Tower Defense

5TH YEAR

REAL-TIME SYSTEMS

PROF. PAULO PEDREIRAS

UNIVERSIDADE DE AVEIRO

2016/2017

GROUP 2
DIOGO SILVA 60337
EDUARDO SOUSA 68633

Contents

1 Introduction

The proposed theme for this work was to develop a Tower Defense game, that was implemented as real-time system.
The Tower Defense game is composed of a path, where monsters have to find their way to the end. The objective of the game is to not allow the monsters to reach the end of the path. To do so a human player must place towers alongside the path, that will try to take out the monsters while they traverse it. The human player can place or remove towers, but it can't interfere with the behaviour of the towers. To make the game more interesting, difficulty will increase when the player surpasses a level. This increase in the difficulty was implemented by introducing a stronger type of monster.

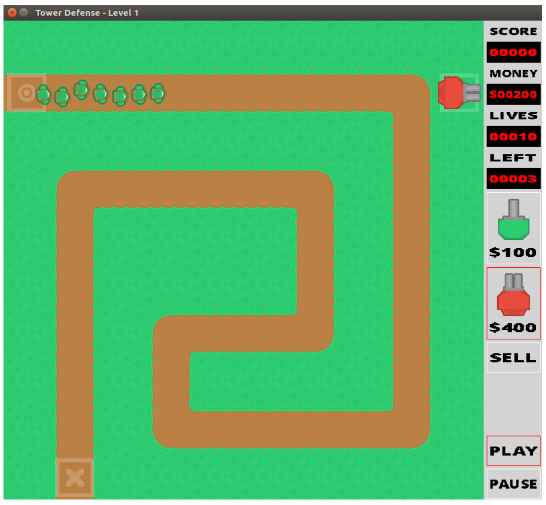

Figure 1: Tower Defense game final aspect.

2 Development tools

The game was developed using C++ and in order to develop the game the following development tools were used:

- **Xenomai** - In order to create the game engine, it was used the Xenomai framework, which is a real-time kernel, that is inserted as co-kernel in the Linux kernel. The Xenomai framework provides APIs that guarantee that the code developed will run in real-time mode and will be scheduled using real-time policies, which is required for the work developed. **Note:** The Xenomai version used was 2.6.5 and the linux kernel version was 3.18.20.
- **Simple DirectMedia Layer 2 (SDL 2)** - In order to create the game visual interface, it was used Simple DirectMedia Layer 2 (SDL 2), which is a cross-platform library that provides access to the graphics hardware via OpenGL and also allows access to computer human interface devices, such as keyboard or mouse.
- **Cereal** - The kernel space and user space need to communicate with each other using the real-time pipes. The library cereal is responsible for the marshalling and unmarshalling of the data. That way we can create communicate messages between the user space and kernel space without much effort (complexity is transparent for the user).
- **CMake** - This tool is responsible to build this project (there are several CMakeLists.txt across the project source code). CMake allows the source code to be compiled easier by any computer without much complicated set-ups and it is also possible to compile only the user space that does not require Xenomai.

3 Game Description

3.1 Objects

The game environment is made up of the following components:

- **Monster** - A monster is an object that will be created in the beginning of the path and will try to traverse the path until it reaches the end. There are two types of monsters: normal and super.
- **Tower** - A tower is an object that will be placed by the user alongside the path and will try to destroy the monsters. There are two types of towers: normal and super.
- **Bullet** - A bullet is an object that will be fired by a tower. A monster will lose health when it's hit by a bullet.
- **Path** - The path is the set of tiles in the map where a monster can walk on.
- **Field** - The field is the set of tiles in the map where the user can place towers.
- **Menu** - The menu is where the user can check statistics or select an option to interact with the game, such as:
  - insert a tower,
  - remove a tower,
  - play the game,
  - pause the game.

3.1.1 Monster

A monster is an object that will be created in the beginning of the path and will try to traverse the path until it reaches the end. There are two types of monsters: normal and super.
The monster has three eyes, that will act as the main sensors. The left eye is at 90 degrees, the middle one is at 0 degrees and the right one is at -90 degrees.
After the monster processes the information perceived by the eyes, it can execute two kinds of actions: move forward or rotate.

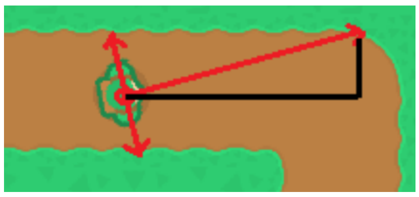
Figure 2: Monster's eyes placement.

3.1.2 Tower

The tower is an object that will be created by the user. The tower can be placed by the user alongside the path. It will try to destroy the monsters. There are two types of towers: normal and super.
The tower has radar, that will act as the main sensor. The radar is omnidirectional, meaning that it can detect a monster in any direction until a certain range.
After the tower processes the information obtained by the radar, it can take two actions, such as: rotate the cannon or shoot.
Note: the shooting range will always be smaller than the radar range.

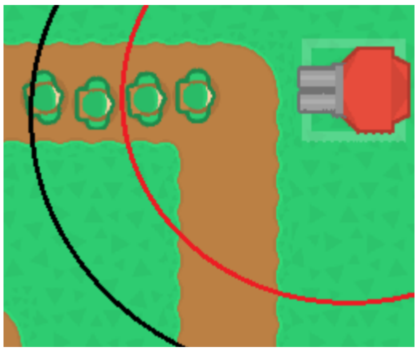

Figure 3: Radar and shooting range for the tower.

3.2 Currency Logic

The user starts with a given amount of money in the game. He will be able to use that money to place towers in the fields.
Every time that a tower destroys a monster, the user receives a specific amount of money (corresponding to the monster's difficulty).

3.3 Levels Logic

The level only finishes after the towers have destroyed every monster in the path.
To increase the difficulty of the game, when a user surpasses a level a super monster will replace a normal monster. A super monster is faster and has more health points.
To give the user a fighting chance, a user can insert a super tower, instead of a normal tower. A super tower will be faster rotating the cannon and it needs less time to cool down between shots.
The user has 10 lives and he loses a life for each monster that reaches the end of the path.

4 Software Architecture

Tasks might be several times mentioned in this section. Take into account that there are 4 tasks: God (responsible for managing the engine) and 3 for the entities (User interaction, Towers and Monsters). If you want to take a look at more detail about those tasks, feel free to go to the next section (section 5).

The game is divided in two modules:

- **Game Engine** - It runs in the kernel space and it is responsible by running all tasks in real-time mode and sending all the world information for the game viewer.
- **Game Viewer** - It runs in the user space and it is responsible for rendering a visual representation of the simulated world.

4.1 Game Engine

The Game Engine creates and manages all the tasks, it also has the responsibility of sending the world state data to the game viewer.
The Game Engine manages the order of task using a request issuing system. If a task wants to execute an action, that task will issue a request to the god task. Every task issues a request per game cycle. This ensures that all the tasks only can issue a request per game cycle.

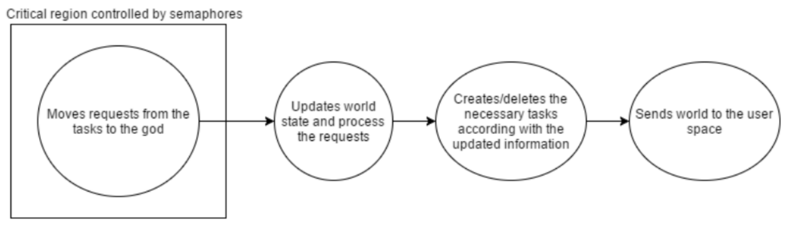
Critical region controlled by semaphores
Moves requests from the tasks to the god
Updates world state and process the requests
Creates/deletes the necessary tasks according with the updated information
Sends world to the user space

Figure 4: Request issuing system used.

Each type of task has a dedicated interface, specifically designed for the purpose of the task type to issue a request for the god task. In the next subsections these interfaces will be discussed.

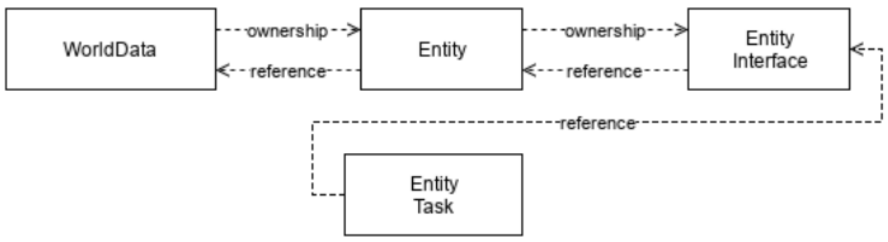

Figure 5: Entity interface.

This kind of logic makes it impossible for the function with a reference for the interface to have any access to the World data. This way the environment is controlled, the towers and monsters cannot get world data information without using the functions from the interface.

4.1.1 Monster Interface

A monster has sensors and actuators and those are the functions that the interface provides.

1. eyes() - The monster task will access this function to retrieve information of the monster's eyes. This function will make the task block while waiting for the information (busy waiting). Also, the information retrieved will have noise.
2. move(Movement) - The monster task will issue a request to move using this function. The request is going to be received and processed by the god task.
3. rotate(Rotation) - The monster task will issue a request to rotate using this function. The request is going to be received and processed by the god task.
4. get_idenitifer() - There are several monsters, so it needs an identifier. This is required by the god task to delete a task faster without have to go through a list comparing every element with the object.

4.1.2 Tower Interface

A tower has sensors and actuators and those are the functions that the interface provides.

1. radar() - The tower task will access this function to retrieve information of the tower's radar. This function will make the task block while waiting for the information (busy waiting). Also, the information retrieved will have noise.
2. shoot() - The tower task will issue a request to shoot using this function. The request is going to be received and processed by the god task.
3. rotate(Rotation) - The tower task will issue a request to rotate the cannon using this function. The request is going to be received and processed by the god task.

4. get_idenitifer() - There are several towers, so it needs an identifier. This is required by the god task to delete a task faster without have to go through a list comparing every element with the object.

The tower also has some access to several attributes about itself, like his current angle and position. Neither of those attributes are critical and do not allow the tower agent to cheat the game with it.

4.1.3 User Interaction Interface

There is only one interface, so it does not require an identifier. And it has functions to add/remove towers and pause/play the game.

1. add_tower(Type, Position) - The user interaction task will issue a request to add a tower to the map using this function. The request is going to be received and processed by the god task.
2. remove_tower(Position) - The user interaction task will issue a request to remove a tower from the map using this function. The request is going to be received and processed by the god task.
3. modify_game_status(GameStatus) - The user interaction task will issue a request to play or pause the game using this function. The request is going to be received and processed by the god task.

4.2 Game Viewer

The game viewer is the module responsible by rendering the state of the world in the screen. The information is sent by the game engine.
It runs in the user space, which makes it impossible to access directly to the World data which is in the kernel space. It uses pipes, in this case, 2 pipes. One pipe to receive the information from the game engine and another to send the user interaction to the game engine.

4.3 Real-time Pipes

The kernel space (game engine) and user space (game viewer) communicate with each other using the real-time pipes.
There are 2 pipes:

1. Receiver - The first pipe receives the world data necessary to represent the graphical interface and nothing more. For example, towers and monsters position, scores, the path.
2. Sender - It sends requests that are created based on the user interaction. The user is able to press pause, play, place towers in the fields or even remove towers.

To communicate those structures it uses a library cereal that is responsible for the marshalling and unmarshalling of the data. That way it is possible to create messages between the user space and kernel space without much effort (complexity is transparent for the user).

4.3.1 Viewer Data Buffering

The Game Viewer is responsible for rendering the information obtained from the game engine. Since the viewer might be slower rendering the images than the game viewer sending the information, it will have to store the latest information received from the game engine. The strategy used is called double buffering and to do this the game viewer will store two frames: the frame being rendered and the most up to date frame.

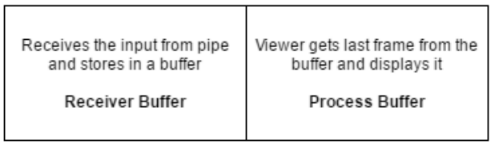
Receives the input from pipe and stores in a buffer
Receiver Buffer
Viewer gets last frame from the buffer and displays it
Process Buffer

Figure 6: Double buffering.

5 Tasks

Since the game was implemented as real-time system, there was the need to fulfill real-time requirements. To fulfill this requirements, the game architecture had to be thought out with the need to use Xenomai, since it provides the capabilities to work in the real-time mode.
A game usually works with a main cycle that checks the player input, processes game events (like moving objects) and then renders the new frame.
However, Xenomai works with tasks, which can be periodic, aperiodic or sporadic. Therefore, we had to adapt this architecture to use the tasks in a meaningful way according to their characteristics:

- **Periodic tasks** - Tasks are activated periodically according to the time period.
- **Aperiodic tasks** - The times are not known, so we cannot predict their behaviour. The only values we can work with are statistics.
- **Sporadic tasks** - Tasks are activated when events occur. It has a minimum interval time (MIT) for new occurrence which means that a new activation occurs after at least MIT. Those kind of tasks are a subgroup of aperiodic tasks.

Periodic tasks:

- **God** - It is responsible for processing all the requests from the other tasks, ensuring that the laws of the simulated world are not violated, creation and removal of tasks and sending the world state information to the viewer.
- **Tower** - It is responsible for checking the information that its radar sees, processing that information and then issue a request to rotate the cannon and shoot. Each tower has its own task.
- **Monster** - It is responsible for checking the information that it receives from the eyes, processing that information and then issue a request to move and rotate. Each monster has its own task.

Sporadic tasks:

- **User interaction** - It is responsible for receiving the user interaction from the viewer and issue a request to the god task to apply that interaction.

5.1 God task

The God task is a periodic task, that runs with 25ms period, to set a 40 frames-per-second standard.
God task is responsible to update the world state, which means to update the pose of each monster and tower, the position of each bullet, check for collisions between bullets and monsters. Any of these actions is validated by the God, it means that a monster is not allowed to exit the path, or move/rotate faster than it is allowed, same for the towers. The user requests are also checked, god validates where the user wants to place a tower and if it has money for that.
He process the requests of the user, the monsters and the towers. For every request that it processes it checks if that if that request is valid and does not violate any of the simulated world laws.
When there is a need to create new monsters or new towers in the simulated world, the god task is responsible by the creation of the object that holds the data and the task that will be its life cycle. When those objects are no longer needed or are destroyed, it's the duty of the god task to eliminate both the object that holds the data, the task associated with it and the semaphore specific for that task.
After all this is done, the last responsibility of the god task is to collect all the world state information and send it to the game viewer through a real-time pipe provided by Xenomai's API. Since the period of the task is set to 25ms, the game will render at 40 frames-per-second, which will create the illusion of movement in the eyes of the human player.
The God task priority is 90 (in a scale from 0 to 99 inclusive) and it's that task with the highest priority.

5.2 User interaction task

The User Interaction task is a sporadic task that is responsible for receiving the user interactions (clicks in the graphical interface created by SDL2) from the game viewer and issuing requests to god task with those interactions, in order for the god task to apply them.
It has a priority of 80 (in a scale from 0 to 99 inclusive). It's the second highest priority. The reason for this is that it shouldn't interfere with the god task.

5.3 Tower task

The Tower task is a periodic task that is responsible for executing the life cycle of a tower. The life cycle of the tower is the following:

- **Sensing -** Check the information that the radar can see (busy waiting).
- **Processing -** Process the information obtained from the radar.
- **Actuating -** Issue a request to rotate the cannon tower and shoot.

During the sensing phase, while the task activates the tower's radar it will have to be doing a busy wait, so it won't yield the use of the processor. This was implemented this way to mimic real world usage of a radar, in a manner that the person operating the radar will take a certain time processing all the information that the system is producing.
After the information has been retrieved from the radar, the tower task will process all that information to define what action it should take next.
When a decision is made, the tower task will issue a request to the god task to perform a movement such as rotate the cannon or shoot. This is done this way in order to check if a certain action won't violate the laws of the simulated world.

5.4 Monster task

The Monster task is a periodic task that is responsible for executing the life cycle of a monster. The life cycle of the monster is the following:

- **Sensing -** Check the information that the eyes can see (busy waiting).
- **Processing -** Process the information obtained from the eyes.
- **Actuating -** Issue a request to move in front and rotate.

During the sensing phase, while the task checks the monster's eyes it will have to be doing a busy wait, so it won't yield the use of the processor. This was implemented this way to mimic real world processing of information by a human being, in a manner that the human being processing the information will take a certain time processing all the information that the eyes are producing.
After the information from the eyes has been checked, the monster task will process all that information to define what action it should take next.
When a decision is made, the monster task will issue a request to the god task to perform a movement such as moving forward or rotate. This is done this way in order to check if a certain action won't violate the laws of the simulated world.

6 Critical Region

Every entity has a request vector that is copied and cleared by the God task, which means that it is a critical region.
When an entity wants to write a new request, it must enter in the critical region by requesting the resource using the semaphore. After using it, it has to release it. When the god wants to get the newest requests, it

requests the resource of every entity, copies the data from the vectors, clear the vectors and releases the resources.
Every tower or monster that is created has a semaphore associated.
Every entity (monster, tower and user interface) will only share resources with god task. Each means that **every entity has a semaphore** for themselves and to share with the god task.

7 Proof of concept

7.1 System's evaluation

This simulation has only tasks with **soft time constraints**, so the system is considered a **soft real-time system**. After a specific temporal limit, the quality of the system would degrade.
The system is **preemptive**, which means that tasks can be interrupted by other tasks with higher priority. Although there is some situations where the tasks **share resources**, so the system also contains **semaphores to grant exclusive access** to those resources.
The scheduling algorithm is **static** because the priorities of tasks do not change through the execution of program. It is also **online**, because the decisions are made during the execution, the algorithm could not know when the tasks would be created, started or even finish.

7.2 Measured times

Since the **Xenomai 2.6.5 uses Priority Inheritance Protocol when the system shares resources**, we had to take into consideration the worst execution times and maximum blocking times.

Most of the time we were able to find out which one was the worst case by code analysis. Some cases had more comparisons than others, we also did assume that a push back into a vector is more expensive than a comparison of a primitive variable, which helps a lot to define the worst case in the Towers and Monsters since they push back to a vector when they have requests to do.

In the case of the **God task**, it was harder. The **code analysis is harder** to do and since we are in a **soft real-time system situation**, we decided to let the system work for some time with the god task and check how the god task times vary. Then we just **set up a confidence interval** over it.

Every value in here was measured with background jobs in the system, in this case, compressing a big file to check how it could influence the tasks performance.

7.2.1 Worst situations

Tower task - Each tower has a radar associated and it will create a vector with the monsters that are in range for the tower. The action that the tower takes also has some different situations, but the one that takes more conditionals checks is when the first monster in the vector is at the left of the tower and the tower wants to rotate and shoot. So, the worst situation for the tower task happens when the **radar is able to reach all monsters, the tower has to rotate to the left and wants to shoot.**

Worst time measured: 350 us

Monster task - The sensors has no impact in these situations, they always have the same cost, the perception that the monster takes is what might have different times. Based in code analysis, the worst situation happens when the **monster detects for the first time a wall in the front and has to rotate to the left**, the actuation takes one additional condition when the robot wants to turn to the left because the right is checked first.

Worst time measured: 266 us

UserInteraction task - The code analysis for this task is simple. The worst situation occurs when the **task receives a message in the pipe (with a request from the graphical interface) and it is requested to add or remove a tower**. Those 2 actions take exactly the same cost because they insert an element in a vector (a request vector that will be checked later by the God task).

Worst time measured: 82 us

God task - This task is hard to predict the worst situation based in code analysis, so we let the system running with the god task alone varying the number of towers and monsters in the game. Basically, we tried to create a maximum number of distinct situations and then we set a confidence interval over it since it is a soft real-time system. The same is not acceptable in a hard real-time system!
Visualising the histogram of the executions times, we can verify the following:

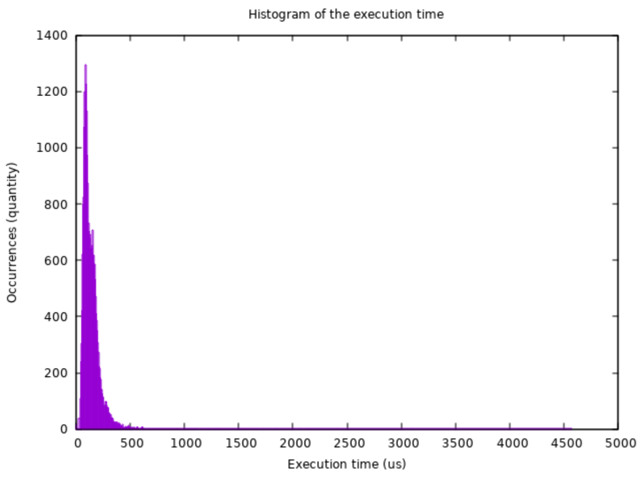

Figure 7: Histogram of God's execution times

The mean of the data is 149.2876.
The standard deviation is 109.7795.
Total values taken are 25787.
With a confidence level of 99%, the interval would be 146.6 to 151.9. But we obtained several values over that interval. The worst time obtained was way to far from the confidence interval and since it is a **soft** real-time system, we are going to consider that values as the worst case since it is a way from the confidence interval.

Worst time measured: 4577 us

7.2.2 Maximum time using the resource

Every tower, monster and user interaction has a specific resource to share between themselves and the god task. E.g. UserInteraction can preempt to store a request from the user when the tower is storing a request, its not the same resource.

Tower task - When the **tower is rotating to the left and wants to shoot** is the worst case scenario when using the resource based in code analysis, it has an extra condition when rotating for the left.

Maximum time using the resource: 36 us

Monster task - When the **monster is rotating to the left and wants to move in front** is the worst case scenario when using the resource based in code analysis, it has an extra condition when rotating for the left.

Maximum time using the resource: 41 us

User Interaction task - The user interaction had **3 distinct situations and hard to conclude anything with code analysis because they include push_back and unique_ptr**, so we measured time for all of them and took the biggest one.

Maximum time when modifying game status: 7 us
Maximum time when adding a tower: 2 us
Maximum time when removing a tower: 3 us
Maximum time using the resource from the 3 situations: 7 us

God task - This task uses all existing resources in the system at the same time, so it has to request for all the semaphores to proceed. This time will be bigger when **there is the number maximum of towers and monsters in the game**, because it has to clear the request in each one of them.

Maximum time using the resource: 746 us

7.3 Measures summary

Since the towers and monsters are created dynamically and there is a specific resource for each one of them, there will be a maximum total of 21 shared resources, 10 for monsters, 10 for towers and 1 for the user interaction. Those resources are shared with god as shown in the table 1 below.

Table 1: Maximum resource usage by each tasks in microseconds (us)

	Resource Tower	**Resource Monster**	**Resource User Interaction**
God Task (P: 90)	746	746	746
User Interaction (P: 80)	0	0	7
Monster (P: 70)	0	41	0
Tower (P: 70)	36	0	0

Table is simplified for 1 tower and 1 monster, although it might have a maximum of 10 towers and 10 towers which is important to keep in mind.

The maximum blocking time can be extracted from the previous table by taking into consideration that Xenomai 2.6.5 uses **Priority Inheritance Protocol when sharing resources**.
So a **task can block any other task just once** and **each task can block once in each resource**.

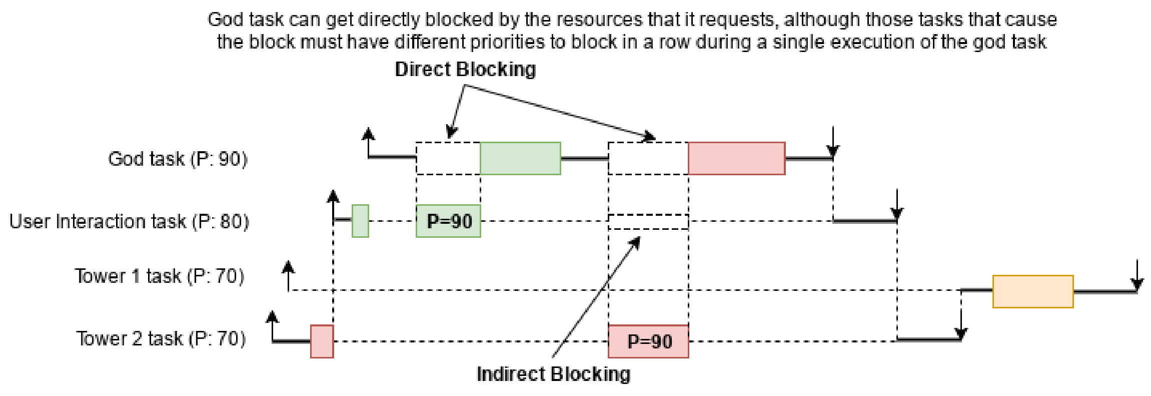
God task can get directly blocked by the resources that it requests, although those tasks that cause the block must have different priorities to block in a row during a single execution of the god task
Direct Blocking
God task (P: 90)
User Interaction task (P: 80)
P=90
Tower 1 task (P: 70)
Tower 2 task (P: 70)
P=90
Indirect Blocking

Figure 8: One of the possible example where the God task gets blocked in resources twice

The image 8 shows an example where the god gets blocked twice during the same execution, it cannot get blocked more than twice because Towers and Monsters share the same priority (70), only User Interface has a different one. So it might get blocked once by Tower/Monster and another one by the User

Interface, no more than that, that is the maximum blocking time for the god.

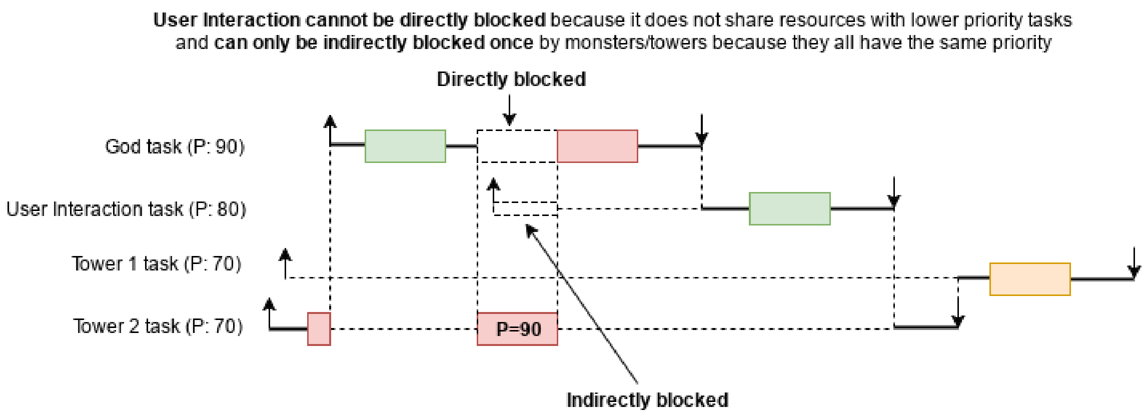
User Interaction cannot be directly blocked because it does not share resources with lower priority tasks
and **can only be indirectly blocked once** by monsters/towers because they all have the same priority
Directly blocked
God task (P: 90)
User Interaction task (P: 80)
Tower 1 task (P: 70)
Tower 2 task (P: 70)
P=90
Indirectly blocked

Figure 9: One of the possible example where the User Interaction task gets indirectly blocked

The image 9 shows an example where the user interaction gets blocked, and it is the worst blocking time since it cannot get directly blocked because it only shares resources with higher priority task. Although it can get indirectly blocked, but only once, because the lower priority tasks have the same priority, 70.

Towers and Monsters tasks cannot get blocked because they already have the lowest priority, they will be only preempted.

The User Interaction task is a **sporadic task**, there is no period. Instead, there is a minimal inter-arrival time which is equal to the God task period and we assume that the user will not be able to click more than 40 times each second (frequency), otherwise the extra requests will be lost, the god only processes every 25 ms. In the worst case, a **sporadic task behaves as a periodic task with period equal to the minimal inter-arrival time**, which is 25 ms.

Taking all the previous notes into account, the following table 2 was built (using the resource usage table also). God blocking time is the biggest blocking from the priority of 70 and the biggest priority of the priority of 80 (only one) like explained before, 41 + 7 us.

Table 2: Measured times for each task type

	Ti	**MIT**	**Ci**	**Bi**
God Task (P: 90)	25	-	4577	48
User Interaction (P: 80)	-	25	82	41
Monster (P: 70)	50	-	266	0
Tower (P: 70)	50	-	350	0

7.4 Schedulability Analysis

Since we are running Xenomai 2.6.5 with semaphores which has implemented PIP, we had to consider the following formula for the utilisation test:

$$\forall_{1\leq i\leq n} \sum_{h:Ph>Pi} \frac{Ch}{Th} + \frac{Ci+Bi}{Ti} \leq i(2^{1/i}-1) \tag{1}$$

The maximum number of tasks, n, in our system is always changing because the number of monsters and towers may decrease or increase. In the worst situation, the maximum number of towers will be 10 and the monsters will also be 10. So, the total number of tasks will be 22.

$i = 1 \rightarrow \frac{4577+48}{25000} \leq 1(2^{1/1} - 1) \Leftrightarrow 0.185 \leq 1$ ✓(God task)
$i = 2 \rightarrow \frac{4577}{25000} + \frac{82+41}{25000} \leq 2(2^{1/2} - 1) \Leftrightarrow 0.188 \leq 0.8284$ ✓(User Interaction task)

After this point, the higher priority tasks will be only God and User Interaction (others have priority 70).

$i = 3 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 3(2^{1/3} - 1) \Leftrightarrow 0.19336 \leq 0.77976$ ✓(Tower 1 task)
$i = 4 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 4(2^{1/4} - 1) \Leftrightarrow 0.19168 \leq 0.75682$ ✓(Monster 1 task)

Skipping to the last element (**The rest of the verifications are on the appendix**). The indexes shown in the appendix have the same results since the higher priority tasks do not vary. **Only the bound gets lower.** The bound will never go lower than $log(2)$ which is the limit when i goes to infinity and our worst results are 0.19336. Anyway, taking into consideration the last situation.

The last task might be a monster or a tower
$i = 22 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 22(2^{1/22} - 1) \Leftrightarrow 0.19168 \leq 0.70418$ ✓(Monster 10 task)
$i = 22 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 22(2^{1/22} - 1) \Leftrightarrow 0.19336 \leq 0.70418$ ✓(Tower 10 task)

The system is **schedulable**.
The margin that the system has is really big. The system could probably increase the maximum number of monsters and towers without losing performance, although it would require new measures for the worst case execution time and blocking time because they are affected by the total number of entities in the fields in several cases.

8 Conclusion

This application is a real-time system where the times defined for periods and MIT's are respected.

The users will be able to interact with the system without suffering loss of performance, they will be able to see the changes as soon as they occur (in this case, 40 Hz which is the time that the God task takes to update the world state)-

Developing this system allowed us to learn to use periodic and sporadic tasks in a real-time framework (Xenomai) and how the definition of periods and priorities modify the system's behaviour.

It also allowed us to better understand how the protocol used for the exclusive access works and how to verify that a system is schedulable or not.

A Schedulability Validations - Complete calculations

This appendix includes all the calculations that were made and were skipped because they were easy to visualize.

$$\forall_{1\leq i\leq n} \sum_{h:Ph>Pi} \frac{Ch}{Th} + \frac{Ci+Bi}{Ti} \leq i(2^{1/i}-1)$$

$i=1 \rightarrow \frac{4577+48}{25000} \leq 1(2^{1/1}-1) \Leftrightarrow 0.185 \leq 1$ ✓(God task)
$i=2 \rightarrow \frac{4577}{25000} + \frac{82+41}{25000} \leq 2(2^{1/2}-1) \Leftrightarrow 0.188 \leq 0.8284$ ✓(User Interaction task)

After this point, the higher priority tasks will be only God and User Interaction (others have priority 70).

$i=3 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 3(2^{1/3}-1) \Leftrightarrow 0.19336 \leq 0.77976$ ✓(Tower 1 task)
$i=4 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 4(2^{1/4}-1) \Leftrightarrow 0.19168 \leq 0.75682$ ✓(Monster 1 task)
$i=5 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 5(2^{1/5}-1) \Leftrightarrow 0.19336 \leq 0.74349$ ✓(Tower 2 task)
$i=6 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 6(2^{1/6}-1) \Leftrightarrow 0.19168 \leq 0.73477$ ✓(Monster 2 task)
$i=7 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 7(2^{1/7}-1) \Leftrightarrow 0.19336 \leq 0.72863$ ✓(Tower 3 task)
$i=8 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 8(2^{1/8}-1) \Leftrightarrow 0.19168 \leq 0.72406$ ✓(Monster 3 task)
$i=9 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 9(2^{1/9}-1) \Leftrightarrow 0.19336 \leq 0.72053$ ✓(Tower 4 task)
$i=10 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 10(2^{1/10}-1) \Leftrightarrow 0.19168 \leq 0.71773$ ✓(Monster 4 task)
$i=11 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 11(2^{1/11}-1) \Leftrightarrow 0.19336 \leq 0.71545$ ✓(Tower 5 task)
$i=12 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 12(2^{1/12}-1) \Leftrightarrow 0.19168 \leq 0.71356$ ✓(Monster 5 task)
$i=13 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 13(2^{1/13}-1) \Leftrightarrow 0.19336 \leq 0.71196$ ✓(Tower 6 task)
$i=14 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 14(2^{1/14}-1) \Leftrightarrow 0.19168 \leq 0.71059$ ✓(Monster 6 task)
$i=15 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 15(2^{1/15}-1) \Leftrightarrow 0.19336 \leq 0.70941$ ✓(Tower 7 task)
$i=16 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 16(2^{1/16}-1) \Leftrightarrow 0.19168 \leq 0.70838$ ✓(Monster 7 task)
$i=17 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 17(2^{1/17}-1) \Leftrightarrow 0.19336 \leq 0.70747$ ✓(Tower 8 task)
$i=18 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 18(2^{1/18}-1) \Leftrightarrow 0.19168 \leq 0.70667$ ✓(Monster 8 task)
$i=19 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 19(2^{1/19}-1) \Leftrightarrow 0.19336 \leq 0.70595$ ✓(Tower 9 task)
$i=20 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 20(2^{1/20}-1) \Leftrightarrow 0.19168 \leq 0.70530$ ✓(Monster 9 task)
$i=21 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{266}{50000} \leq 21(2^{1/21}-1) \Leftrightarrow 0.19168 \leq 0.70471$ ✓(Monster 10 task)
$i=22 \rightarrow \frac{4577}{25000} + \frac{82}{25000} + \frac{350}{50000} \leq 22(2^{1/22}-1) \Leftrightarrow 0.19336 \leq 0.70418$ ✓(Tower 10 task)

$$\forall_{1\leq i\leq n} \sum_{h:Ph>Pi} \frac{Ch}{Th} + \frac{Ci+Bi}{Ti} \leq i(2^{1/i}-1) \checkmark$$