduling algorithms (such as FIFO, RR, EDF, GANG, ...). These modules are registered at startup time, and each module is associated to a fixed priority that the kernel use to understand which is the next task to schedule, from high to low priority. In this scenario, the GANG module is initialized with the highest priority, as shown in Figure 4.2. The not-gang tasks are considered *best-effort* and are scheduled with schedulers of lower priority than the gang one.

In PSE53, each thread has a *thread model*, that allows to define a set of parameters, such as priority, deadlines, budgets, and so on. After the creation, a thread is assigned to a scheduling module through a matching process with its thread model.

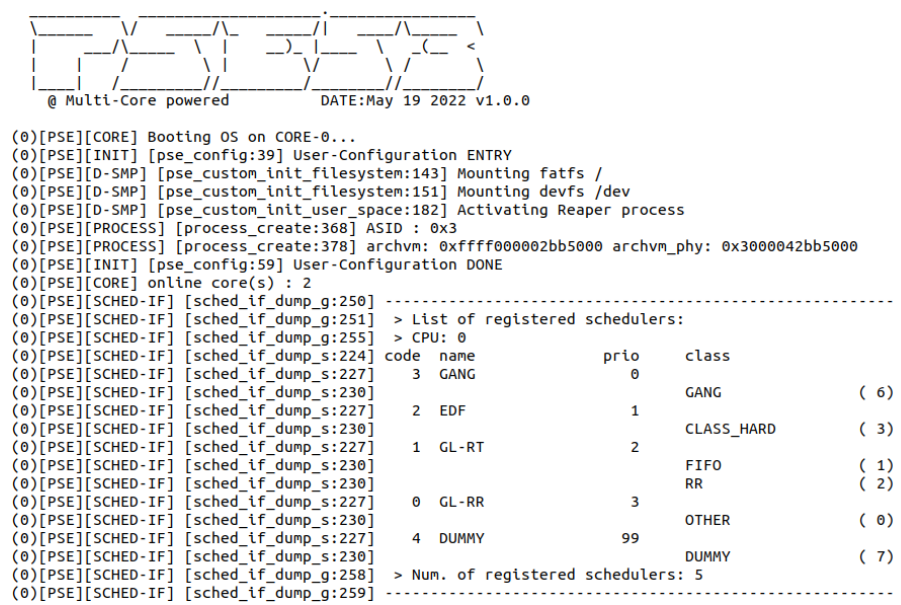


Figure 4.2: Scheduling modules of PSE53

4.2 The gang scheduling module

The idea of the gang scheduling algorithm is that all the threads that compose a gang should run in parallel, to take advantage of the multiple cores in the system. Also, to avoid resource conflicts and having an accurate WCET, there cannot be *more than one gang running at the same time*, this is a difference from the standard gang algorithm but is important in a real-time system to avoid deadline misses [2]. During this research, it has been assumed that the WCET is estimated in isolation.

The configuration and formation of the gang is made by the user, so we can assume that within the gang there are not conflicts of resources and the tasks do not interfere with each other. Moreover, the tasks that form the gang cannot exceed the number of CPUs.

A known issue with the gang scheduling policy is the resource under-utilization, this problem can be addressed in multiple ways: for example, allowing tasks of multiple gangs to form a *virtual gang*, so we can co-schedule multiple gangs at the same time, or to schedule best-effort tasks in the idle cores. The virtual gang concept defies the one gang at a time policy, so I did not implement it, instead I choose to allow idle cores to schedule best-effort tasks. In PSE53, the scheduling of best-effort tasks is handled by other scheduling modules (FIFO, Round-Robin, EDF, ...). The differentiation between gang and not-gang tasks is possible thanks to the modular scheduling frameworks of PSE53. In fact, the gang module handles exclusively *gang* tasks, so the scheduler should only guarantee that two or more different gangs cannot run at the same time; if a task is not gang we consider it best-effort and is has to be handled by another scheduler. Moreover, thanks to the priority of the scheduling modules, it is guaranteed that *gang tasks* preempt all the others, in fact the gang scheduler has the highest priority among all the modules in the system.

There are more variants to the gang algorithm; in this thesis is presented an algorithm similar to the FIFO scheduler available in the POSIX API. In addition, it has been taken the article *RT-gang* by Waqar Ali and Heechul Yun as baseline for the design of the algorithm [2]. The RT-gang algorithm has been adapted to the different scenario of PSE53. In fact, the gang implementation presented in the research paper is made for the Linux OS, meanwhile working with PSE53 allowed to modify directly the OS source code and to more possibilities to the scheduler implementation.

The main difference between the RT-gang paper and the implementation presented in this thesis is the value that identifies the gang. In the article the parameter that differentiates one gang from the other is the *priority*, instead the gang scheduler developed in PSE53 has a specific *ID*, an integer representing the gang. Using an ID, instead of the priority, to represent the gang allows the creation of multiple gangs with the same priority.

The gang algorithm is designed similarly to the Linux SCHED_FIFO, however a difference from the standard SCHED_FIFO policy is the implementation of the *ready queue*. A scheduling algorithm can have one or more queues where the scheduler inserts the tasks that are *ready* to run, and usually the ones with the higher priority are the ones at the top of the queue and the first to be chosen to be executed. The FIFO scheduler in Linux supports multiple priorities, at least 32, and each task with the same priority is inserted in the same ready queue. It is up to the scheduler to determine which thread runs next and how to handle the tasks in the queues. However, the version of gang scheduling that has been implemented does not have multiple queues, the gang supports a priority as parameter but all the gangs are inserted in the same queue. In PSE53, the queues are owned either by the kernel or the scheduling modules, each scheduling module maintains a ready queue for all the thread it owns.

As mentioned earlier, it was first developed a FIFO-like scheduler, so the first gang to be started would execute until the end and all the others are inserted in a ready queue in order of arrival. Later on, it was added a preemption mechanism, so the gang with the highest priority is the one that is scheduled, and the ready queue becomes an *ordered* queue.

4.2.1 Preemption Mechanism

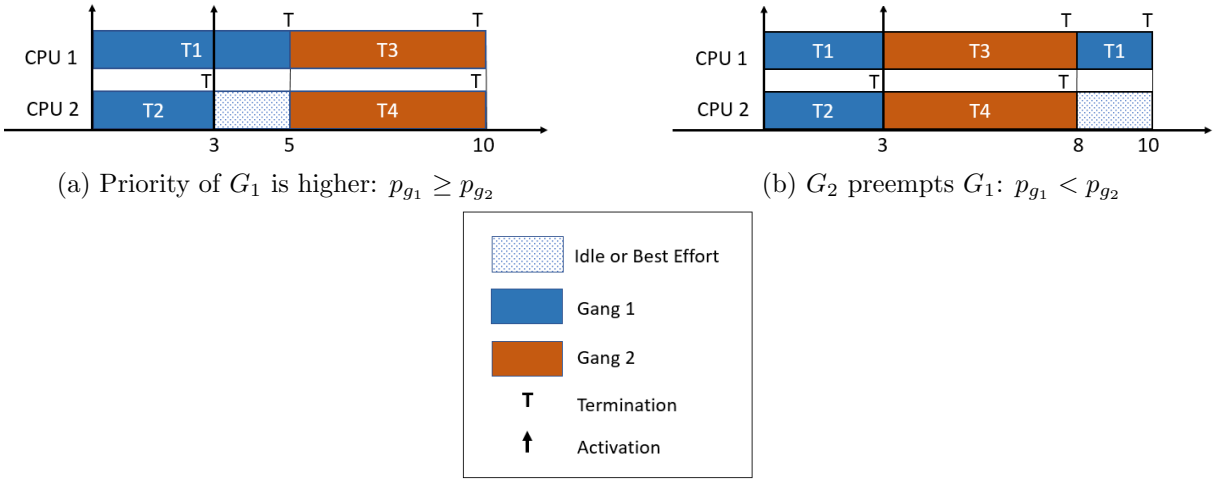
Preemption is the act of temporarily interrupting an executing task, with the intention of resuming it at a later time, this may be done to guarantee that is always running the thread with the highest priority. Such a change in the currently executing task of a processor is known as *context switching* [16]. In the case of the gang scheduler, the preemption mechanism works confronting the priority of each gang, when a new gang is *activated*, the scheduler performs a *query_next* operation that verifies whether the new gang priority is higher than the one that is currently running, if this is the case then the running task is *preempted*, so is inserted again in the ready queue and the new gang can execute.

4.2.2 Illustrative Example

Task	WCET (C)	Gang ID
T1	5	1
T2	3	1
T3	5	2
T4	5	2

Table 4.1: Taskset parameters of the example

Figure 4.3a and 4.3b show how the gang preemption mechanism should work. As we can see from table 4.1, there are two gangs with different priority, and four gang tasks. T1 and T2 are associated to the first gang, G_1, while T3 and T4 are associated to the second, G_2. Each gang has two tasks associated, so in order to execute this example, we should have at least two available cores. The tasks of the first gang are activated before the ones belonging to the second one and the only parameter that changes between the examples is the priority: in figure 4.3a the priority of G_1 is higher than the priority of G_2; whereas, in figure 4.3b is the opposite. T2 always terminates before T1 and frees its allocated resources. After that, the main process can execute and activate the second gang, which is inserted in the ready queue. Here, the scheduler performs a *query_next* function and if the priority of the new gang is higher than the gang currently running (as shown in figure 4.3b) the second gang *preempts* the first one: the



(a) Priority of G_1 is higher: $p_{g_1} \geq p_{g_2}$

(b) G_2 preempts G_1: $p_{g_1} < p_{g_2}$

Figure 4.3: Gang scheduling example with preemption

scheduler sends an *interrupt* to the first core and T1 is replaced by T3.

Instead, in figure 4.3a, G_1 cannot be preempted by G_2 since its priority is higher, so G_2 has to wait until the end of the first gang in order to start.

In both the tests, when the core remains idle, the gang policy allows *best-effort* tasks to execute. In this thesis are considered best-effort (BE) all the tasks that are not gang tasks, for example FIFO or RR tasks.

4.3 Scheduling

A scheduling module implemented in PSE53 is composed by a series of functions, specific for each scheduler, but the scheduling mechanism is the same for all the modules. In general, in order to perform a scheduling decision three main operations are executed by the scheduler: a *detain*, a *query_next* and a *dispatch* function. Each operation is associated to a specific method in the modules, depending on the task model. For example, if a task is a *gang task*, then the *detain* function will execute the *GANG_detain_task* method, then the *GANG_query_next_task* and the *GANG_dispatch_task*. This scenario happens every time the system needs to perform a scheduling decision due to any "special events", for example, when a thread is preempted by a newly activated thread.

4.4 Life cycle of a task in PSE53

Figure 4.4 shows the life cycle of a task in PSE53, this paragraph will describe the main phases. When a thread is created, with the *pthread_create* function, a task is firstly owned by the kernel. The kernel allocates a *task_descriptor* for each task in the system to keep track of all the *free tasks* in the OS (i.e. the ones that are not associated to a scheduler). A task is *removed* from the descriptor when it is created in a scheduling module, and *inserted back* when it terminates. After the termination, the kernel can deallocate all the resources associated to the task with the *task_free_descriptor* method.

Create

The operation to associate the task to the scheduler is the *task_create* function. A thread is created using a *task model*; each model has different parameters so it can be accepted only from some schedulers. For instance, a *GANG task* model has one parameter, the GANG ID, and it can be handled only by a gang scheduler. There can be task models that can be accepted from more than one scheduler in PSE53.

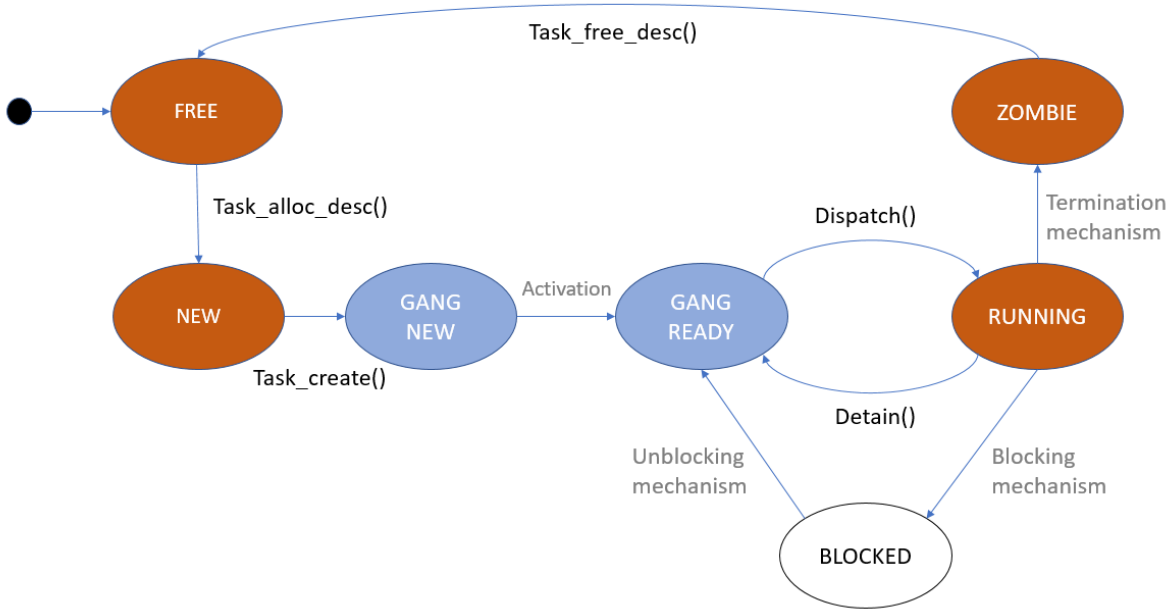


Figure 4.4: Task states

Activate

After the creation, each scheduling module oversee all its tasks. The task is ready to execute but it cannot be selected from the scheduler until the *activation* function. The activation function is called after the creation, its main job is usually to insert the task in the ready queue so the scheduler can select it. However, in the gang scheduler, the activation is different because a task cannot be inserted in the ready_queue until all the other threads of the gang have been created.

Query next and Dispatch

After the task is inserted in the ready_queue, it can be selected from the scheduler. The scheduler chooses which task is the next to run; this operation is done for the CPU that is currently executing the scheduler. In a multicore system, the scheduler can run simultaneously on each core.

In the GANG scheduling module, the *query_next* method should return the task to schedule next, considering: the gang that may be already running in other cores and the gangs in the ready_queue. The next task should be one of the running gang if its priority is higher than all the other tasks that are waiting to execute, otherwise should be the first one of the ordered queue; this new gang will preempt the one currently running in the other cores, as shown in figure 4.3b.

The operation that comes after selecting the next task to execute is the *dispatch*. The dispatch function removes the task from the ready queue of the scheduler and the task becomes RUNNING. In fact, a RUNNING task is an executing one and it is owned by the kernel, not the scheduler.

Detain

When a task stops executing, it can return ready to run with a *detain* function. This function generally inserts a task in the ready queue. Furthermore, the *GANG_detain_task* function, before inserting a task in the ready queue, verifies that a task of the same gang has not been already inserted. In fact, in this implementation only one task of the gang can be present in the queue, this task represents the entire gang.

Termination

When a task terminates, the scheduler and the kernel can release all the previously allocated resources. The termination mechanism of the gang module is more complex than the ones of the other schedulers because when a thread that belongs to the currently running gang ends, the gang scheduler has to verify if there are other tasks of the same gang that are still executing. The last gang task to end should signal the scheduler that another gang can run; the implementation on how this is achieved will be discussed in the next chapter.

Block and Unblock

With the utilization of *mutexes* (or locks) in user space, to synchronize threads, a task can become BLOCKED. A mutex is an object that can protect a shared resource; it is called before and after the locked region of code. When a thread, inside its body function calls a *pthread_mutex_lock*, the tasks that want to gain access to the locked resource are suspended (i.e. blocked) and cannot execute until the *pthread_mutex_unlock* function.

From the scheduler point of view, if a thread locks a portion of code, a function *block_task* is called when other threads try to access the locked resource. This function also notifies the scheduler that the task is *blocked*, this can be done using a flag *"blocked"*.

After setting the lock, if the query_next function is called and a thread of the running gang is blocked, the function should not return anything. However, in the meantime, other tasks best-effort can execute.

At the end of the locked portion of code, the thread unlocks the resource and the scheduler runs the *unblock_task* function. This function, in the gang scheduler, aims to unblock all the blocked tasks (recognized with the *"blocked"* flag) and forces a *reschedule* operation sending an interrupt to all the cores where blocked tasks were previously running. This allows all the tasks of the gang to run again simultaneously.

4.5 Task states

As shown in figure 4.4, we can identify different states for the different phases of the life of a task. TASK_FREE, TASK_NEW, TASK_RUNNING and TASK_ZOMBIE are states owned by the kernel, meanwhile TASK_GANG_NEW and TASK_GANG_READY are owned by the gang scheduling module. Each scheduler has different states, I reported only the ones concerning the gang module. In PSE53, each scheduling module is responsible to define and set the status of the task when it becomes the owner of the task.

The first one, TASK_FREE, represents a task in the free pool of the kernel, the resources of the task have not been allocated yet. After the allocation of the task descriptor in kernel space, a task is said to be TASK_NEW, in this state is not owned by any scheduler yet. A TASK_GANG_NEW identifies a task owned by the gang scheduler but not ready to execute. To represent a ready task, we can set a TASK_GANG_READY status. Ready tasks are the ones that could run (they are not in the BLOCKED state) but are not currently executing because a different task of equal or higher priority is already in the RUNNING state. TASK_RUNNING is the state of an executing task; the task is currently utilizing the processor.

A TASK_ZOMBIE is a task that has completed its execution, and it is waiting to be joined.

The TASK_BLOCKED status identifies a task which is waiting an event, that can be a semaphore, a condition variable, a mutex, a file system operation or a delay operation (such as yield, sleep, sigwait,...). The blocked task is owned neither by the scheduler nor the kernel. Moreover, tasks in the BLOCKED state cannot be selected to enter the RUNNING state.

```
-------------------------------------------
> List of registered status code:
code  name
   0  TASK_FREE
   1  TASK_NEW
   2  TASK_ZOMBIE
   3  TASK_JOIN
   4  TASK_RUNNING
   5  PROC_ACTIVATED
   6  PROC_TERMINATED
   7  WQ_BLOCKED
   8  MTX_BLOCKED
   9  WAITPID_ANY_BLO
  10  WAITPID_PID_BLO
  11  GLRR_READY
  12  GLRT_READY
  13  GLRT_NEW
  14  EDF_IDLE
  15  EDF_FAULT
  16  EDF_NEW
  17  EDF_READY
  18  GANG_NEW
  19  GANG_READY
  20  DUMMY_READY
> Num. of registered status code: 21
-------------------------------------------
```

Figure 4.5: Task states of PSE53

5 Implementation

The scheduler was implemented using the C language. The scheduling module is formed mainly by two files, a .c and a .h.

```
valeria@valeria-VirtualBox:~/workspace/pse53/core/schedulers$ cloc gang/
       6 text files.
       6 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 1.82  T=0.02 s (351.0 files/s, 58727.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                                2            153            176            538
C/C++ Header                     3             22             63             36
CMake                            1              1              0             15
-------------------------------------------------------------------------------
SUM:                             6            176            239            589
-------------------------------------------------------------------------------
```

Figure 5.1: Lines of code

5.1 Data Structures

The gang scheduler works using two main data structures: a *gang_task* data type and a *gang_descriptor*.

5.1.1 Gang task

This data structure contains all the parameters used by the gang task.

5.1.2 Gang descriptor

This data structure defines the *gang* object in the scheduler.

```
struct gang_task
    int gang_id;                /* ID of the gang to which the task belongs */
    bool running;                                /* 1 if the task is running */
    bool blocked;                                /* 1 if the task is blocked */
    gang_desc *gang;          /* pointer to the gang containing the task */
end
```

```
struct gang_desc
    int gang_id;                                                /* ID of the gang */
    int priority;                                           /* priority of the gang */
    bool ready;                                  /* 1 when the gang is ready to run */
    gang_task *tasks[MAX_CPU];  /* array of tasks associated to the gang */
    bool inqueue;                               /* 1 when the gang is in ready queue */
    bitmask ready_tasks;                            /* mask with the not-ended tasks */
end
```

5.2 Gang functions

All the schedulers in PSE53 should have some standard methods:

- init: initialize the scheduler
- create: creates the tasks inside the scheduler
- activate: inserts the tasks in the ready queue
- query next: selects the next task to be executed
- dispatch: removes the task from the ready queue, the task then becomes running and is owned by the kernel
- detain: reinsert the task in the ready queue and sets the task as ready to run
- block: sets a task as blocked and prevents the blocked task from been chosen by the query next function
- unblock: sets the task as not blocked
- yield: relinquish the control of the CPU from the task and insert it at the end of the ready queue
- destroy: frees all the resources allocated by the task

In addition to these methods, two *custom system calls* are used to define some non-standard behaviours of the gang scheduler. The first system call allows creating new gangs, and the second to signal that a gang is ready to start.

5.2.1 Custom system calls

PSE53 allows defining custom system calls. The typical usage of this kind of system call is the implementation of non-POSIX-standard behaviours that are somehow linked to the custom scheduler semantic.

Firstly, has been implemented a custom system call to allocate a gang in the scheduler and generate a unique identifier to reference the gang, *GANG_ID*. Secondly, has been implemented a *GANG_START* system call that handles the activation of the gang.

System call to generate a gang identifier

This function is called before the creation of the gang threads. It allocates a gang descriptor in the scheduler and generates a unique ID to identify the newly created gang. The identifier is an integer that is returned to the user. The user can associate the task to the gang by passing the *gang id* to the scheduler at the creation of the task (using the gang task model).

The gang is added to a list of gangs that is used to keep track of both the gangs that are ready to execute (or running) and the ones that can still accept other tasks.

System call to activate the gang

This custom system call notifies the scheduler that the current gang, identified by the ID, is ready to execute, all the task have been already created and added to the gang. After this system call has been executed, no more tasks can be inserted in the gang.

This function calls a *task_activate* function after setting the flag *ready* of the gang to 1. This parameter is important because the *GANG_activate_task* method adds to the list of *ready tasks* only the gangs with the flag ready to 1.

5.2.2 Initialization function

The initialization function is called at startup time. It is responsible for allocating and initializing all the resources. Moreover, it registers the scheduler in the system and binds the private functions of the scheduling module to the *function pointers*, used to associate the task model to the schedulers of PSE53. For instance, if a task is defined as FIFO then the FIFO scheduler methods are called, instead of the gang ones.

Furthermore, the gang scheduler uses a *gang_running* data structure, it is a global *gang_desc* pointer that each time saves the currently running gang. It is used by the module to keep track of the running gang and to make scheduling decisions on the different cores.

5.2.3 Gang Create function

The gang create function is the first function called after the standard POSIX pthread_create function, the main objective of this method is to associate the given task to a gang. This is done through the *gang_id* parameter of the gang task. In fact, the gang descriptor with the corresponding *gang_id* has already been created with the system call, GANG_ID. The GANG_Create_task method, however, should always return an error if the *gang_id* of the task does not exist.

Algorithm 1: GANG_Create_task

input : Gang task (*tsk*)

gang_desc *gang* = get_gang_desc(*tsk*→*gang_id*);
tsk→*gang* = *gang*;
tsk→*running* = 0;
add_to_gang(*tsk*);
set_status(*tsk*, NEW);

5.2.4 Gang Activate function

The gang is activated from user space through the GANG_START custom system call described above.

Implementation:

Firstly, the activate function checks if the gang has been activated, i.e. the flag *ready* is equal to one. If the condition is met, it inserts a gang in the ready queue. Since the queue contains tasks, not gangs, only one task of the gang is inserted, its representative. The activation function is called just once and activates the entire gang.

Algorithm 2: GANG_Activate_task

input : Gang task (*tsk*)

if (*tsk→gang→ready*) **then**
  insert(*tsk*,*ready_queue*);
end

5.2.5 Gang Query Next function

The GANG_query_next function is the most important method of the scheduler, in fact, it chooses which task to schedule next. The task selection is done through a series of controls. First of all, if there is a gang currently running, i.e. $gang\_running \neq NULL$, then it verifies whether there is a *task ready* that can be selected to be dispatched on the CPU that is executing the scheduler.

After that, it is important to check if the currently running gang has the *highest priority* among all the tasks in the ready queue, if that is not the case, then the gang with the highest priority in the queue is selected as the next task.
If there are no tasks that match all the controls, then the function returns *NULL*.

Algorithm 3: GANG_Query_next_task

output: The next task to schedule

task *next* = *NULL*;
if (*gang_running* ≠ *NULL* ***and*** *test_bit(cpu, gang_running→ready_tasks)* ***and***
 !(*gang_running→tasks*[*cpu*]*→running*)) ***and*** !(*gang_running→tasks*[*cpu*]*→blocked*))
 then
  next = *gang_running→tasks*[*cpu*];
end
if (!*empty*(*ready_queue*)) **then**
  tsk = *get_first_entry*(*ready_queue*);
  if (*gang_running* = *NULL* ***or*** *tsk→gang→priority* > *gang_running→priority*)
   then
    next = get_first_task(*tsk→gang*);
  end
end
return *next*;

5.2.6 Gang Dispatch function

The GANG_dispatch_task function is called after the GANG_query_next_task selects a task to execute next. The *dispatcher* has the job of removing tasks from the ready queue so that they can be executed by the kernel and become *running*. In this case, if the task is part of the *running_gang*, the representative of the gang has already been removed from the queue. In fact, only the first task of the gang to be dispatched exist in the queue, and it represents the entire gang.

The dispatched task executes and it is handled by the kernel until it is suspended or it terminates.

Implementation

Firstly, the function verifies if the task is part of the currently running gang. For instance, if there is not a gang currently running or there is a gang in the ready queue that has higher priority, then the gang in the ready queue is the one that executes. Secondly, the dispatch function extracts the representative of the gang from the ready queue. Lastly, it sends a *reschedule* IPI (Inter Process Interrupt) to the cores where a task of the gang can be scheduled. The IPI notify the CPU to perform the detain, query next and dispatch operations to schedule a new task.

Algorithm 4: GANG_Dispatch_task

input : Task *tsk* returned by GANG_query_next(*tsk*)

```
tsk→running = 1;
if (gang_running = NULL or tsk→gang→priority > gang_running→priority) then
    extract(tsk, ready_queue);
    gang_running = tsk→gang;
    gang_running→inqueue = 0;
    foreach cpu in CPUs do
        if test_bit(cpu, gang_running→ready_tasks) then
            reschedule(cpu);
        end
    end
end
```

5.2.7 Gang Detain function

The GANG_detain_task function is responsible for reinserting a task in the ready queue, this is done calling the GANG_rq_insert method. It also and sets the task status to READY, notifying that the task can be selected again by the query_next function. If there is already a representative of the *gang* in the *ready_queue*, i.e. the flag *inqueue* of the gang is equal to one, the task is not inserted.

Algorithm 5: GANG_Detain_task

input : Gang task (*tsk*)

```
GANG_rq_insert(tsk);
```

Algorithm 6: GANG_rq_insert

input : Gang task (*tsk*)

```
tsk→running = 0;
set_status(tsk, READY);
if (!(gang_running = tsk→gang) and !(tsk→gang→inqueue)) then
    insert(tsk, ready_queue);
    tsk→gang→inqueue = 1;
end
```

5.2.8 Block unblock behaviour

This function is called when a task tries to access a previously *locked* portion of code. The boolean values *running* and *blocked* are used to notify the state of the task. In fact, in the GANG_query_next method, a task can be selected only if not blocked.

Algorithm 7: GANG_Block_task

input : Gang task (*tsk*)

```
tsk→running = 0;
tsk→blocked = 1;
```

The GANG_unblock function is called by the task after it *unlocks* a protected resource, it unblocks all the blocked gang tasks and insert them in the ready queue. To insert them in the ready queue, it calls GANG_rq_insert, the same function called by the detain method. Moreover, it sends an interrupt to the cores where the blocked tasks were previously running to see if they can be rescheduled.

Algorithm 8: GANG_Unblock_task

```
input : Gang task (tsk)
GANG_rq_insert(tsk);
foreach task in gang_running→tasks do
    if task→blocked then
        task→blocked = 0;
        reschedule(task_cpu);
    end
end
tsk→blocked = 0;
```

5.2.9 Yield

Yield is an action that occurs in a computer program during multithreading, it forces a processor to relinquish control of the current running thread, and sending it to the end of the ready queue [24]. The implementation of the yield function has the same behaviour as the one in the Linux Operating System. In fact, the yield function moves the thread to the end of the queue and a new thread gets to run.

If the calling thread is the only thread in the highest priority list at that time, it will continue to run even after the standard POSIX sched_yield function has been called [21].

The task is inserted in the ready queue with the GANG_rq_insert method, the same as the detain function.

Algorithm 9: GANG_Yield_task

```
input : Gang task (tsk)
GANG_rq_insert(tsk);
```

5.2.10 Destroy

The destroy function is called after the termination of the task, all the resources associated to the task are removed and a custom function GANG_end_task is called to remove the task from the gang data structure. When all the *gang* tasks have been removed, the *gang_running* pointer is set to *NULL*, allowing the GANG_query_next to choose another *gang* to schedule.

Algorithm 10: GANG_End_task

```
input : Gang task (tsk)
remove_from_set(tsk_position, gang_running→ready_tasks);
if (mask_is_zero(gang_running→ready_tasks)) then
    gang_running = NULL;
end
```

6 Testing

Testing is an indispensable part of quality assurance for embedded software. Nowadays, the standards for safety-critical software development set precise requirements for test methods and test coverage. As a rule, the more critical the application, the higher are the requirement concerning the code coverage.

A series of tests has been implemented to test the gang scheduler. These tests run sequentially and allow checking if the scheduler is behaving correctly. In fact, have been developed tests that verify the correctness of the preemption mechanism, check the block and unblock mechanism, the sched_yield behaviour and also that tested the gang scheduler alongside other schedulers of

tasks best-effort.

Furthermore, has been tested the code coverage, the Misra C compliance and has been implemented a test that verifies if the scheduler can execute for a long time and handle numerous tasks without errors.

Lastly, has been added a test to prove one of the benefits of the gang scheduler. In fact, the gang policy allows to synchronize tasks using busy waiting. In the test is shown the difference between synchronizing tasks that are scheduled with FIFO and GANG.

6.1 Lauterbach TRACE32

Lauterbach TRACE32 is a software debugger that allows developers to quickly and easily develop and debug code for embedded systems. Moreover, it provides a trace port for code coverage and instruction trace. It has been used to debug the scheduler and verify the trace of the tests.

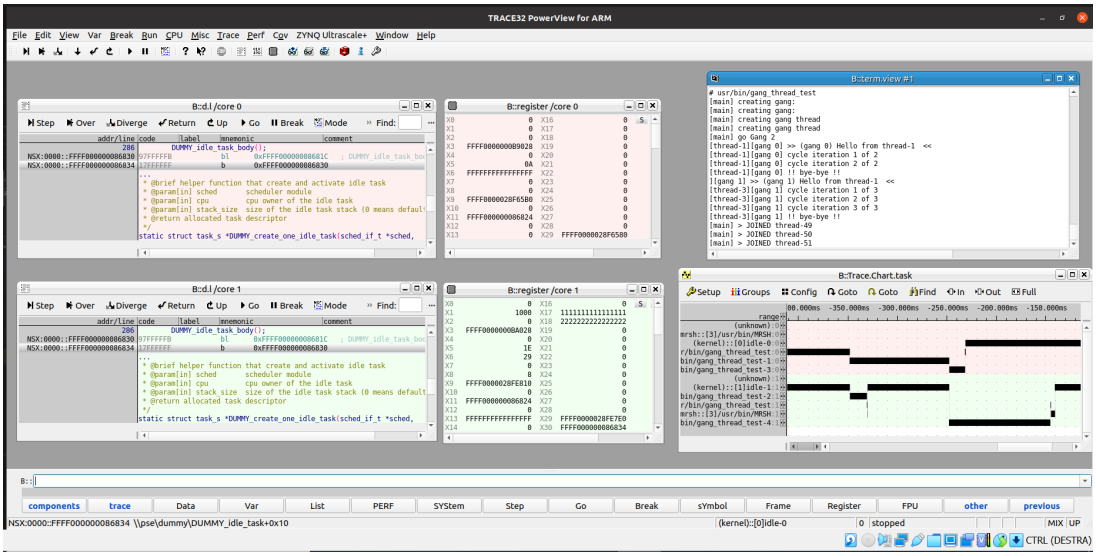


Figure 6.1: Screenshot from Lauterbach TRACE32

Figure 6.1, represents the Lauterbach TRACE32 software, it was taken after the execution of one of the tests. The white window on the right shows the terminal, the one below it is the trace of the test. The two windows on the left show the source code that is being executed; in the middle there are the CPU registers. The pink colour refers to the first core, meanwhile the green represents the second one.

6.2 CUnit

The aspects of the scheduler were tested using the CUnit testing framework. CUnit is a lightweight system for writing, administering, and running unit tests in C. It provides a basic testing functionality with a flexible variety of user interfaces [7].

6.3 Test suite

The tests were implemented to run in succession by adding them to a *test suite*, a collection of tests. CUnit provided an easy interface and allowed to use *asserts* to check the correctness of the *control variables*.
The tests designed to work correctly with two cores; some tests have been tested on a XILINX Ultrascale+ and others on a Lauterbach TRACE32 simulator.

```
# usr/bin/gang_cunit_test


     CUnit - A unit testing framework for C - Version 1.0
     http://cunit.sourceforge.net/


Suite: Suite_1
  Test: 1 - no gang preemption() ...passed
  Test: 2 - gang preemption() ...passed
  Test: 3 - gang activation time() ...passed
  Test: 4 - gang preempt best effort() ...passed
  Test: 5 - gang best effort() ...passed
  Test: 6 - gang yield() ...passed
  Test: 7 - gang mutex() ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      7      7      7      0        0
             asserts     27     27     27      0      n/a

Elapsed time =   16.517 seconds
#
```

Figure 6.2: Testing suite

6.3.1 Test 1 - No gang preemption

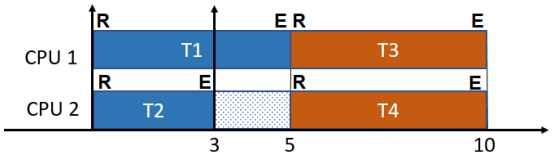


Figure 6.3: Priority of G_1 is higher: $p_{g_1} \geq p_{g_2}$

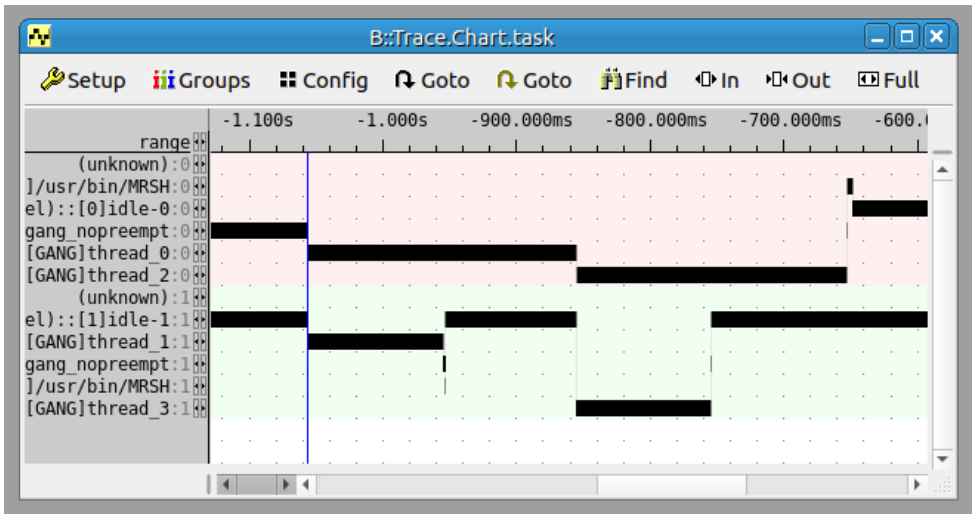


Figure 6.4: Screenshot from Lauterbach TRACE32. Test no preemption

This test check if the preemption is respected. The main process creates four gang threads. The first two threads belong to G_1, meanwhile the other two belong to G_2. The priority of G_1 is higher than G_2, $p_{G_1} \geq p_{G_2}$. The first gang is started before the second, so it is expected that G_1 executes until both its tasks end.

During the execution is monitored the correct order of execution of the tasks, to do so, is used a *control_variable*. This variable can assume different status: READY, RUNNING and

ENDED. At the beginning of the main process, the CV of the different threads is setted to READY for each task. As soon as each task starts, CV is setted to RUNNING (R in figure 6.3) and the last instruction of the body changes the CV to ENDED (E in figure 6.3). Each thread uses different CU_ASSERT in its body to make sure of the correctness of the value of CV.
T_0 and T_1 verify that during their execution T_2 and T_3 are not running yet:

CU_ASSERT(*control_value[2]* = READY **and** *control_value[3]* = READY);

Instead, T_2 and T_3 check that CV_0 and CV_1 is setted to ENDED. In fact, in order to respect the preemption mechanism, when T_2 and T_3 start, T_0 and T_1 should already have finished.

CU_ASSERT(*control_value[0]* = END **and** *control_value[1]* = END);

The minimum number of cores to run this test is two.

6.3.2 Test 2. Gang preemption

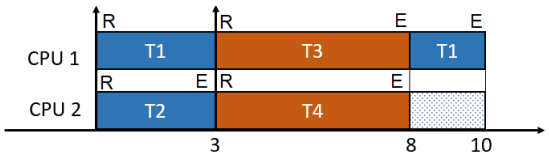


Figure 6.5: G_2 preempts G_1: $p_{g_1} < p_{g_2}$

This test checks if the preemption is respected. It has the same logic as the previous test, however the priority of the gangs is inverted. There are two gangs, the first one has priority lower than the second; if the second one preempts the first gang, then the test passes. During the execution is monitored the correct order of execution of the tasks, to do so, have been used CUnit asserts.
Figure 6.6 shows the trace of the execution of the test.

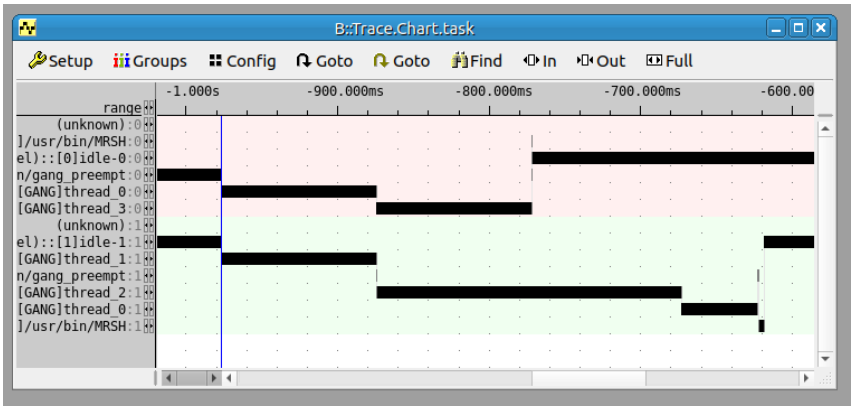


Figure 6.6: Screenshot from Lauterbach TRACE32. Trace of the preemption test

6.3.3 Test 3. Activation time

The third test verifies whether the activation time between two gang threads is within a certain range, it is used to verify if all the tasks of the gang start at the same time. The starting time depends on the interrupts that are being sent by the first gang task to be dispatched, to the other CPUs. In fact, the first CPU that dispatch a gang task has to notify the others to schedule all the gang tasks. If the other CPUs have interrupts disabled, the tasks of the gang will start with a bounded delay. However, assuming the interrupts have not been disabled, has been measured an upper limit to the activation time of all the gang tasks (it depends on the

number of cores), and the test verifies that all the gang tasks have been scheduled fast enough to not exceed this upper bound.

6.3.4 Test 4. Best-effort preemption

This test is used to see if a gang preempts two tasks best-effort that are running prior to the activation of the gang. The tasks best-effort have been scheduled using a scheduler FIFO. Firstly, are created two tasks best-effort, when the shortest one between the two ends, the gang is activated and should preempt the second task best-effort. The preempted task, *[BE]thread_1* in the image 6.7, is then resumed after the gang ends.

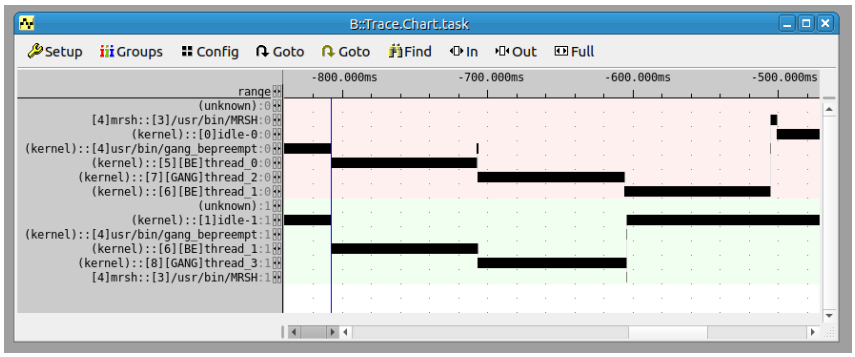


Figure 6.7: Screenshot from Lauterbach TRACE32. Best-effort preemption test

6.3.5 Test 5. Best-effort

This test is used to control if a task best-effort does execute when a gang is running but it does not occupy all the cores, so there is at least an idle core. In fact, the gang policy allows best-effort tasks to run simultaneously with the gang to reduce core idleness.

First, two gang threads are launched, one longer than the other, when the shortest one ends a task best-effort can be scheduled, the test is correct if the task is indeed dispatched.

In figure 6.8, the best-effort task is *[BE]thread_2*, meanwhile *[GANG]thread_0* and *[GANG]thread_1* form the gang. From the image it is clear that when thread_1 ends, the gang is still running but thread_2 can execute.

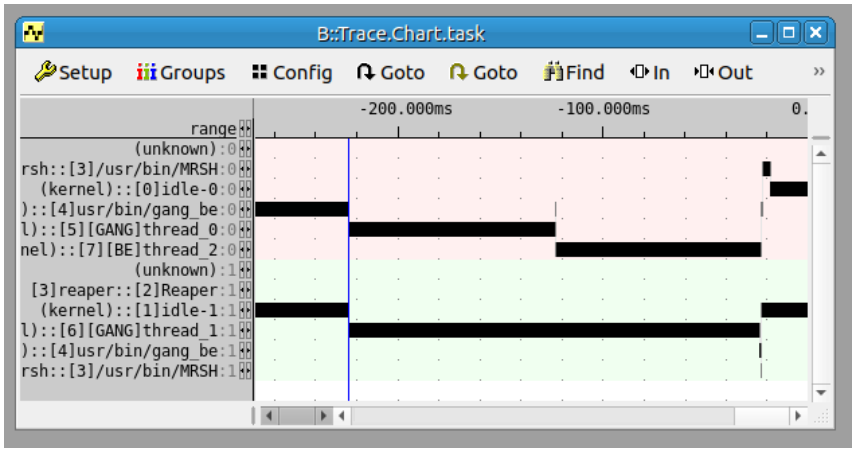


Figure 6.8: Screenshot from Lauterbach TRACE32. Best-effort test

6.3.6 Test 6. Yield

The test verifies the correctness of the *GANG_yield_task* function. This test launches five threads; the first one -with the highest priority, so it cannot be preempted- creates two gangs, with two threads for each gang. These two gangs have different priorities, the one that execute first, i.e.

the gang with the highest priority, calls a *sched_yield()* system call. The test aim to check that after the execution of *sched_yield*, the same gang, the one with the highest priority is scheduled again.

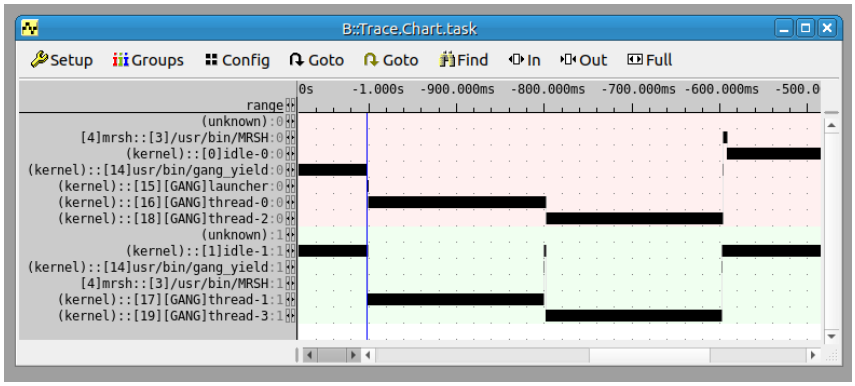


Figure 6.9: Screenshot from Lauterbach TRACE32. Yield test

6.3.7 Test 7. Block and unblock

This test verifies the correct functioning of the gang scheduler block and unblock behaviour. Two gangs threads contends a shared variable and use a *mutex* lock to protect the access to the variable.

It is not recommended to use the block and unblock functions of the scheduler because, as is shown in figure 6.10, the gang task are not running together, it is preferred to use a busy waiting mechanism to synchronize the tasks.

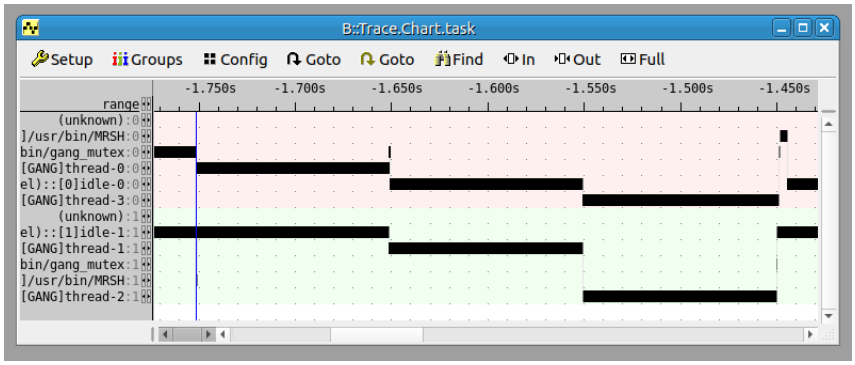


Figure 6.10: Screenshot from Lauterbach TRACE32. Mutex test

6.4 Long test

This test runs an arbitrary number of gangs and numerous task best-effort. The gangs have a random number of tasks associated ($\leq$ *cpu_number*) and also the priority of the gangs is chosen randomly. There is a gang thread whose job is to create all the gang and start them. When this *gang_launcher* thread ends, all the gang have been activated and we can see graphically, using a TRACE32 debugger, the succession of all the gangs and the best-effort tasks. The image 6.11 shows an example of the test launched with 10 gangs and two cores. The first terminal window shows the output of the first core and the second window the output of the other.

The actual test has been executed for several hours without encountering errors, testing the correct functioning of the scheduler in a situation of stress.

In figure 6.11, the black rectangles are gang tasks, meanwhile the green ones are best-effort tasks. The numbers between brackets represent respectively: the priority, the ID of the gang and the length of the gang task (multiplied by 100ms). For instance, [2-10-1] means that the

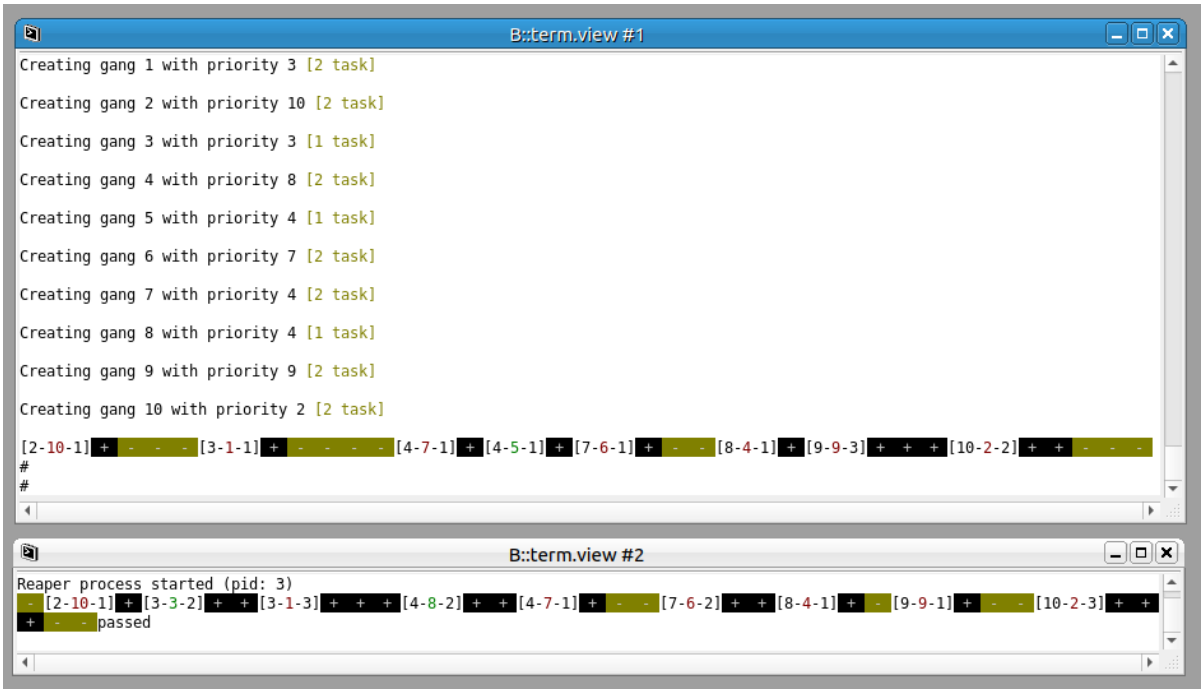


Figure 6.11: Screenshot from Lauterbach TRACE32. Long test

gang has a priority of two (the highest between the 10 tasks), it is the 10th gang and the task that belongs to the gang has a duration of 100ms. Moreover, the color of the gang ID shows if the gang has one (green) or two tasks (red).

6.5 Coverage test

Code coverage is a measure of the amount of code that is executed when a particular test suite runs. A high code coverage means that most of the code is being exercised by the tests. A low code coverage means that there is a lot of code that is not being tested. Code coverage is an important metric to track because it can help to identify areas of code that are not being adequately tested. Moreover, it can be used to improve the quality of the tests and to make sure that all parts of the code are covered by them.

There are a number of ways to measure code coverage. In this case, has been used *function coverage* in Lauterbach TRACE32, which measures the percentage of functions that are called when the test suite runs. To test the coverage has been used the CUnit test suite presented previously in this chapter.

The code coverage for the gang scheduler is 93.589%. This means almost all the functions in the scheduler are being called when the tests are executed. Figure 6.12 shows the coverage of the scheduler, all the functions of the gang scheduler (the ones that are named starting with *GANG*) have a coverage percentage above 90%. Moreover, all the code that could not be reached by the tests launched from user space has been verified.

The function named *GANG_sched_ctl* it is the only *GANG* method with a coverage of 0%. In fact, it is a control method that has not been implemented and is not used in the operating system. However, it was necessary to declare it, as it is one of the methods that PSE53 uses to associate custom system calls to the scheduler.

6.6 Misra C

Misra C is a set of coding standards developed by MISRA, the Motor Industry Software Reliability Association, to promote best practices in developing safety-related systems in the automotive industry. Compliance with the MISRA guidelines is regularly required for software development in industries of the embedded sector to reduce the risk of safety-critical errors in software. To

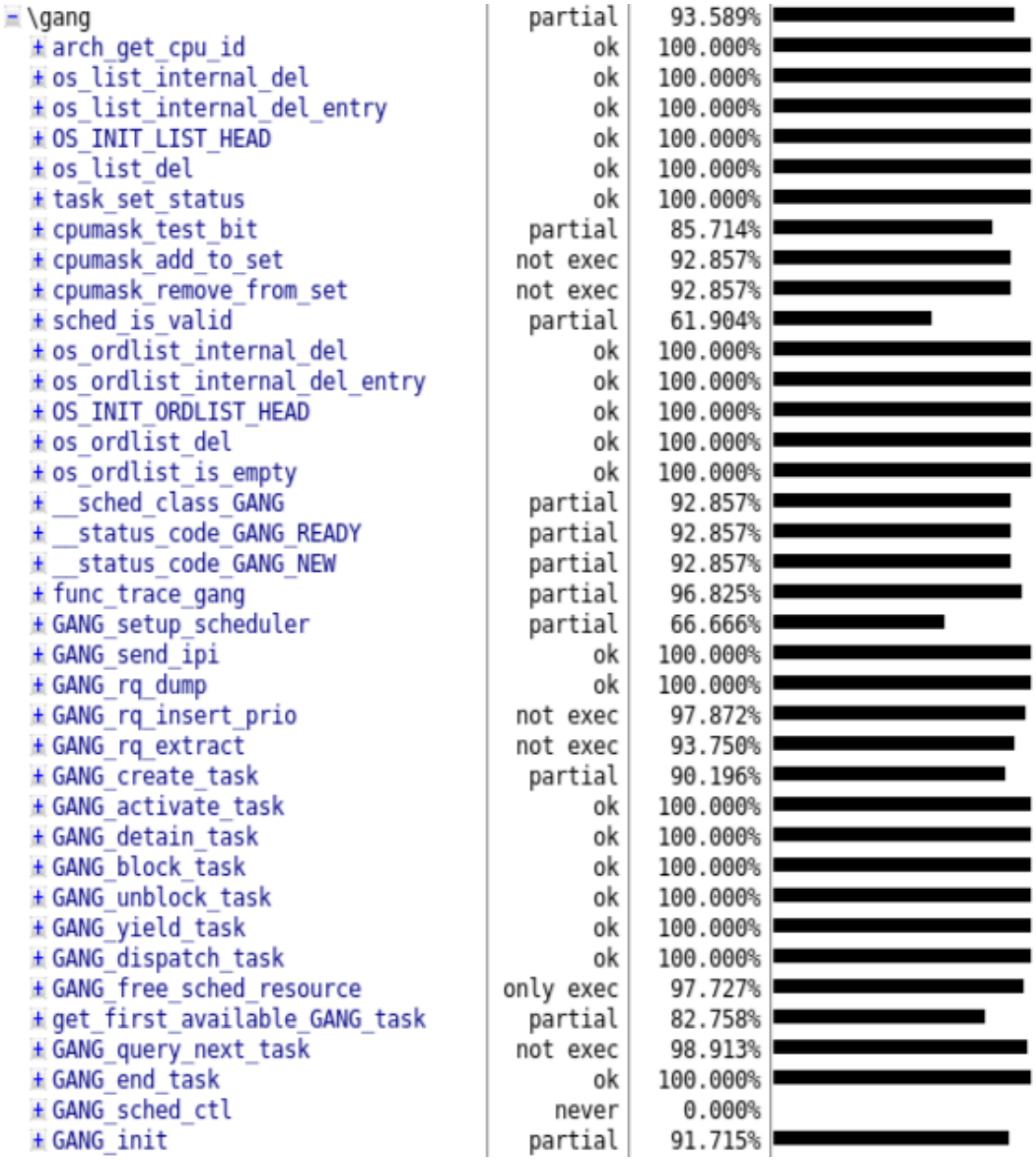


Figure 6.12: Screenshot from Lauterbach TRACE32. Code coverage

test Misra C compliance has been used the *Axivion analysis* tool, a software used to detect errors and style violations. I was able to reduce the style violations, violations of the rules of Misra, from 303 to 99, as shown in figure 6.13. The remaining violations have been checked one by one and could not be removed, in fact they were mostly void pointer errors. Misra does not allow the use of void pointers, however they were necessary on the implementation of the scheduler.

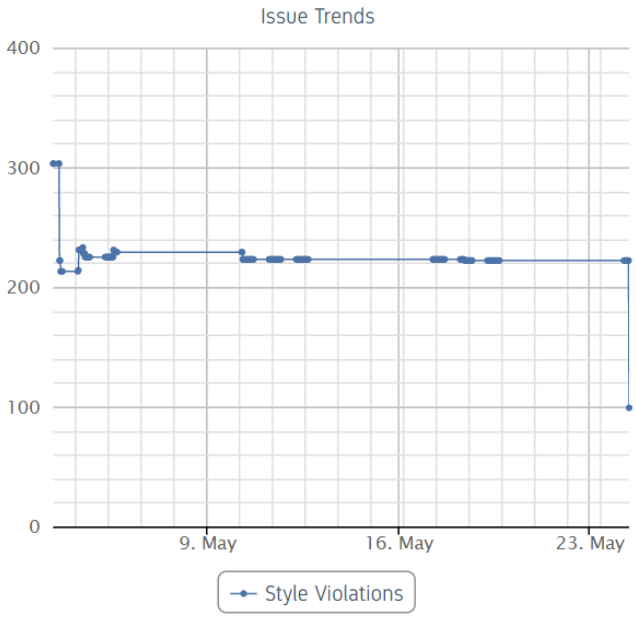


Figure 6.13: Misra C compliance.

6.7 Test. Gang vs FIFO, synchronization with busy-waiting

To prove the benefits of gang scheduling with busy waiting, I compared the gang scheduling algorithm with the POSIX FIFO scheduling algorithm. I implemented a test where the same workload is run first with Gang and then with FIFO. The test was developed to show that scheduling threads using Busy-Waiting for synchronization is better with a Gang policy than with a FIFO policy. In this case, the decision to use FIFO to demonstrate the difference depends on the implementation of the Gang algorithm, which is based on the FIFO one.

These tests were performed in the C language on a real-time operating system, PSE53. Both tests were run on a XILINX UltraScale+ architecture with four cores.
The tests generates the same number of threads, four of them (one for each core) have to synchronize with Busy-Waiting. An increasing number of threads, n, has to interfere. All these tasks have to execute for at least 100ms. Moreover, the four synchronizing tasks, after the CPU burst of 100ms, have to *wait* for all other Busy-Waiting tasks to finish.

The test was launched 5000 times for each n (task number); each time the order of execution of the tasks was different.

The following graph shows the results of the tests. The synchronization time (the time between the first and the last of the four tasks that had to synchronize) is always the same for the tasks scheduled in a gang and can vary greatly for the tasks executed with FIFO. In the graph, the best and worst times are given as confidence intervals for the tasks run with the FIFO policy.

As expected, the four tasks that run simultaneously with Gang Scheduling have a synchronization time of about 100ms, which is also the execution time for each task. Instead, FIFO has a best-case execution time of $100ms$ when the four tasks are activated sequentially, and a WCET of $n * 100 + 100$, i.e. the case where three of the busy-waiting threads start first and

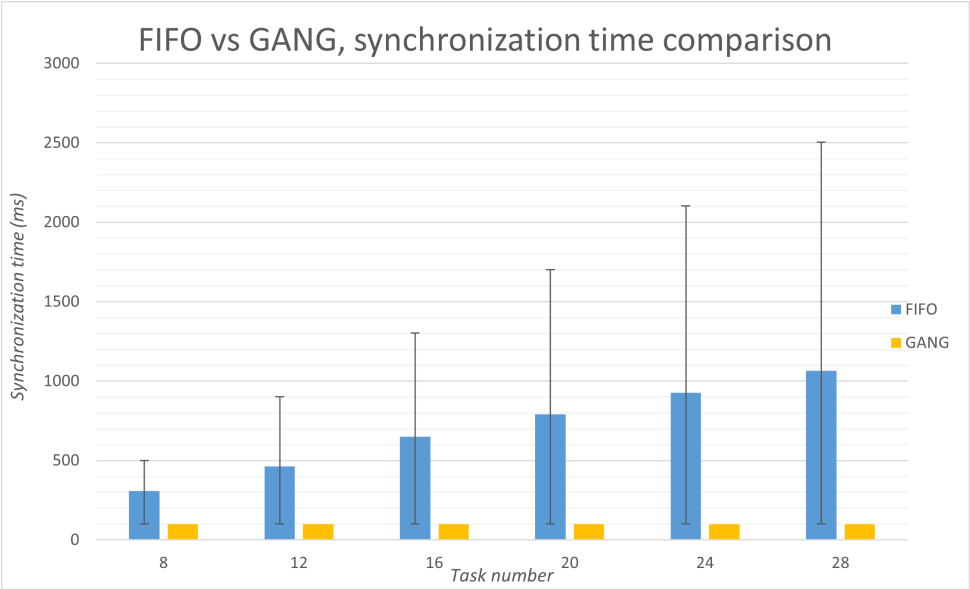
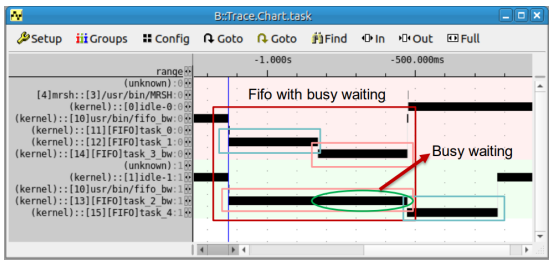
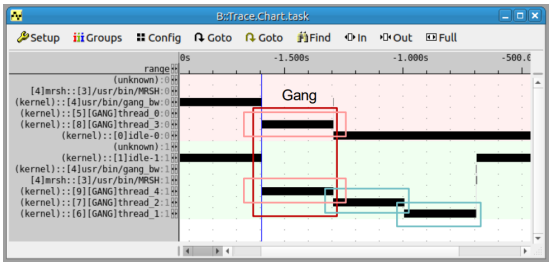


Figure 6.14: FIFO vs GANG

wait the fourth that runs last, after all the interfering tasks.



(a) Tasks scheduled with FIFO scheduling



(b) Tasks scheduled with GANG scheduling

Figure 6.15: Screenshots of Lauterbach TRACE32.

Figure 6.15 shows an example of the test run in a system with two cores. On the left, figure 6.15a, you can see a situation where one task busy waits until the end of the other task. On the right, figure 6.15b shows the same example with a gang scheduler. Here the tasks do not have to wait long because they are executed (and terminate) together.

7 Building

To integrate the scheduler and the tests in the operative system was used BitBake, a tool that is used to build software for embedded systems.

BitBake is a generic task execution engine that allows shell and Python tasks to be run efficiently and in parallel while working within complex inter-task dependency constraints. In short, BitBake is a build engine that works through recipes written in a specific format in order to perform sets of tasks. BitBake Recipes, which are denoted by the file extension .bb, are the most basic metadata files [4].

In addition to compiling source code, BitBake is also responsible for managing dependencies, retrieving source code from remote repositories, and creating and maintaining a complete build history. BitBake is used by a number of high-profile projects, including the Yocto Project,

OpenEmbedded, and OpenWrt. In fact, BitBake is a core component of the Yocto Project [25], an open source collaboration project that helps developers create custom Linux-based systems. It is also used by the OpenEmbedded build system to build images, another tool that allows developers to create a complete Linux Distribution for embedded systems [11].

BitBake was used in PSE53 to create recipes to describe how to build the software and to manage software dependencies.

(a) Scheduler files

(b) CUnit library and gang test files

Figure 7.1: BitBake repositories and files of the scheduler

Figure 7.1 shows the repository containing all the recipes, the files of the scheduler, the tests and the configuration files for BitBake.

8 Conclusions

Gang scheduling can be an effective way to improve the performance of a real-time operating system. By making use of multiple processors, gang scheduling can provide a more efficient way to schedule processes and can help to raise the overall performance of the system. In fact, gang scheduling helps to boost the responsiveness of the system by allowing processes to be executed in parallel. Other benefits of using the gang policy are that it can help to improve the predictability of the system and can provide a more flexible way to manage resources. Moreover, using busy waiting to synchronize gang tasks can minimize the number of context switches and can better the performance of the system. For these reasons, gang scheduling is a valid alternative to the standard Linux schedulers in a multicore real time system.

The scheduler presented in this thesis has been implemented on a novel real-time embedded

system, PSE53. It was based on the FIFO policy and the one presented by Waqar Ali and Heechul Yun in their research paper, RT-gang [2]. All the main behaviours and functions of the scheduler have been tested and the 93% of the code has been covered. Moreover, has been verified the Misra C compliance, and corrected the majority of the style violations.

To develop the scheduler has been used a tool to debug and trace the correct execution, Lauterbach TRACE32. Also, to build and integrate the scheduler in the operating system has been used BitBake.

In conclusion, in this thesis has been discussed the advantages of using the gang scheduler on a RTOS. Furthermore, it has been shown how it has been designed, tested and implemented for PSE53.

Bibliography

[1] Peter Baer Galvin Abraham Silberschatz and Greg Gagne. *Operating System Concepts*. Wiley, 10th edition, 2018.

[2] Waqar Ali and Heechul Yun. Rt-gang: Real-time gang scheduling framework for safety-critical systems. *IEEE Intl. Conference on Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, August, 2018.

[4] Bitbake manual. https://docs.yoctoproject.org/bitbake/1.46/bitbake-user-manual/bitbake-user-manual-intro.html. Retrieved 2022-06-17.

[5] Busy waiting. https://en.wikipedia.org/wiki/Busy_waiting. Retrieved 2022-06-11.

[6] Contributions to gang scheduling. https://www.tesisenred.net/bitstream/handle/10803/5968/07Jcg07de08.pdf. Retrieved 2022-06-17.

[7] Cunit. http://cunit.sourceforge.net/. Retrieved 2022-06-15.

[8] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, 1992.

[9] Free rtos. https://www.freertos.org/FAQ.html, https://www.freertos.org/about-RTOS.html. Retrieved 2022-05-05.

[10] Kernel definition. http://www.linfo.org/kernel.html. Retrieved 2022-05-05.

[11] Openembedded wiki. https://www.openembedded.org/wiki/Main_Page. Retrieved 2022-06-17.

[12] Operating system. https://en.wikipedia.org/wiki/Operating_system. Retrieved 2022-05-05.

[13] Posix. https://en.wikipedia.org/wiki/POSIX. Retrieved 2022-05-05.

[14] Posix austin. https://www.opengroup.org/austin/papers/posix_faq.html. Retrieved 2022-05-05.

[15] Posix standard. https://linuxhint.com/posix-standard/. Retrieved 2022-05-05.

[16] preemption. https://en.wikipedia.org/wiki/Preemption_(computing). Retrieved 2022-06-17.

[17] Process management. https://en.wikipedia.org/wiki/Process_management_(computing). Retrieved 2022-06-25.

[18] Real time computing. https://en.wikipedia.org/wiki/Real-time_computing. Retrieved 2022-05-05.

[19] Real time operating system. https://en.wikipedia.org/wiki/Real-time_operating_system. Retrieved 2022-06-15.

[20] Linux manual page. https://man7.org/linux/man-pages/man7/sched.7.html. Retrieved 2022-06-15.

[21] Linux manual page, sched yield. https://man7.org/linux/man-pages/man2/sched_yield.2.html. Retrieved 2022-06-17.

[22] Coscheduling. https://en.wikipedia.org/wiki/Coscheduling. Retrieved 2022-06-24.

[23] Thread. https://en.wikipedia.org/wiki/Thread_(computing). Retrieved 2022-06-15.

[24] Yield. https://en.wikipedia.org/wiki/Yield_(multithreading). Retrieved 2022-06-17.

[25] Yocto project manual. https://docs.yoctoproject.org/3.1.5/overview-manual/overview-manual-yp-intro.html. Retrieved 2022-06-17.