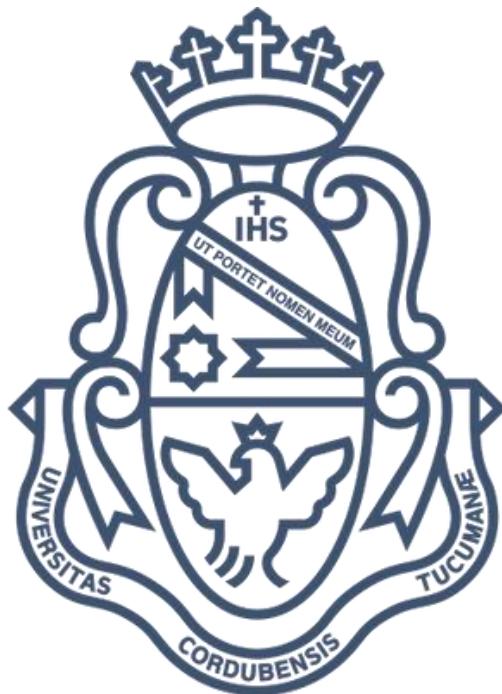


UNIVERSIDAD NACIONAL DE CÓRDOBA
FACULTAD DE CIENCIAS EXACTAS FÍSICAS Y NATURALES
INGENIERÍA EN COMPUTACIÓN



PROYECTO INTEGRADOR

**DESARROLLO DE UN DEVICE DRIVER DE LINUX Y SU LIBRERÍA DE ESPACIO USUARIO
PARA GESTIONAR REDES DE PETRI GENERALIZADAS EN EL KERNEL**

Autor

Sosa Ludueña Gabriel
36.356.433
gsosaludu@gmail.com
03548 – 15500666

Director

Doctor Micolini Orlando
omicolini@compuar.com

Resumen

Este trabajo presenta un enfoque diferente para la gestión de Redes de Petri Generalizadas (RDPG) desde el espacio usuario. Se propone el uso de múltiples hilos en programas de espacio usuario, que interaccionan con un Device Driver de Linux denominado MatrixmodG. El driver cuenta con soporte al multiprocesamiento simétrico seguro (SMPs) en el kernel, permite que la gestión de RDPG sea independiente del lenguaje de programación a utilizar en el espacio usuario y provee mejoras de rendimiento al gestionar la RDPG desde el kernel de Linux. El driver se comporta como una nueva funcionalidad del sistema operativo Linux, brindando disponibilidad a cualquier programa y/o aplicación de usuario. Para cumplir con el enfoque propuesto se desarrolló una librería de C++ que permite la comunicación de subprocesos con el driver en las gestiones de la RDPG cargada en el kernel, a través de llamadas al sistema, llevando de esta forma todo su rendimiento al espacio usuario. Se abordan dos casos de aplicación, que hacen uso de múltiples hilos, demostrando las ventajas de gestionar RDPG con el driver a comparación de una aplicación de usuario de características similares.

Agradecimientos

Por estar siempre presentes, su confianza infinita y enseñanzas en todo momento que permiten mi crecimiento personal y profesional.

A mis padres y hermanos, Susana, Marcelo, Andres y Diego.

Por permitirme participar y descubrir el proyecto, su confianza, paciencia y todo su acompañamiento.

A mi director de proyecto Dr. Orlando Micolini.

Por su compañía, sus ganas de compartir y trabajar en equipo y por aportar siempre su granito de arena.

A mis compañeros de facultad.

Por sus diferentes aportes.

Marcelo Cebollada, Luis Ventre, Maximiliano Eschoyez, miembros y ex miembros del LAC

Por mostrar compromiso y responsabilidad, por toda la enseñanza de calidad brindada, por el espacio de estudio y por su espíritu de crecimiento.

A mi facultad
FCEFyN – UNC.

ÍNDICE

Índice de figuras	vi
Índice de tablas	viii

1. Introducción.....	1
1.1. Descripción General	1
1.2. Objetivos	2
1.2.1. Objetivo general	2
1.2.2. Objetivos específicos	2
1.3. Metodología de trabajo.....	2
1.4. Definiciones, acrónimos y abreviaturas	3
2. Marco Teórico.....	5
2.1. Redes de Petri ordinarias	5
2.1.1. Definición de una Red de Petri Ordinaria	5
2.1.2. Estructura matemática de una RDP	6
2.1.3. Dinámica de una RDP	7
2.2. Redes de Petri Generalizadas.....	7
2.2.1. Fundamentos.....	8
2.2.2. Extensión de la ecuación de estado.....	9
2.2.3. Ecuación de estado extendida	10
2.3. Kernel de Linux	12
2.3.1. Funciones del kernel	13
2.3.2. Arquitectura del kernel	14
2.3.3. Espacio kernel y espacio usuario	15
2.3.4. El rol de las llamadas al sistema	15
2.3.5. El rol de las interrupciones	16
2.3.6. Kernel de Linux versus Kernel Unix	17
2.4. Llamadas al sistema.....	17
2.4.1. Comunicación con el kernel.....	17
2.4.2. APIs, POSIX y la biblioteca C	18
2.4.3. Controlador de llamadas al sistema	19
2.5. Device Drivers de Linux.....	19
2.5.1. Introducción.....	19
2.5.2. Módulos del kernel versus aplicaciones.....	22
2.5.3. Compilación y ejecución de módulos	23

2.5.4. Carga y descarga de módulos	26
2.5.5. Símbolos del kernel	27
2.5.6. Preliminares	28
2.5.7. Inicialización y limpieza	29
2.5.8. Char device driver.....	30
2.6. Asignación de memoria en el Kernel de Linux	36
2.6.1. La familia de funciones Kmalloc	36
2.6.2. La familia de funciones Vmalloc	41
2.6.3. Buenas prácticas de asignación de memoria	43
2.6.4. Kernel Memory Leaks	43
2.7. Sincronización en el Kernel de Linux	44
2.7.1. Regiones críticas y condiciones de carrera	44
2.7.2. Bloqueo	45
2.7.3. Interbloqueo.....	48
2.7.4. Contención y Escalabilidad.....	49
2.7.5. Spin Locks.....	50
2.7.6. Spin Locks Lector-Escritor	52
2.8. Patrones de diseño en el kernel de Linux	54
2.8.1. Paso de métodos	55
2.8.2. Archivos de cabecera	57
2.8.3. Estructuras opacas	58

3. Estudio del sistema	61
3.1. Especificación de Requerimientos	61
3.1.1. Ámbito del sistema	61
3.1.2. Características de los usuarios	62
3.1.3. Diagramas de casos de uso	62
3.1.4. Requerimientos funcionales.....	69
3.1.5. Requerimientos no funcionales.....	72
3.1.6. Interfaces externas	74
3.1.7. Atributos del sistema	75
3.2. Plan de gestión de riesgos	75
3.2.1. Identificación de riesgos	75
3.2.2. Análisis de riesgos	77
3.2.3. Planificación de estrategias de riesgos	78
3.3. Trazabilidad de Requerimientos	79
3.3.1. Matrices de trazabilidad.....	80

3.4.	Análisis de uso de bytes en matrices y vectores.....	83
4.	Diseño e implementación	85
4.1.	Iteraciones	85
4.1.1.	Iteración 0.....	85
4.1.2.	Iteración 1	85
4.1.3.	Iteración 2.....	86
4.1.4.	Iteración 3.....	87
4.1.5.	Iteración 4.....	88
4.1.6.	Iteración 5.....	89
4.2.	Composición del sistema final	89
4.2.1.	Comandos de driver MatrixmodG	89
4.2.2.	Objetos de driver MatrixmodG	94
4.2.3.	Característica SMPs de driver MatrixmodG	99
4.2.4.	Macros de mensajes de debug.....	100
4.2.5.	Disparo de transición	101
4.3.	Arquitectura del sistema	103
4.3.1.	Fundamentos.....	103
4.3.2.	Arquitectura de alto nivel	104
4.4.	Librería LibMatrixmodG de espacio usuario	112
4.5.	Disponibilidad a multiples lenguajes	113
5.	Plan de pruebas.....	115
5.1.	Introducción.....	115
5.1.1.	Objetivo del plan de pruebas.....	115
5.1.2.	Alcance de pruebas	115
5.1.3.	Estrategia del plan de pruebas.....	116
5.2.	Características de pruebas.....	116
5.2.1.	Características a ser probadas	116
5.2.2.	Características a no ser probadas	117
5.3.	Entornos de pruebas.....	117
5.3.1.	Técnicas de depuración en el kernel	118
5.3.2.	Herramientas de pruebas unitarias e integrales	119
5.3.3.	Herramienta de pruebas de sistema.....	121
5.4.	Procedimiento de trabajo sobre las pruebas	123
5.4.1.	Especificación de caso de prueba.....	123
5.4.2.	Ejecución del caso de prueba.....	125

5.4.3.	Análisis y registro de resultados	125
5.4.4.	Gestión y reporte de defectos.....	126
5.5.	Cobertura de pruebas	126
5.5.1.	Trazabilidad de Pruebas.....	127
6.	Resultados	129
6.1.	Modo de uso de objetos en driver MatrixmodG.....	130
6.1.1.	Construcción y creación de objetos	130
6.1.2.	Creación de objetos RDPG	131
6.1.3.	Obtener información de objetos RDPG	133
6.1.4.	Disparo de transición de objetos RDPG.....	135
6.1.5.	Seteo de parámetros en objetos RDPG	135
6.1.6.	Protección SMPs de objetos RDPG	136
6.2.	Escritura y lectura de driver MatrixmodG	137
6.3.	Modo de uso de librería de espacio usuario	138
6.3.1.	Instalación de driver MatrixmodG	138
6.3.2.	Gestión de objetos RDPG con librería	139
6.4.	Caso de aplicación 1	142
6.4.1.	Estudio de RDP de procesador monocore.....	142
6.4.2.	Conversión de RDP a RDPG	144
6.4.3.	Modificaciones de RDPG	146
6.4.4.	Extensión de RDPG a procesador dualcore	147
6.4.5.	Política de RDPG de procesador dualcore	149
6.4.6.	Componentes base de la RDPG	150
6.4.7.	Modo de ejecución de la RDPG.....	153
6.5.	Caso de aplicación 2	155
6.5.1.	Estudio de RDP Productor-Consumidor	156
6.5.2.	Componentes base de la RDP	156
6.5.3.	Modo de ejecución de la RDP.....	157
6.6.	Rendimiento del sistema.....	157
6.6.1.	Tiempos de asignación de memoria dinámica	158
6.6.2.	Tiempos de operaciones Driver MatrixmodG.....	161
6.6.3.	Tiempos de operaciones APP C++	164
6.6.4.	Comparación Driver vs Aplicación de usuario	166
6.6.5.	Tiempo caso de aplicación RDPG	169
6.6.6.	Tiempo caso de aplicación RDP	170

7. Conclusiones y trabajos futuros	171
7.1. Conclusiones	171
7.1.1. Conclusiones referidas al driver MatrixmodG	171
7.1.2. Conclusiones referidas al Proyecto	172
7.2. Trabajos Futuros	172
8. Anexos	175
8.1. Desarrollo de software profesional y Procesos de software	175
8.1.1. Desarrollo de software profesional	175
8.1.2. Procesos de software.....	175
8.1.3. Clasificación de los procesos de software.....	176
8.1.4. Metodología de trabajo Ágil Kanban.....	179
8.1.5. Desarrollo iterativo incremental.....	181
8.2. Trazabilidad de requerimientos.....	183
8.3. Diseño Arquitectónico	184
8.3.1. Decisiones en el diseño arquitectónico	186
8.3.2. Vistas arquitectónicas	187
8.3.3. Arquitectura en capas.....	188
8.4. Gestión de riesgos en proyectos de software	188
8.4.1. Tipos de riesgos	189
8.4.2. Proceso de gestión de riesgos	191
8.5. Matrices y vectores dinámicos.....	193
8.5.1. Implementación de una matriz dinámica	195
8.5.2. Implementación de un vector dinámico	196
9. Referencias Bibliográficas	197

ÍNDICE DE FIGURAS

Figura 2.1 Representación gráfica de una RDP	5
Figura 2.2 Transición temporal	10
Figura 2.3 Funciones del kernel	13
Figura 2.4 Arquitectura del kernel monolítico (a) versus el microkernel (b)	14
Figura 2.5 El rol de llamadas al sistema	16
Figura 2.6 Relación API-C Library-kernel	18
Figura 2.7 Vinculación de un módulo al kernel	22
Figura 2.8 Comunicación con char device drivers	30
Figura 2.9 Argumentos del método lectura	35
Figura 3.1 Características de máquina virtual	62
Figura 3.2 Diagrama general de los casos de uso de matrixmodG	63
Figura 3.3 Diagrama específico de los casos CU1 y CU3	65
Figura 3.4 Diagrama específico de los caso de uso CU4	65
Figura 3.5 Diagrama específico de caso de uso CU2	65
Figura 3.6 Interfaces externas de driver MatrixmodG	74
Figura 3.7 Grafico de análisis de riesgos	77
Figura 4.1 Diseño de objeto matrix_o	94
Figura 4.2 Diseño de objeto vector_o	95
Figura 4.3 Atributos de identificación y estado de un objeto RDPG_o	97
Figura 4.4 Atributos componentes de un objeto RDPG_o	98
Figura 4.5 Métodos de objeto RDPG_o	99
Figura 4.6 Objeto rwlock_t	100
Figura 4.7 Macros de mensajes de debug	101
Figura 4.8 Diagrama de secuencia, método de disparo de transición	102
Figura 4.9 Vista física general de la arquitectura del sistema	104
Figura 4.10 Vista física detallada de la arquitectura del sistema	105
Figura 4.11 Vista física de la arquitectura del sistema y servicios brindados	105
Figura 4.12 Arquitectura de alto nivel del sistema: Vista de desarrollo general	106
Figura 4.13 Arquitectura de Interfaz de usuario	107
Figura 4.14 Arquitectura de Interfaz de pruebas de sistema	108
Figura 4.15 Arquitectura de Driver MatrixmodG	108
Figura 4.16 Arquitectura de driver MatrixmodG_tests	108
Figura 4.17 Diagrama de clase de manera general	109
Figura 4.18 Detalle de enumeraciones en diagrama de clase	110
Figura 4.19 Diagrama de objetos de driver MatrixmodG	111
Figura 4.20 Diagrama de clases desde espacio usuario	112
Figura 4.21 Métodos de objeto RDPG_Driver desde libMatrixmodG	113
Figura 4.22 Diagrama de clases combinado (espacio usuario – espacio kernel)	113
Figura 4.23 Extensión diagrama de clases a múltiples aplicaciones de lenguajes diferentes	114
Figura 6.1 Funciones desde el espacio usuario	129
Figura 6.2 Funciones desde el espacio kernel	129
Figura 6.3 Modelo de RDP de procesador monocore: Plazas	143
Figura 6.4 Modelo de RDP de procesador monocore: Transiciones	143
Figura 6.5 Modelo de RDPG de procesador monocore	145
Figura 6.6 Modelo de procesador monocore alternativa 1	147
Figura 6.7 Modelo de procesador monocore alternativa 2	147

Figura 6.8 Modelo de RDPG para procesador dual Core	148
Figura 6.9 RDPG de procesador dual core, resumen en nombre de plazas	151
Figura 6.10 Ejecución de RDPG procesador dual core con 1 hilo	153
Figura 6.11 Ejecución de RDPG procesador dual core con 3 hilos	154
Figura 6.12 Ejecución de RDPG procesador dual core con 5 hilos	155
Figura 6.13 Modelo de RDP Productor/consumidor con buffer limitado a n	156
Figura 8.1 Especificación ágil y dirigida por un plan	176
Figura 8.2 Porcentajes de inclinación a una metodología ágil y basada en un plan	179
Figura 8.3 Matriz de trazabilidad de Requerimientos vs. Casos de uso	184
Figura 8.4 Arquitectura de un sistema de control para un robot	185
Figura 8.5 Arquitectura genérica en capas	188
Figura 8.6 Proceso de gestión de riesgos	191
Figura 8.7 Estructura de una matriz dinámica	194
Figura 8.8 Estructura de un vector dinámico	195

ÍNDICE DE TABLAS

Tabla 2.1 Arco inhibidor, lector y reset	9
Tabla 2.2 Modificadores de acción	38
Tabla 2.3 Arco inhibidor, lector y reset	38
Tabla 2.4 Banderas de tipo	39
Tabla 2.5 Asociación banderas de tipo con modificadores	39
Tabla 2.6 Situaciones y banderas recomendadas	40
Tabla 2.7 Bloqueo	45
Tabla 2.8 Interbloqueo	48
Tabla 2.9 Métodos de spinlocks	52
Tabla 2.10 Métodos spinlock lector-escritor	53
Tabla 3.1 - CU1: Crear una RDPG	63
Tabla 3.2 - CU2: Gestionar una RDPG	63
Tabla 3.3 - CU3: Eliminar una RDPG	64
Tabla 3.4 - CU4: Configurar parámetros de gestión en driver	64
Tabla 3.5 - CU1.1: Solicitar la creación de una RDPG	66
Tabla 3.6 - CU1.2: Iniciar los componentes bases de la RDPG	66
Tabla 3.7 - CU1.3: Confirmar la creación de la RDPG	66
Tabla 3.8 - CU2.1: Disparar una transición de la RDPG	66
Tabla 3.9 - CU2.2: Actualizar el vector de guardas de transiciones de la RDPG	67
Tabla 3.10 - CU2.3: Solicitar el estado actual de cualquier componente de la RDPG	67
Tabla 3.11 - CU2.4: Solicitar información de la RDPG	67
Tabla 3.12 - CU2.5: Solicitar información cualquier componente de la RDPG	68
Tabla 3.13 - CU4.6: Configurar modo de reserva de memoria	68
Tabla 3.14 - CU4.7: Configurar presentación del estado actual de componentes	68
Tabla 3.15 Requerimientos funcionales del usuario	69
Tabla 3.16 - RF1: El driver permite crear cualquier RDPG en el kernel	70
Tabla 3.17 - RF2: El driver permite eliminar una RDPG previamente cargada en el kernel	70
Tabla 3.18 - RF3	70
Tabla 3.19 - RF4	70
Tabla 3.20 - RF5	71
Tabla 3.21 - RF6	71
Tabla 3.22 - RF7	71
Tabla 3.23 - RF8	72
Tabla 3.24 - RF9	72
Tabla 3.25 Requerimientos de desarrollo	73
Tabla 3.26 Requerimientos de rendimiento	73
Tabla 3.27 - RNF1	73
Tabla 3.28 - RNF2	73
Tabla 3.29 Requerimientos de funcionales de interfaz externa	75
Tabla 3.30 Riesgos posibles que afectan el proyecto	76
Tabla 3.31: Análisis de Riesgos	77
Tabla 3.32: Riesgos confirmados	78
Tabla 3.33: Planificación de estrategias de riesgos	78
Tabla 4.1 Iteraciones del sistema desarrollado	85
Tabla 4.2 Requerimientos cubiertos en iteración 1	86
Tabla 4.3 Requerimientos cubiertos en iteración 2	87

Tabla 4.4 Requerimientos cubiertos en iteración 3	88
Tabla 4.5 Requerimientos cubiertos en iteración 4	89
Tabla 4.6 Requerimientos cubiertos en iteración 5	89
Tabla 4.7 Comandos de configuración de parámetros	90
Tabla 4.8 Comandos de configuración de componentes de una RDPG	92
Tabla 4.9 Comandos de reporte de información de la RDPG	93
Tabla 4.10 Comandos de control del comportamiento de la RDPG	93
Tabla 4.11 Comandos de lectura de componentes de la RDPG	93
Tabla 5.1: Características a ser probadas	116
Tabla 5.2: Características a no ser probadas	117
Tabla 5.3 Comparación Kmemleak vs. LeakChek (KEDR)	123
Tabla 5.4 Modelo de plantilla para pruebas de sistema	124
Tabla 5.5 Modelo plantilla para gestión de defectos	126
Tabla 6.1	131
Tabla 6.2	143
Tabla 6.3	143
Tabla 6.4	145
Tabla 6.5	149
Tabla 6.6	149
Tabla 6.7: PROMEDIO DE TIEMPOS DE METODOS DE ALLOC EN DRIVER (API OPENMP)	158
Tabla 6.8: PROMEDIO DE TIEMPOS DE METODOS EN ALLOC DE DRIVER (LIB TIME)	158
Tabla 6.9: PROMEDIO DE TIEMPOS DE METODOS DE ALLOC EN DRIVER (API OPENMP)	159
Tabla 6.10: PROMEDIO DE TIEMPOS DE METODOS EN ALLOC DE DRIVER (LIB TIME)	159
Tabla 6.11: PROMEDIO DE TIEMPOS DE OPERACIÓN DELETE EN DRIVER (API OPENMP)	160
Tabla 6.12: PROMEDIO DE TIEMPOS DE OPERACIÓN DELETE EN DRIVER (LIB TIME)	160
Tabla 6.13: PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER (API OPENMP)	161
Tabla 6.14 PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER (LIB TIME)	161
Tabla 6.15: PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER (API OPENMP)	162
Tabla 6.16 PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER (LIB TIME)	162
Tabla 6.17: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON OPENMP)	164
Tabla 6.18: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON LIB TIME)	165
Tabla 6.19: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON OPENMP)	165
Tabla 6.20: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON LIB TIME)	165
Tabla 6.21: Comparación Op3	167
Tabla 6.22: Comparación Op4	167
Tabla 6.23: Comparación Op5	167
Tabla 6.24: Comparativa de tiempos de gestión de RDPG	169
Tabla 6.25: Comparativa de tiempos de gestión de la RDP	170

1. INTRODUCCIÓN

1.1. DESCRIPCIÓN GENERAL

En el presente proyecto se realizó la creación de un sistema de software que se adiciona sobre el kernel de Linux, se trata de un device driver que gestiona Redes de Petri Generalizadas en el kernel. Conforme con los objetivos del proyecto se analizó e investigó la posibilidad de obtener un sistema de software que brinde mayor rendimiento en la gestión de Redes de Petri Generalizadas, que el rendimiento que obtiene una aplicación de espacio usuario. La creación del sistema de software, su análisis, estudio e investigación forman parte de los objetivos del proyecto.

Una de las principales ventajas del device driver, es que permite independizarse del lenguaje de programación. Esto significa que todo el trabajo del sistema desarrollado puede ser utilizado por cualquier lenguaje de programación del espacio usuario utilizando llamadas al sistema, mecanismo con el que cuentan la mayoría de los lenguajes. El driver se encuentra en un único lugar, el kernel de Linux, y cualquier aplicación del espacio usuario puede hacer uso del mismo mediante llamadas al sistema.

Desarrollar software de esta manera tiene algunos beneficios importantes, como por ejemplo, permitir que la implementación del sistema de software se ubique en un único lugar, evitando su réplica sobre diferentes lenguajes de programación. Un único plan de pruebas será necesario para impactar sobre todos los lenguajes de programación al mismo tiempo.

Otro beneficio del driver es que permite llevar su rendimiento a cualquier aplicación de usuario. Para lograr esto solo es necesario diseñar una librería de espacio usuario que interactúe con el file device del driver a través de llamadas al sistema compatibles con el lenguaje de usuario utilizado. Esta es una alternativa interesante de usar en conjunto con aplicaciones de lenguajes de programación de espacio usuario que requieren mucha infraestructura y en consecuencia cuentan con bajo rendimiento.

El driver puede ser gestionado desde cualquier interfaz de usuario que utilice llamadas al sistema como por ejemplo la consola o terminal de Linux. Por otro lado para utilizar el driver en aplicaciones de usuario más específicas se desarrolló su librería asociada. La librería se implementó para los lenguajes C y C++, siendo fácil de migrar a cualquier otro lenguaje de espacio usuario como Java, Python, Perl, etc. Esta librería automatiza y gestiona un gran número de operaciones con el driver que se hacen transparentes para el usuario final, buscando que sea todo más simple y sencillo. Los usuarios pueden gestionar las Redes de Petri Generalizadas en el kernel de Linux y aprovechar su rendimiento simplemente haciendo uso de la librería asociada.

Las Redes de Petri Generalizadas extienden la capacidad de expresión de las Redes de Petri Ordinarias mediante el uso de arcos inhibidores, arcos lectores y arcos reset en conjunto con guardas, eventos y semánticas temporales sobre sus transiciones. La solución expresada por la ecuación de estado de las Redes de Petri Generalizadas da origen a un algoritmo que preserva el modelo original, facilita su ejecución en paralelo y permite abordar problemas de mayor tamaño y complejidad [1]. En este proyecto, el sistema desarrollado abarca las extensiones relacionadas a los arcos inhibidores, lectores y reset en conjunto con el uso de guardas sobre las transiciones, la semántica temporal se simula desde la librería de espacio usuario y la extensión de eventos se plantea en trabajos futuros como mejora del presente proyecto.

Por último, como modelo de ejemplo se desarrolló una aplicación de usuario en el lenguaje C++ que hace uso de la librería del driver para probar todas sus funcionalidades y demostrar cómo gestionar el driver desde la librería. Se utiliza la interfaz de usuario para simular un caso de aplicación particular de RDP y otro caso de aplicación de RDGP, dando garantía a todas las funcionalidades del driver propuestas para la gestión de las Redes de Petri Generalizadas, en cada funcionalidad se mide el rendimiento en la gestión de RDGP con una aplicación de espacio usuario que hace uso del driver y otra aplicación que no hace uso del driver, obteniendo una tabla comparativa entre los diferentes resultados de cada aplicación.

El resultado del sistema desarrollado se mantiene en un repositorio remoto cuyo control de versiones se realizó aprovechando todas las funcionalidades brindadas por GIT.

El enlace del repositorio es el siguiente: <https://github.com/gslAgile/Proyecto-MatrixmodG>.

1.2. OBJETIVOS

1.2.1. OBJETIVO GENERAL

Crear un device driver de Linux que gestione Redes de Petri Generalizadas en el kernel y su librería de espacio usuario asociada.

1.2.2. OBJETIVOS ESPECÍFICOS

De manera específica se buscaron los siguientes objetivos:

- Adquirir mayor comprensión en las capacidades de expresión de las RDPG para la solución de problemas y su aplicación práctica en la computación.
- Investigar y analizar posibles mejoras respecto a las funcionalidades y rendimiento del device driver en base a versiones de trabajos previos y al avance continuo de su desarrollo.
- Explorar un nuevo enfoque para la gestión de RDPG en sistemas computacionales y medir su impacto en el rendimiento.
- Adaptar patrones de diseño y el paradigma orientado a objetos (POO) en la programación C del kernel de Linux, con el fin de aprovechar las ventajas de la POO.
- Desarrollar una librería de kernel que permita instanciar las Redes de Petri Generalizadas como objetos.
- Desarrollar una librería de espacio usuario para gestionar el driver.
- Desarrollar una interfaz de usuario que utilice la librería del driver a modo de ejemplo.
- Realizar un caso de aplicación que permita validar la implementación del device driver para la gestión de las RDP.
- Realizar un caso de aplicación que permita validar la implementación del device driver para la gestión de las RDPG.
- Crear una aplicación de espacio usuario de iguales características que el device driver para comparar el rendimiento a través de pruebas Benchmark.

1.3. METODOLOGÍA DE TRABAJO

Durante el desarrollo del sistema de software, en este caso un device driver de Linux, se buscó realizar un desarrollo profesional. Para esto se hizo uso de Ingeniería de Software y en base a la capacidad para juzgar y tomar decisiones, se seleccionó una metodología de trabajo adecuada basada en los conceptos y el análisis de la sección 8.1, qué brinda una guía para tomar una decisión más acertada.

De acuerdo con el análisis realizado en la sección 8.1 sobre que metodología utilizar para el desarrollo del presente proyecto, se observó que en la mayoría de los proyectos de software se necesita encontrar un equilibrio entre procesos dirigidos por un plan y procesos ágiles. Teniendo en cuenta esta última idea y visualizando el equilibrio a través del cuestionario, la tabla y sus gráficas asociadas al análisis, se decidió seleccionar una metodología ágil que se complementa con una metodología dirigida por un plan en los puntos donde se ve que es necesario.

La combinación de metodologías de trabajo que se decidió utilizar para llevar adelante el presente proyecto son Kanban, como metodología ágil, y el desarrollo iterativo incremental como metodología basada en un plan.

La metodología ágil permite mantener el enfoque y la motivación en el desarrollo del sistema de software, ayudará a gestionar y controlar todas las tareas y a cumplir con los tiempos estimados. Por otro lado la metodología dirigida por un plan brindará previsibilidad en los temas y tareas que lo requieran si son de alta complejidad, proporcionará la organización de los requerimientos del sistema de software a crear, su verificación y análisis continuo en cada iteración. De esta forma conociendo el equilibrio asociado al sistema de software se podrá trabajar de una manera adecuada mejorando la eficiencia y calidad del producto final.

1.4. DEFINICIONES, ACRÓNIMOS Y ABREVIATURAS

A lo largo del presente informe del proyecto se encuentran varios acrónimos y abreviaturas para simplificar repetir frases y palabras que aparecen con frecuencia sobre el texto, en los casos donde es necesarios se realizan definiciones.

API: Application Programming Interface, traducido como Interfaz de programación de una aplicación.

Bash: Bourne again shell.

BPI: Banco de casos de pruebas integrales.

BU: Banco de casos de pruebas unitarias.

CA: Caso de Aplicación.

Device driver: Modulo o controlador de un dispositivo particular.

ERS: Especificación de requerimientos de software.

File device: Archivo de dispositivo.

GTEST: GoogleTest.

ID: Identificador.

IEEE: Institute of Electrical and Electronics Engineers, traducido como Instituto de Ingeniería Eléctrica y Electrónica, es una asociación mundial de ingenieros dedicada a la normalización y el desarrollo en áreas técnicas.

IUTS: Interfaz de usuario para pruebas/tests de sistema.

KTF: Kernel Test Framework.

KEDR: Kernel-mode Drivers in Runtime. Es un framework para la detección de pérdidas de memoria en el kernel de Linux.

mII: Matriz de incidencia I de una RDPG.

mIH: Matriz de incidencia H de una RDPG.

mIR: Matriz de incidencia R de una RDPG.

mIRe: Matriz de incidencia de transiciones reset de una RDPG.

vMI: Vector de marcado inicial de una RDPG.

Open sources: Fuentes libres y abiertas para cualquiera.

POO: Programación Orientada a Objetos.

POSIX: Portable Operating System Interface for Unix. Es una norma escrita por la IEEE, que define una interfaz estándar del sistema operativo y el entorno, incluyendo un intérprete de comandos.

RDP: Red de Petri ordinaria.

RDPG: Red de Petri Generalizada.

SMP: Symmetric multiprocessing traducido al español como multiprocesamiento simétrico.

SMPs: Secure Symmetric multiprocessing traducido al español como multiprocesamiento simétrico seguro.

STDIN: Entrada estándar.

STDOUT: Salida estándar.

UML: Unified Modeling Language. También conocido en español como Lenguaje de modelado unificado.

UP: Uniprocessor se traduce como procesador único.

VM: Virtual machine. También es conocido en español como máquina virtual.

2. MARCO TEÓRICO

2.1. REDES DE PETRI ORDINARIAS

Tanto por el alcance de los resultados teóricos como por la diversidad y el número de aplicaciones, las redes de Petri constituyen en la actualidad un modelo formal avanzado y completo para la descripción lógica de las estructuras de control del paralelismo [2].

Las redes de Petri conforman una herramienta gráfica y matemática que puede aplicarse especialmente a los sistemas paralelos que requieran simulación y modelado de la concurrencia en los recursos compartidos, de la misma manera que lo hace los autómatas finitos en los sistemas secuenciales [3].

En su tesis doctoral “Comunicación con autómatas”, Karl Adam Petri en 1962, estableció las bases para una teoría de comunicación entre componentes asíncronos de un sistema de cómputo. Definió una herramienta matemática de propósito general para describir las relaciones que existen entre condiciones y eventos [4].

2.1.1. DEFINICIÓN DE UNA RED DE PETRI ORDINARIA

Una Red de Petri Ordinaria (RDP) es un modelo gráfico, formal y abstracto para describir y analizar el flujo de información. El análisis de las RDP ayuda a mostrar información importante sobre la estructura y el comportamiento dinámico de sistemas modelados como así también la representación matemática del sistema mencionado.

Las RDP están asociadas con la teoría de grafos y podemos considerarlas como autómatas formales o como generadores de lenguajes formales. La siguiente figura es un ejemplo de la representación gráfica de una RDP.

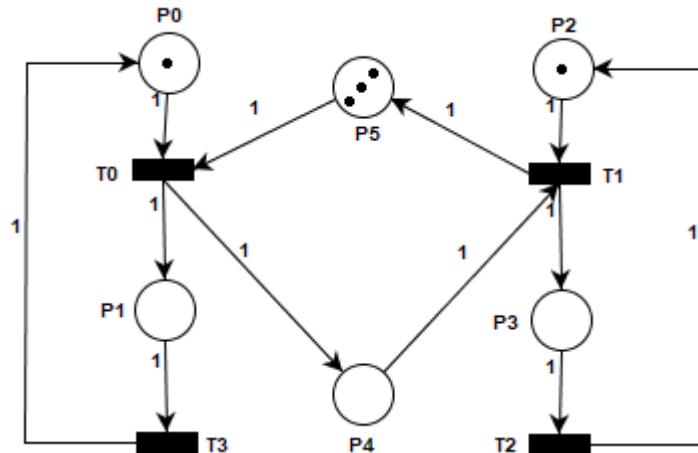


Figura 2.1 Representación gráfica de una RDP.

La representación gráfica de una RDP comprende los siguientes componentes:

- **Plazas:** Graficados por un círculo, nos permite representar estados del sistema. Más precisamente las plazas juegan el papel de variables de estado y toman valores enteros. Las plazas pueden contener tokens.
- **Tokens:** Graficados por un punto negro. Estos representan el valor específico de la condición o estado, generalmente se interpretan como la presencia de recursos de un cierto tipo.
- **Transiciones:** Graficadas por rectángulo, representan el conjunto de sucesos cuya aparición provoca la modificación de los estados del sistema.
- **Arcos:** Indican las interconexiones entre plazas y transiciones y viceversa, estableciendo el flujo de los token, por lo que tienen una flecha que indica el sentido del flujo.

Podemos decir ahora, una RDP es un gráfico dirigido bipartito. Consta de dos componentes: una estructura de red y un marcado inicial.

La estructura de la red contiene dos tipos de nodos: plazas y transiciones. Estos están vinculados por arcos, los cuales están dirigidos de las plazas a las transiciones y viceversa; pero no podemos interconectar plazas o transiciones entre sí.

Las plazas son gráficamente representadas por círculos y las transiciones por cajas o barras.

Una plaza puede contener tokens representados por puntos, o un número entero que señalan la cantidad de tokens. La distribución de los tokens en las plazas de una RDP se denomina marcado de la red y esto se corresponde con un estado del sistema modelado. Las marcas que corresponden al primer estado se denominan marcado inicial.

2.1.2. ESTRUCTURA MATEMÁTICA DE UNA RDP

Una RDP se define matemáticamente como una 4-tupla definida por $RDP = (P, T, I, M_0)$. Donde el significado de cada elemento es el siguiente:

- $P = \{p_1, p_2, \dots, p_n\}$ es un conjunto finito y no vacío. Representa las plazas y su cardinalidad es n .
- $T = \{n_1, n_2, \dots, n_m\}$ es un conjunto finito y no vacío. Representa las transiciones y su cardinalidad es m .
- I = es una función de incidencia de todas las salidas y entradas por cada plaza y transición de la red. Es la función de las relaciones dadas por los arcos entre las plazas y transiciones y viceversa. La función representa el mapeo de todos los arcos entre plazas y transiciones. Esta función se representa por una matriz denominada matriz de incidencia I que relaciona las n plazas con las m transiciones en una matriz de dimensión $n \times m$, en donde cada elemento a_{ij} toman los siguientes valores:
 - $a_{ij} = 0$, si entre p_i hacia t_j o t_j hacia p_i no existe relación de arco.
 - $a_{ij} = W_{ij}$, si entre t_j hacia p_i existe relación de arco.
 - $a_{ij} = -W_{ij}$, si entre p_i hacia t_j existe relación de arco.

Donde W_{ij} es el peso del arco que une la plaza p_i con transiciones t_j o viceversa.

- M_0 es el marcado inicial de la red, se representa como un vector de asignación de recursos a las plazas de la red, de esta forma se define la configuración inicial de los tokens de la red. Por ejemplo puede definirse el marcado de una plaza como $M[i]$, esto indicaría la cantidad de tokens ubicados en la plaza “ i ”.

Por ejemplo, supongamos la RDP de la figura 2.1, de la cual deseamos obtener los conjuntos P y T , I y su marcado inicial M_0 . Entonces los conjuntos de plazas, transiciones y marcado inicial serán:

$$P = \{p_0, p_1, p_2, p_3, p_4, p_5\}; T = \{t_0, t_1, t_2, t_3\}; M_0 = \{1, 0, 1, 0, 0, 3\};$$

La función de incidencia se representa por la matriz de incidencia I de la siguiente manera:

Matriz de incidencia I				
	T0	T1	T2	T3
P0	-1	0	0	1
P1	1	0	0	-1
P2	0	-1	1	0
P3	0	1	-1	0
P4	1	-1	0	0
P5	-1	1	0	0

2.1.3. DINÁMICA DE UNA RDP

Definida la estructura matemática de una RDP, ahora queremos introducir el comportamiento dinámico de esta, el cual está determinado por la habilitación y disparo de sus transiciones, lo que se explica a continuación.

Las reglas de ejecución de una RDP son:

- Una RDP se ejecuta por los disparos de sus transiciones.
- Para que una transición pueda ser disparada debe estar habilitada (o sensibilizada).

Una transición está sensibilizada si cada una de las plazas de entrada a la transición tiene al menos la cantidad de token indicados en el peso, que está en el arco que une la plaza con la transición.

En la figura 2.1, por ejemplo, las transiciones t_0 y t_1 están sensibilizadas mientras que las transiciones t_2 y t_3 no están sensibilizadas. Donde el peso de todos los arcos es $W_{ij} = 1$.

Disparar cualquiera de las transiciones de una RDP implica avanzar a un nuevo estado o permanecer en el estado actual, de acuerdo a si la transición a disparar esta sensibilizada o no lo está respectivamente. La ecuación que define la dinámica de una RDP cuando se dispara una transición se denomina ecuación de estado de la RDP y se define de la siguiente forma.

$$\boxed{M_{i+1} = M_i + (I \times \sigma_i)} \quad (1)$$

Donde Los componentes son:

M_{i+1} : Nuevo estado a alcanzar de acuerdo al disparo dado por el vector U_i .

M_i : Estado actual de la RDP en el instante i .

I : Matriz de incidencia I .

σ_i : Vector disparo responde a la función uno, donde todos sus componentes son nulos a excepto la del disparo a realizar en el instante i .

Esta ecuación de estado responde de manera ideal si se dispara una transición sensibilizada o no sensibilizada, brindando información en ambos casos para determinar un nuevo estado de la RDP si se disparó una transición sensibilizada o permanecer en el mismo estado si se disparó con una transición no sensibilizada.

Es posible que en una RDP se tengan distintas transiciones listas en el mismo instante de tiempo. Las transiciones listas en este caso son concurrente, el sistema de control de disparo es quien realiza la decisión de cual transición se dispara [5].

2.2. REDES DE PETRI GENERALIZADAS

La implementación de hilos en los sistemas informáticos permite explotar los recursos de las arquitecturas multicore. Estos hilos cooperan y se ejecutan concurrentemente. Las aplicaciones diseñadas de esta manera tienen una complejidad intrínseca adicional con respecto a los programas secuenciales. Esta complejidad se manifiesta en el diseño, detección de errores, testing, validación y mantenimiento. Por lo mencionado, es necesario introducir mecanismos de control, como los semáforos, que penalizan los tiempos de ejecución. Por todo esto es conveniente generar una solución formal que garantice y facilite el desarrollo e implementación del sistema [1].

La ejecución y simulación de sistemas complejos requieren modelado de RDP con mayor semántica. Para ampliar la capacidad semántica de las redes se incluyen eventos, guardas, distintos tipos de brazos (inhibidores, lectores, etc.) y semánticas temporales. Esto amplía el dominio de los problemas, el tamaño y la complejidad.

2.2.1. FUNDAMENTOS

Ecuación de estado

Las RDP se utilizan para representar gráficamente el comportamiento dinámico de un sistema. Por lo tanto, las características gráficas de las RDP pueden explotarse para inspeccionar la dinámica de un sistema. Este enfoque es adecuado para sistemas pequeños. Sin embargo, la metodología gráfica no es eficiente cuando los sistemas son grandes y complejos. En esta sección, se desarrolla la ecuación de estado para representar modelos de sistemas grandes y complejos. Es importante introducir la ecuación de estado de las RDP para extenderla con distintos tipos de brazos, eventos, guardas y tiempo. Con esta ecuación es posible obtener el siguiente estado del sistema. Este es el modelo matemático y más simple que la metodología gráfica para analizar la evolución de los sistemas.

Teniendo en cuenta la ecuación de estado (1) definida en la sección 2.1.3, para una RDP de n plazas y m transiciones, cuando se dispara una transición sensibilizada, se puede calcular matemáticamente el siguiente estado utilizando dicha ecuación.

La ecuación de estado representa matemáticamente el comportamiento dinámico del sistema. La matriz de incidencia caracteriza el comportamiento del sistema. Por lo tanto, la matriz de incidencia representa el sistema en sí.

Transición sensibilizada

Una transición está sensibilizada si todos las plazas de entrada a la transición tienen una marca igual o mayor al peso del arco que une cada plaza con la transición. Esto se expresa como:

$$M(p_i) \geq W_{ji}, \forall p_j \in I(t_i)$$

Las transiciones sensibilizadas se expresan como un vector binario E de dimensión $m \times 1$. Cada componente del vector indica con uno la transición sensibilizada y un cero la que no.

$$E = (\text{sign}(S^0), \text{sign}(S^1), \dots, \text{sign}(S^{m-1}))$$

Donde la relación (\cdot) de cada vector S_i es:

- S^i es igual a $M_j + I^i$, un vector con el estado que se obtendría de disparar la i -ésima transición una vez.
- I^0, I^1, \dots, I^{m-1} , son las columnas de la matriz I .
- $\text{sign}(S^i)$, es binario siendo cero si alguna de las componentes de S^i es negativa, de otra forma es uno.

Disparo de una transición

Una transición se puede disparar solo si esta sensibilizada. Cuando se dispara se calcula el nuevo estado $Mj+1$ y el nuevo vector de transiciones sensibilizadas E .

Interpretación de la matriz de incidencia

De la semántica de disparo de una RDP, se interpreta la matriz de incidencia como la evaluación conjuntiva entre las columnas (transiciones) y las restricciones que imponen las filas (plazas). Es decir, en una matriz de

incidencia de dimensión $n \times m$ se evalúan m combinaciones de n variables lógicas, como se expresa en la siguiente ecuación:

$$\left(\bigwedge_{h=0}^{n-1} M(p_h) \geq i_{hi} \right), \forall i = 0, \dots, m-1$$

Donde i_{hi} son los elementos de la matriz I y (ph) es la marca en la plaza h . El resultado son las componentes del vector E .

2.2.2. EXTENSIÓN DE LA ECUACIÓN DE ESTADO

Arcos inhibidores, lectores y reset

Los arcos inhibidores, lectores y reset conectan una plaza con una transición, aumentando la capacidad de expresión de la RDP. Cabe destacar que los arcos inhibidores y lectores posibilitan modelar prioridades.

Arco	Representación grafica	Consecuencia sobre la semántica
Inhibidor		Si $h(p_j, t_i)$ es un arco inhibidor y hay token en p_j , entonces t_i no está sensibilizada.
Lector		Si $(,)$ es un arco lector y no hay token en p_j , entonces t_i no está sensibilizada. El disparo de t_i no consume los token de p_j .
Reset		Si $(,)$ es un arco reset y se dispara t_i se consumen todos los token de p_j . El disparo de t_i pone a cero la marca de la plaza p_j .

Tabla 2.1 Arco inhibidor, lector y reset.

Guardas

Una guarda es una variable lógica asociada a una transición que aumenta la capacidad de expresión. Gráficamente, la guarda se representa con una etiqueta junto a la transición. Si (ti) es una guarda, para disparar t_i se requiere que la transición esté sensibilizada y el valor de la guarda sea verdadero. Las guardas posibilitan comunicar la RDP con el medio haciéndolas no autónoma.

Eventos

Un evento se almacena en una cola que se asocia a una transición, aumentando la capacidad de expresión de la RDP. Gráficamente, la cola se representa como una etiqueta junto a la transición. La cola ci , es un contador, que se incrementa cuando llega un evento y se decrementa cuando la transición asociada se dispara. Para que la transición esté sensibilizada se requiere que la cola tenga al menos un evento. Los eventos comunican la RDP con el medio para hacerla no autónoma.

Tiempo

Hay distintas semánticas temporales. Estas imponen limitaciones de tiempo sobre los disparos, como un intervalo de tiempo asociado a cada transición.

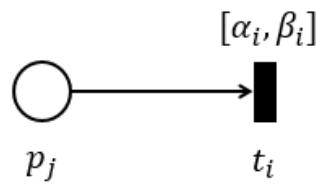


Figura 2.2 Transición temporal.

Es decir, cada transición activa tiene asociada un cronómetro implícito que mide el tiempo transcurrido desde la última vez que se sensibilizó. Una transición sensibilizada puede dispararse si el valor cronometrado, asociado a la transición, está dentro del intervalo de tiempo predeterminado. Por ejemplo, la etiqueta de intervalo $[\alpha_i, \beta_i]$ asociada a t_i , tiene como instante de inicio α_i y β_i como el instante de fin.

Política de selección de disparo

Se refiere a la elección de la próxima transición a disparar de entre todas las sensibilizadas. Para una RDP, incluidas las no autónomas o temporales, si esta elección es aleatoria el sistema resulta no determinístico. Para que el sistema sea determinístico hay diferentes soluciones, como la inclusión de: prioridades, probabilidades, arcos inhibidores, arcos lectores, etc. Con el fin de evitar conflictos en las transiciones usamos la política de servidor único. Es decir, calculamos el nuevo estado considerando solo un disparo de una transición. Si se requieren más de uno, se realiza una secuencia de disparos.

2.2.3. ECUACIÓN DE ESTADO EXTENDIDA

Consideraciones para la generalización de la ecuación

En todas las ampliaciones de la semántica expuesta, se destaca que siempre se trata de determinar cuáles transiciones están sensibilizadas. Una vez determinada la transición a disparar, el disparo se realiza con la ecuación de estado original, excepto para el brazo reset que retira todos los token de la plaza. Para representar matemáticamente la existencia de los brazos enumerados anteriormente se requiere de una matriz para indicar la conexión plaza transición.

Estas matrices son similares a la matriz I . Los términos de las matrices son binarios, dado que los pesos de los arcos son uno. Es decir, para que una transición esté sensibilizada se requiere:

- Si tiene brazo inhibidor que la plaza no tenga token.
- Si tiene brazo lector que la plaza tenga uno o más token
- Si tiene guarda que el valor de la guarda sea verdadero.
- Si tiene evento que tenga uno o más eventos en la cola asociada.
- Si tiene etiqueta con intervalo de tiempo, que el contador se encuentre en el intervalo.

Nueva ecuación de estado

De las consideraciones anteriores observamos que para disparar una transición se requiere la conjunción lógica entre todas las condiciones enumeradas en el punto anterior y el vector E de transiciones sensibilizadas.

El vector de transiciones des-sensibilizadas por arco inhibidor B , es un vector de valores binarios de dimensión $m \times 1$, que indica con un cero cuales transiciones están inhibidas por el arco y uno las que no, y se obtiene de la siguiente ecuación:

$$B = H \cdot Q$$

Donde:

- H es una matriz de dimensión $n \times m$ y Q un vector binario de dimensión $n \times 1$.
- H es la matriz de incidencia que relaciona las plazas que conectan las transiciones con un brazo inhibidor, los términos h_{ij} de la matriz son uno si hay brazo de la plaza p_i a la transición t_j y cero si no lo hay.
- Las componentes q_i del vector Q se obtienen de aplicar la relación

$$\mathbf{q}_i = \text{cero}(\mathbf{M}(\mathbf{p}_i))$$
- La relación $\text{cero}(M(p_i))$, es cero si la marca en la plaza p_i es distinta de cero, en otro caso es uno.

El vector de transiciones des-sensibilizadas por arco lector L , es un vector de valores binarios de dimensión $m \times 1$, que indica con un cero cuales transiciones están inhibidas por el arco y un uno las que no, y se obtiene de la siguiente ecuación:

$$L = R \times W$$

Donde:

- R es una matriz de dimensión $n \times m$ y W un vector binario de dimensión $n \times 1$.
- R es la matriz de incidencia que relaciona las plazas que conectan las transiciones con un brazo lector, los términos r_{ij} de la matriz son un uno si hay brazo de la plaza p_i a la transición t_j y un cero si no lo hay.
- Las componentes w_i del vector W se obtienen de aplicar la relación

$$\mathbf{w}_i = \text{uno}(\mathbf{M}(\mathbf{p}_i))$$

- La relación $\text{uno}(M(p_i))$, es uno si la marca en la plaza p_i es distinta de cero, en otro caso es cero.

El vector de transiciones des-sensibilizadas por guarda G , es un vector de valores binarios de dimensión $m \times 1$, que indica con un cero cuales transiciones están inhibidas por la guarda y un uno las que no, y se obtiene directamente de la guarda.

Vector de transiciones des-sensibilizadas por evento V , es un vector de valores binarios de dimensión $m \times 1$, que indica con un cero cuales transiciones están inhibidas porque no hay evento solicitando el disparo de la transición. Las componentes v_i del vector V se obtienen de aplicar la relación:

$$\mathbf{v}_i = \text{uno}(\text{bufferDeEventos}_i)$$

La relación $(\text{bufferDeEventos}_i)$, es uno si hay uno o más eventos en el buffer asociado con la transición, en otro caso es cero.

El vector de transiciones des-sensibilizadas por tiempo Z , es un vector de valores binarios de dimensión $m \times 1$, que indica con un cero cuales transiciones están inhibidas porque no se ha alcanzado o se ha superado el intervalo de tiempo transcurrido desde de que la transición fue sensibilizada.

$$Z = \text{Tim}(\mathbf{q}(\mathbf{E}, \mathbf{B}, \mathbf{L}, \mathbf{G}, \text{clk}), \text{intervalos})$$

Donde:

- E es el vector de sensibilizado, B es el vector de no-sensibilizado por arco inhibidor, L es el vector de no-sensibilizado por arco lector y G es el vector de no-sensibilizado por guarda.
- La relación $\text{Tim}(q, \text{intervalos})$ es un vector binario de $m \times 1$, el valor de la componente i es uno si q_i , que es un contador, tiene valor en el intervalo indicado por la componente $\text{intervalos}_i = [\alpha_i, \beta_i]$ de otra forma es cero.
- Para que el contador q_i arranque la ecuación $e_i \text{ and } b_i \text{ and } l_i \text{ and } g_i$ debe ser uno, de otra forma el contador se pone a cero, este contador se incrementa en una unidad de tiempo por cada pulso de reloj, de la base de tiempo (clk).
- La matriz intervalos es de dimensión $m \times 2$ y contiene el límite inferior el límite superior de la ventana de tiempo para el disparo.

El vector de transiciones reset A , es un vector de valores enteros de dimensión $m \times 1$, que tiene el valor de la marca de la plaza que se quiere poner a cero, mientras que las otras componentes son uno.

La matriz Re expresa matemáticamente los arcos de reset. El arco que une la plaza pi con la transición tj se indica con un uno en la componente $reij$ de la matriz de otro modo es cero.

$$A = \text{Marca}(Re \times M_j)$$

La multiplicación $Re \times M_j$ es un vector con cero si no hay brazo reset y el valor de la marca si hay brazo reset.

La relación *Marca* es uno si la componente es cero en otro caso el valor de la componente.

El vector A se multiplica elemento a elemento con el vector de disparo σ , operación que indicamos con $\#$. El vector σ tiene un uno en la transición que se quiere disparar por lo que obtenemos la cantidad de disparos necesarios para sacar todas las marcas de la plaza a resetear en otro caso solo un disparo.

Vector de sensibilizado extendido y ecuación de estado

Este vector de sensibilizado extendido Ex se obtiene de la conjunción lógica de todos los vectores anteriores.

$$Ex = E \text{ and } B \text{ and } L \text{ and } V \text{ and } G \text{ and } Z$$

Para introducir el brazo reset hay que multiplicar elemento a elemento ($\#$) a el vector que resulte de la conjunción por A . Por lo que la ecuación de estado extendida resulta:

$$\boxed{M_{i+1} = M_i + I \times ((\sigma \text{ and } Ex) \# A)}$$

La importancia de esta ecuación de estado es que es de simple implementación como circuito combinacional en una FPGA.

2.3. KERNEL DE LINUX

El sistema operativo es la parte del sistema responsable del uso básico y la administración. Esto incluye el kernel y los controladores de dispositivos, el cargador de arranque, el shell de comandos u otra interfaz de usuario, y las utilidades básicas de archivos y sistemas.

El kernel (o núcleo) es la parte más interna del sistema operativo. Este núcleo interno: es el software que proporciona servicios básicos para todas las demás partes del sistema, administra el hardware y distribuye los recursos del sistema.

Los componentes típicos de un kernel son:

- Controladores de interrupciones para atender solicitudes de interrupción.
- Un planificador para compartir el tiempo de procesador entre múltiples procesos.
- Un sistema de gestión de memoria para gestionar espacios de direcciones de procesos.
- Servicios de sistemas como redes y comunicación entre procesos.

Todos los conceptos relacionados al kernel y necesarios para la realización del presente proyecto son extraídos de las bibliografías [6], [7] y [8].

2.3.1. FUNCIONES DEL KERNEL

En un sistema Unix, varios procesos concurrentes atienden diferentes tareas. Cada proceso solicita recursos del sistema, ya sea potencia de cómputo, memoria, conectividad de red o algún otro recurso. El kernel es la gran parte del código ejecutable a cargo de manejar todas estas solicitudes. Aunque la distinción entre las diferentes tareas del kernel no siempre está claramente marcada, el rol del kernel se puede dividir (como se muestra en la Figura 2.3) en las siguientes partes [9]:

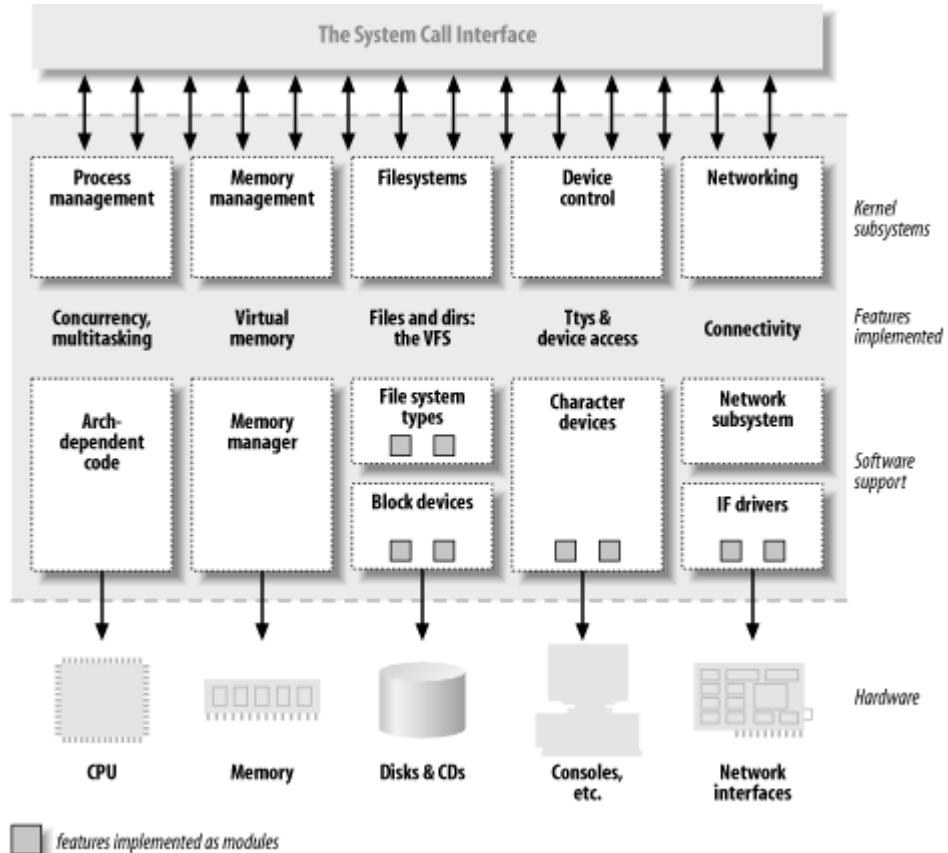


Figura 2.3 Funciones del kernel.

Gestión de proceso: El kernel está a cargo de crear y destruir procesos y manejar su conexión con el mundo exterior (entradas y salidas). La comunicación entre diferentes procesos (a través de señales, tuberías o primitivas de comunicación entre procesos) es básica para la funcionalidad general del sistema y también es manejada por el kernel. Además, el programador, que controla cómo los procesos comparten la CPU, forma parte de la gestión de procesos. De manera más general, la actividad de administración de procesos del kernel implementa la abstracción de varios procesos sobre una sola CPU o algunos de ellos.

Gestión de la memoria: La memoria de la computadora es un recurso importante, y la política utilizada para tratar con ella es crítica para el rendimiento del sistema. El kernel crea un espacio de direccionamiento virtual para todos y cada uno de los procesos, además de los limitados recursos disponibles. Las diferentes partes del kernel interactúan con el subsistema de administración de memoria a través de un conjunto de llamadas a funciones, que van desde un par simple malloc/free a funcionalidades mucho más complejas.

Sistemas de archivos: Unix está fuertemente basado en el concepto de sistema de archivos; casi todo en Unix puede ser tratado como un archivo. El kernel construye un sistema de archivos estructurado sobre el hardware no estructurado, y la abstracción del archivo resultante es muy utilizada en todo el sistema. Además, Linux admite varios tipos de sistemas de archivos, es decir, diferentes formas de organizar datos en el medio físico. Por ejemplo, los discos pueden estar formateados con el sistema de archivos ext3 estándar de Linux, el sistema de archivos FAT comúnmente usado o varios otros.

Control del dispositivo: Casi todas las operaciones del sistema finalmente se asignan a un dispositivo físico. Con la excepción del procesador, la memoria y algunas otras entidades, todas y cada una de las operaciones de control del dispositivo se realizan mediante un código que es específico del dispositivo al que se dirige. Ese código se llama un controlador de dispositivo. El kernel debe tener incorporado un controlador de dispositivo para cada periférico presente en un sistema, desde el disco duro hasta el teclado y la unidad de cinta.

Redes: La red debe ser gestionada por el sistema operativo, porque la mayoría de las operaciones de red no son específicas de un proceso: los paquetes entrantes son eventos asíncronos. Los paquetes deben recogerse, identificarse y enviarse antes de que un proceso se ocupe de ellos. El sistema se encarga de entregar paquetes de datos a través de interfaces de programa y red, y debe controlar la ejecución de los programas de acuerdo con su actividad de red. Además, todos los enrutamiento y los problemas de resolución de direcciones se implementan dentro del núcleo.

2.3.2. ARQUITECTURA DEL KERNEL

La mayoría de los núcleos de Unix son monolíticos: cada capa del núcleo está integrada en el programa entero del núcleo y se ejecuta en modo kernel en nombre del proceso actual. En contraste, los sistemas operativos microkernel exigen un pequeño conjunto de funciones del kernel, por lo general incluyendo algunas primitivas de sincronización, un planificador simple, y un mecanismo de comunicación entre procesos. Varios procesos del sistema que se ejecutan en la parte superior del microkernel implementan otras funciones de las capas del sistema, como los asignadores de memoria, controladores de dispositivos y controladores de llamadas al sistema.

Aunque la investigación académica en los sistemas operativos se orienta hacia los micronúcleos, estos son generalmente más lentos que los monolíticos, debido a que el mensaje explícito que pasa entre las diferentes capas del sistema operativo tiene un costo. Sin embargo, los sistemas operativos microkernel podrían tener algunas ventajas teóricas sobre las monolíticas. Los microkernels obligan a los programadores de sistemas a adoptar un enfoque modular, ya que cada capa del sistema operativo es un programa relativamente independiente que debe interactuar con las otras capas de software a través de interfaces bien definidas y limpias. Por otra parte, un sistema operativo microkernel puede ser fácilmente portado a otras arquitecturas, ya que todos los componentes dependientes de hardware son encapsulados generalmente en el código microkernel.

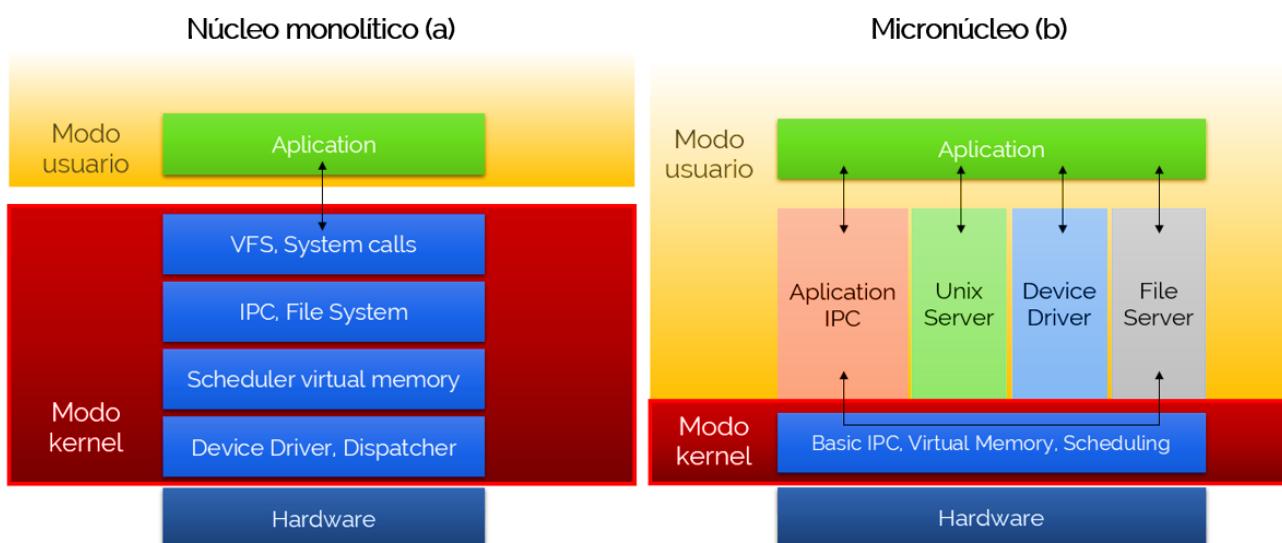


Figura 2.4 Arquitectura del kernel monolítico (a) versus el microkernel (b).

Por último, los sistemas operativos con microkernel tienden a hacer un mejor uso de la memoria de acceso aleatorio (RAM) que los monolíticos, debido a que los procesos del sistema que no están implementando funcionalidades importantes pueden ser cambiados o destruidos.

2.3.3. ESPACIO KERNEL Y ESPACIO USUARIO

La figura 2.4 (a) presenta un esquema general de los componentes principales de la arquitectura del kernel. En la parte inferior se muestra el hardware, que consiste en circuitos integrados (chips), tarjetas, discos, un teclado, un monitor y objetos físicos similares. Por encima del hardware se encuentra el software. La mayoría de las computadoras tienen dos modos de operación: modo kernel y modo usuario. El sistema operativo es la pieza fundamental del software y se ejecuta en modo kernel. En este modo, el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en modo usuario, en el cual sólo un subconjunto de las instrucciones de máquina es permitido. En particular, las instrucciones que afectan el control de la máquina o que se encargan de la E/S (entrada/salida) están prohibidas para los programas en modo usuario.

De manera general se definen el espacio kernel y el espacio usuario como:

Espacio kernel: En los sistemas modernos con unidades de administración de memoria protegidas, el kernel generalmente reside en un estado de sistema elevado, que incluye un espacio de memoria protegido y acceso completo al hardware. Este estado del sistema y el espacio de memoria se denominan colectivamente espacio-kernel.

Espacio usuario: Las aplicaciones se ejecutan en el espacio usuario, donde pueden acceder a un subconjunto de los recursos disponibles de la máquina y pueden realizar ciertas funciones del sistema, acceder directamente al hardware, acceder a la memoria fuera de la asignada por el kernel o, de lo contrario, se tiene un mal comportamiento.

Al ejecutar el código del kernel, el sistema se encuentra en el espacio kernel ejecutándose en modo kernel. Cuando se ejecuta un proceso regular, el sistema está en espacio usuario ejecutándose en modo de usuario.

2.3.4. EL ROL DE LAS LLAMADAS AL SISTEMA

Las aplicaciones se comunican con el kernel a través de llamadas al sistema (Figura 2.5). Una aplicación normalmente llama a las funciones de una biblioteca (por ejemplo, la biblioteca C) que se basa en la interfaz de llamadas del sistema para indicar al núcleo que realice tareas en nombre de la aplicación [6].

- Algunas llamadas a la biblioteca proporcionan muchas características que no se encuentran en la llamada al sistema. Por lo tanto, llamar al kernel es solo un paso en una función que de otro modo sería grande. Por ejemplo, la función printf() proporciona formato y almacenamiento en búfer de los datos; solo un paso en su trabajo es invocar write() para escribir los datos en la consola.
- Algunas llamadas de biblioteca tienen una relación de uno a uno con el kernel. Por ejemplo, la función open() de biblioteca hace poco, excepto la llamada al open() del sistema.
- Otras funciones de la biblioteca de C, como, por ejemplo strcpy(), no hacen uso directo del núcleo.

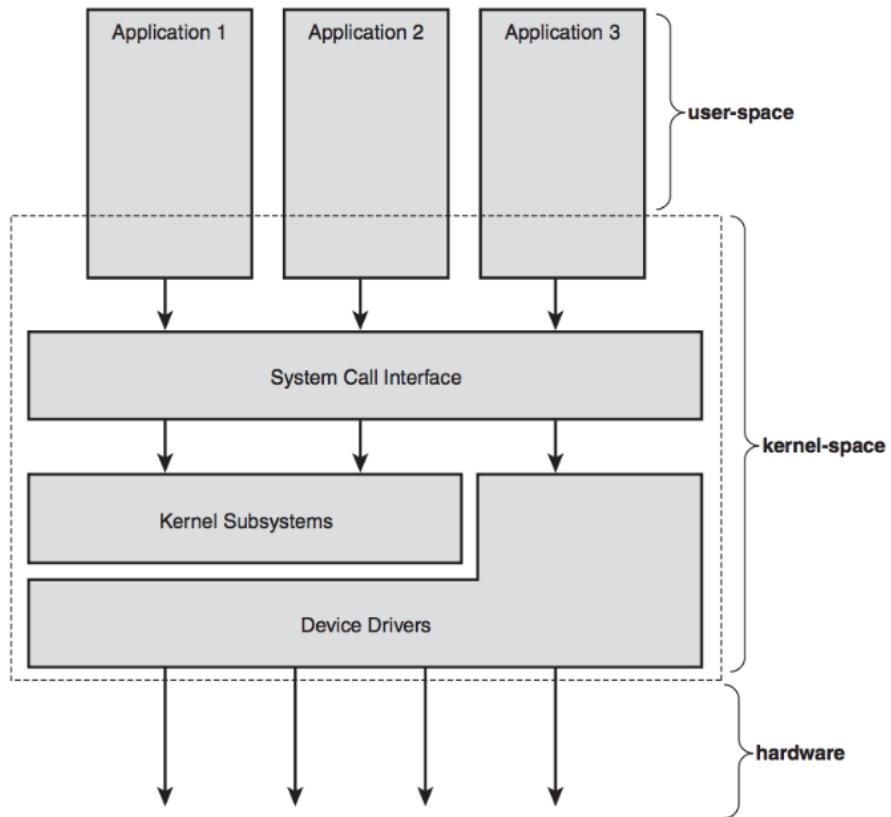


Figura 2.5 El rol de llamadas al sistema en el kernel.

Cuando una aplicación ejecuta una llamada al sistema, el kernel se ejecuta en nombre de la aplicación. Además, se dice que la aplicación está ejecutando una llamada del sistema en el espacio kernel, y el kernel se está ejecutando en el contexto del proceso. Esta relación (que las aplicaciones llaman al kernel a través de la interfaz de llamadas al sistema) es la manera fundamental en que las aplicaciones realizan el trabajo.

2.3.5. EL ROL DE LAS INTERRUPCIONES

El kernel gestiona el hardware del sistema a través de interrupciones. Cuando el hardware quiere comunicarse con el sistema, emite una interrupción que interrumpe el procesador, lo que a su vez interrumpe el núcleo. Un número identifica las interrupciones y el kernel usa este número para ejecutar un controlador de interrupciones específico para procesar y responder a la interrupción. Para proporcionar sincronización, el kernel puede deshabilitar las interrupciones (ya sea todas las interrupciones o solo un número específico de interrupción). En Linux, los controladores de interrupción no se ejecutan en un contexto de proceso. En su lugar, se ejecutan en un contexto de interrupción especial que no está asociado con ningún proceso. Este contexto especial existe únicamente para permitir que un controlador de interrupciones responda rápidamente a una interrupción y luego salga [6].

En Linux, podemos generalizar que cada procesador está haciendo exactamente una de tres cosas en un momento dado:

- En espacio usuario, ejecutando código de usuario en un proceso.
- En kernel-space, en contexto de proceso, ejecutándose en nombre de un proceso específico.
- En el espacio kernel, en el contexto de interrupción, no asociado con un proceso, manejar una interrupción.

2.3.6. KERNEL DE LINUX VERSUS KERNEL UNIX

Existen diferencias notables entre el kernel de Linux y los sistemas Unix clásicos:

- Linux admite la carga dinámica de módulos del kernel. Aunque el kernel de Linux es monolítico, puede cargar y descargar dinámicamente el código del kernel a pedido.
- Linux tiene soporte multiprocesador simétrico (SMP).
- El kernel de Linux es preventivo.
- Linux tiene un enfoque interesante para el soporte de subprocessos: no distingue entre subprocessos y procesos normales. Para el kernel, todos los procesos son iguales (algunos simplemente comparten recursos).
- Linux proporciona un modelo de dispositivo orientado a objetos con clases de dispositivo, eventos de acoplamiento activo y un sistema de archivos de dispositivo de espacio usuario (sysfs).
- Linux ignora algunas características comunes de Unix que los desarrolladores del kernel consideran mal diseñadas, como STREAMS, o estándares que son imposibles de implementar limpiamente.
- Linux es gratis en todos los sentidos de la palabra.

2.4. LLAMADAS AL SISTEMA

En cualquier sistema operativo moderno, el núcleo proporciona un conjunto de interfaces mediante las cuales los procesos que se ejecutan en el espacio de usuario pueden interactuar con el sistema. Estas interfaces dan aplicaciones para:

- Acceso controlado al hardware,
- Un mecanismo con el cual crear nuevos procesos y comunicarse con los procesos existentes y,
- Capacidad de solicitar otros recursos del sistema operativo.

La existencia de estas interfaces, y el hecho de que las aplicaciones no tienen la libertad de hacer directamente lo que quieran, es clave para proporcionar un sistema estable [6].

2.4.1. COMUNICACIÓN CON EL KERNEL

Las llamadas al sistema proporcionan una capa entre el hardware y los procesos del espacio de usuario, que sirve para tres propósitos principales:

- Proporcionar una interfaz de hardware abstracta para el espacio de usuario.
Por ejemplo, al leer o escribir desde un archivo, las aplicaciones no se preocupan por el tipo de disco, medio o incluso el tipo de sistema de archivos en el que reside el archivo.
- Garantizar la seguridad y estabilidad del sistema. El kernel actúa como un intermediario entre los recursos del sistema y el espacio de usuario, por lo que puede arbitrar el acceso según los permisos, los usuarios y otros criterios.
Por ejemplo, este arbitraje evita que las aplicaciones usen hardware de manera incorrecta, roben los recursos de otros procesos o dañen el sistema.
- Una única capa común entre el espacio de usuario y el resto del sistema permite el sistema virtualizado disponible a los procesos. Sería imposible implementar la multitarea y la memoria virtual si las aplicaciones tuvieran la libertad de acceder a los recursos del sistema sin el conocimiento del núcleo.

En Linux, las llamadas al sistema son el único medio que tiene el espacio de usuario para interactuar con el kernel y el único punto de entrada legal en el kernel que no sean las excepciones y traps. A otras interfaces,

como archivos de dispositivos o /proc, se accede a través de llamadas al sistema. Curiosamente, Linux implementa muchas menos llamadas al sistema que la mayoría de los sistemas.

2.4.2. APIs, POSIX Y LA BIBLIOTECA C

APIs

Las aplicaciones normalmente se programan contra una Interfaz de programación de aplicaciones (API) implementada en el espacio de usuario, no directamente a las llamadas del sistema, ya que no se necesita una correlación directa entre las interfaces utilizadas por las aplicaciones y la interfaz real proporcionada por el núcleo.

Una API define un conjunto de interfaces de programación utilizadas por las aplicaciones. Esas interfaces pueden ser:

- Implementada como una llamada al sistema,
- Implementada a través de múltiples llamadas al sistema, o
- Implementada sin el uso de llamadas al sistema en absoluto.

La misma API puede existir en múltiples sistemas y proporcionar la misma interfaz a las aplicaciones, mientras que la implementación de la API en sí misma puede diferir mucho de un sistema a otro.

La figura 2.6 muestra la relación entre una API POSIX, la biblioteca C y las llamadas al sistema.

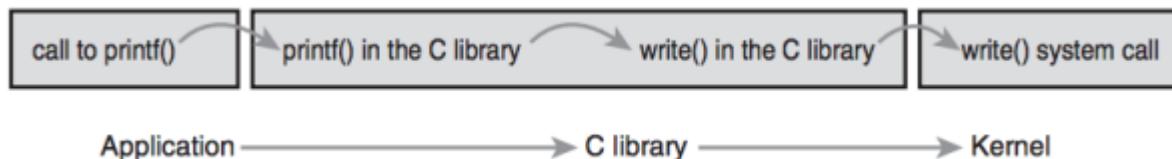


Figura 2.6 Relación, API-C Librar-Kernel.

POSIX

Las APIs más comunes en el mundo Unix se basan en POSIX. Técnicamente, POSIX se compone de una serie de estándares del IEEE que tienen como objetivo proporcionar un estándar de sistema operativo portátil basado aproximadamente en Unix. Linux se esfuerza por ser compatible con POSIX y SUSv3 cuando corresponda.

En la mayoría de los sistemas Unix, las llamadas a la API definidas por POSIX tienen una fuerte correlación con las llamadas al sistema. Algunos sistemas que son más bien Un-Unix, como Microsoft Windows, ofrecen bibliotecas compatibles con POSIX.

Biblioteca C

La interfaz de llamadas al sistema en Linux, como en la mayoría de los sistemas Unix, se proporciona en parte por la biblioteca de C.

La biblioteca C implementa la API principal en los sistemas Unix, que incluye:

- La biblioteca estándar de C.
- La interfaz de llamada al sistema.

La biblioteca de C es utilizada por todos los programas de C y, debido a la naturaleza de C, es fácilmente envuelta por otros lenguajes de programación para usar en sus programas. La biblioteca C, además, proporciona la mayoría de las API POSIX.

Desde el punto de vista del programador de aplicaciones, las llamadas al sistema son irrelevantes; Todo lo que le preocupa al programador es la API. A la inversa, el núcleo solo se ocupa de las llamadas al sistema; El uso

de las llamadas de biblioteca y las aplicaciones que hacen uso de las llamadas del sistema no es una preocupación del núcleo. No obstante, es importante que el núcleo realice un seguimiento de los posibles usos de una llamada al sistema y mantenga la llamada al sistema lo más general y flexible posible.

Un elemento mantenido por los sistemas relacionado con las interfaces en Unix es “Proporcionar un mecanismo, no una política”. En otras palabras, las llamadas al sistema Unix existen para proporcionar una función específica en un sentido abstracto. La manera en que se utiliza la función no es asunto del núcleo.

2.4.3. CONTROLADOR DE LLAMADAS AL SISTEMA

No es posible que las aplicaciones de espacio usuario ejecuten el código del kernel directamente. No pueden simplemente hacer una llamada a una función a un método existente en el espacio kernel porque el kernel existe en un espacio de memoria protegido. De lo contrario, la seguridad y estabilidad del sistema serían inexistentes.

Las aplicaciones de espacio usuario le indican al kernel que desean ejecutar una llamada del sistema y hacen que el sistema cambie al modo kernel, donde el kernel puede ejecutar la llamada del sistema en el espacio kernel en nombre de la aplicación. Este mecanismo es la interrupción del software: incurre en una excepción, y el sistema cambiará al modo kernel y ejecutará el controlador de excepciones. El controlador de excepciones en este caso es en realidad el controlador de llamadas al sistema (system call handler).

La interrupción de software definida en x86 es la interrupción número 128, que se incurre a través de la instrucción int \$0x80. Activa un cambio al modo kernel y la ejecución del vector de excepción 128, que es el controlador de llamadas al sistema. El controlador de llamadas al sistema es la función adecuadamente llamada system_call(). Es dependiente de la arquitectura; en x86-64 se implementa en ensamblador en entry_64.S (arch/x86/kernel/entry_64.S).

Recientemente, los procesadores x86 agregaron una característica conocida como sysenter, que proporciona una forma más rápida y más especializada de interceptar un kernel para ejecutar una llamada al sistema que usar la instrucción int de interrupción. El soporte para esta característica se agregó rápidamente al kernel. Sin embargo, independientemente de cómo se invoque el manejador de llamadas al sistema, la noción importante es que, de alguna manera, el espacio usuario hace que una excepción o captura ingrese al kernel.

2.5. DEVICE DRIVERS DE LINUX

Los drivers, también conocidos como módulos o controladores, son piezas de código que pueden cargarse y descargarse del kernel en tiempo de ejecución. Extienden la funcionalidad del kernel sin la necesidad de reiniciar el sistema. Por ejemplo, un tipo de módulo es el controlador de dispositivo, que permite al núcleo acceder al hardware conectado al sistema. Sin los módulos, tendríamos que construir núcleos monolíticos y agregar una nueva funcionalidad directamente en la imagen del núcleo. Además de tener núcleos más grandes, esto tiene la desventaja de requerirnos reconstruir y reiniciar el núcleo cada vez que queremos una nueva funcionalidad. Por esta razón y muchas otras los módulos son muy útiles en los sistemas operativos [10].

2.5.1. INTRODUCCIÓN

La característica open sources del sistema operativo Linux lo hizo un sistema fácil de examinar, comprender y modificar por cualquier persona con las habilidades necesarias. De esta forma Linux ha ayudado a democratizar los sistemas operativos. Sin embargo el kernel de Linux sigue siendo un cuerpo de código grande y complejo, y los especialistas del kernel necesitan un punto de entrada donde puedan acercarse al código sin ser abrumados por la complejidad. A menudo, los device driver proporcionan esa puerta de enlace [9].

Los controladores de dispositivos asumen un rol especial en el kernel de Linux. Son distintas “cajas negras” que hacen que una pieza particular de hardware responda a una interfaz de programación interna bien definida; Ocultan completamente los detalles de cómo funciona el dispositivo. Las actividades del usuario se realizan por medio de un conjunto de llamadas estandarizadas que son independientes del controlador específico; la asignación de esas llamadas a operaciones específicas del dispositivo que actúan sobre hardware real es, entonces, la función del controlador del dispositivo. Esta interfaz de programación es tal que los controladores pueden compilarse por separado del resto del kernel y “conectarse” en tiempo de ejecución cuando sea necesario. Esta modularidad hace que los controladores de Linux sean fáciles de escribir, hasta el punto de que ahora hay cientos de ellos disponibles.

Hay varias razones para estar interesado en la escritura de los controladores de dispositivos de Linux. La velocidad a la que el nuevo hardware se vuelve disponible (y se vuelve obsoleto!) garantiza que los escritores de controladores estarán ocupados en el futuro inmediato. Es posible que las personas necesiten conocer los controladores para poder acceder a un dispositivo en particular que sea de su interés. Los proveedores de hardware, al hacer que un controlador de Linux esté disponible para sus productos, pueden agregar la amplia y creciente base de usuarios de Linux a sus mercados potenciales. Y la naturaleza de la característica open source (código abierto) del sistema Linux significa que si el controlador así lo desea, la fuente a un controlador se puede difundir rápidamente a millones de usuarios.

El rol de un device driver

Como programador, usted es capaz de tomar sus propias decisiones sobre su controlador, y elegir un compromiso aceptable entre el tiempo de programación requerido y la flexibilidad del resultado. Aunque parezca extraño decir que un controlador es "flexible", nos gusta esta palabra porque enfatiza que la función del controlador de un dispositivo es proporcionar Mecanismo, no política, la política queda libre a criterio de cada programador.

La distinción entre mecanismo y política es una de las mejores ideas detrás del diseño de Unix. La mayoría de los problemas de programación pueden dividirse en dos partes: "qué capacidades deben proporcionarse" (el mecanismo) y "cómo se pueden usar esas capacidades" (la política). Si se abordan los dos problemas en diferentes partes del programa, o incluso en programas diferentes en conjunto, el paquete de software es mucho más fácil de desarrollar y adaptar a las necesidades particulares.

Módulos cargables

Una de las buenas características de Linux es la capacidad de extender en tiempo de ejecución el conjunto de características ofrecidas por el kernel. Esto significa que puede agregar funcionalidad al kernel (y eliminar la funcionalidad también) mientras el sistema está en funcionamiento.

Cada pieza de código que se puede agregar al kernel en tiempo de ejecución se llama módulo. El kernel de Linux ofrece soporte para bastantes tipos (o clases) diferentes de módulos, incluidos, entre otros, controladores de dispositivos. Cada módulo está compuesto por código de objeto (no vinculado a un ejecutable completo) que puede ser enlazado dinámicamente al núcleo en ejecución por el comando insmod y puede ser desvinculado por el comando rmmod.

La figura 2.3 identifica diferentes clases de módulos a cargo de tareas específicas: se dice que un módulo pertenece a una clase específica de acuerdo con la funcionalidad que ofrece. La ubicación de los módulos en la figura 2.3 cubre las clases más importantes, pero está lejos de ser completa porque cada vez hay más funcionalidad. De esta forma Linux se está modularizando.

Clases de módulos

La forma de ver de Linux a los dispositivos, distingue entre tres tipos de dispositivos fundamentales. Cada módulo generalmente implementa uno de estos tipos, y por lo tanto es clasificable como un módulo carácter,

un módulo de bloque o un módulo de red. Esta división de módulos en diferentes tipos, o clases, no es rígida; el programador puede optar por construir enormes módulos que implementen diferentes controladores en una sola porción de código. No obstante, los buenos programadores suelen crear un módulo diferente para cada nueva funcionalidad que implementan, porque la descomposición es un elemento clave de la escalabilidad y la capacidad de ampliación.

Las tres clases de módulos son:

Dispositivos de carácter: Un dispositivo de caracteres (char) es uno al que se puede acceder como un flujo de bytes (como un archivo); un controlador de char se encarga de implementar este comportamiento. Dicho controlador generalmente implementa al menos las llamadas de sistema de apertura, cierre, lectura y escritura. La consola de texto (/dev/console) y los puertos serie (/dev/ttys0 y amigos) son ejemplos de dispositivos char, ya que están bien representados por la abstracción de la secuencia. Se accede a los dispositivos char mediante nodos del sistema de archivos, como /dev/tty1 y /dev/lp0. La única diferencia relevante entre un dispositivo char y un archivo normal es que siempre puede moverse hacia adelante y hacia atrás en el archivo normal, mientras que la mayoría de los dispositivos char son solo canales de datos, a los que solo puede acceder de forma secuencial. Sin embargo, existen dispositivos de char que parecen áreas de datos, y puede moverse hacia adelante y hacia atrás en ellas; por ejemplo, esto generalmente se aplica a los capturadores de fotogramas, donde las aplicaciones pueden acceder a toda la imagen adquirida utilizando mmap o lseek.

Dispositivos de bloque: Como los dispositivos char, los nodos del sistema de archivos acceden a los dispositivos de bloques sobre el directorio /dev. Un dispositivo de bloque es un dispositivo (por ejemplo, un disco) que puede alojar un sistema de archivos. En la mayoría de los sistemas Unix, un dispositivo de bloque solo puede manejar operaciones de E/S que transfieren uno o más bloques completos, que generalmente tienen una longitud de 512 bytes (o una potencia mayor de dos). Linux, en cambio, permite que la aplicación lea y escriba un dispositivo de bloque como un dispositivo char, permite la transferencia de cualquier número de bytes a la vez. Como resultado, los dispositivos de bloque y char difieren solo en la forma en que los datos son administrados internamente por el kernel, y por lo tanto en la interfaz del software del kernel/controlador. Al igual que un dispositivo char, a cada dispositivo de bloque se accede a través de un nodo del sistema de archivos, y la diferencia entre ellos es transparente para el usuario.

Interfaces de red: Cualquier transacción de red es realizada a través de una interfaz, es decir, un dispositivo que puede intercambiar datos con otros hosts. Generalmente, una interfaz es un dispositivo de hardware, pero también puede ser un dispositivo de software puro, como la interfaz de bucle de retorno. Una interfaz de red se encarga de enviar y recibir paquetes de datos, controlados por el subsistema de red del kernel, sin saber cómo las transacciones individuales se asignan a los paquetes reales que se transmiten. Muchas conexiones de red (especialmente las que usan TCP) están orientadas a la transmisión, pero los dispositivos de red están, por lo general, diseñados alrededor de la transmisión y recepción de paquetes. Un controlador de red no sabe nada acerca de las conexiones individuales; solo maneja paquetes.

Hay otras formas de clasificar los módulos de controladores que son ortogonales a los tipos de dispositivos anteriores. En general, algunos tipos de controladores funcionan con capas adicionales de funciones de soporte del kernel para un tipo determinado de dispositivo. Por ejemplo, se puede hablar de Módulos de bus serie universal (USB), módulos serie, módulos SCSI, etc. Todos los dispositivos USB se controlan mediante un módulo USB que funciona con el subsistema USB, pero el dispositivo aparece en el sistema como un dispositivo char (un puerto serie USB, por ejemplo), un dispositivo de bloque (un lector de tarjetas de memoria USB), o un dispositivo de red (una interfaz Ethernet USB).

Seguridad base en los módulos

La seguridad es una preocupación cada vez más importante en los últimos tiempos. Cualquier verificación de seguridad en el sistema se aplica mediante el código del kernel. Si el núcleo tiene agujeros de seguridad,

entonces el sistema en su conjunto tiene agujeros. En la distribución oficial del kernel, solo un usuario autorizado puede cargar módulos; la llamada al sistema `init_module` comprueba si el proceso de invocación es autorizado para cargar un módulo en el kernel. Por lo tanto, cuando se ejecuta un kernel oficial, solo el superusuario, o un intruso que haya logrado tener privilegios, puede explotar el poder del código privilegiado.

2.5.2. MÓDULOS DEL KERNEL VERSUS APLICACIONES

Existen diferencias entre los módulos del kernel contra las aplicaciones del espacio usuario. Si bien la mayoría de las aplicaciones pequeñas y medianas realizan una sola tarea de principio a fin, cada módulo del núcleo simplemente se registra para atender futuras solicitudes, y su función de inicialización termina inmediatamente. En otras palabras, la tarea de la función de inicialización del módulo es preparar la invocación posterior de las funciones del módulo; es como si el módulo dijera: "Aquí estoy, y esto es lo que puedo hacer". La función de salida del módulo (`hello_exit` en el ejemplo "hello world") se invoca justo antes de que se descargue el módulo. Debería decirle al núcleo: "Ya no estoy disponible; no hago nada más". Este tipo de enfoque a la programación es similar a la programación dirigida por eventos, pero si bien no todas las aplicaciones son impulsadas por eventos, todos los módulos del núcleo lo son. Otra diferencia importante entre las aplicaciones controladas por eventos y el código del kernel está en la función de salida: mientras que una aplicación que termina puede ser perezosa al liberar recursos o evitar la limpieza total, la función de salida de un módulo debe deshacer con cuidado todo lo que creó la función `init` o las piezas permanecerán alrededor hasta que se reinicia el sistema.

Como programador, usted sabe que una aplicación puede llamar a funciones que no define: la etapa de enlace resuelve referencias externas utilizando la biblioteca de funciones apropiada. `printf` es una de esas funciones que se pueden llamar y se define en `libc`. Un módulo, por otro lado, está vinculado solo al núcleo, y las únicas funciones que puede llamar son las exportadas por el núcleo; no hay bibliotecas para enlazar. La función `printk` utilizada en `hello.c` anteriormente, por ejemplo, es la versión de `printf` definida dentro del kernel y exportada a los módulos. Se comporta de manera similar a la función original, con algunas diferencias menores, siendo la principal la falta de soporte de punto flotante.

La figura 2.7 muestra cómo se utilizan las llamadas de función y los punteros de función en un módulo para agregar una nueva funcionalidad a un kernel en ejecución.

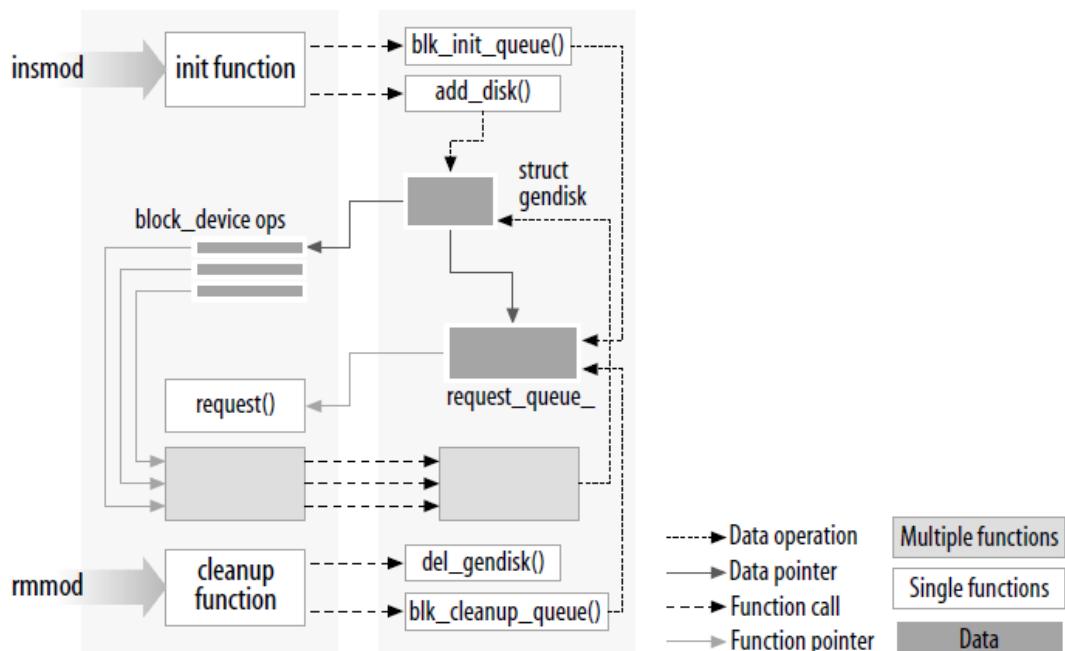


Figura 2.7 Vinculación de un módulo al kernel.

Debido a que ninguna biblioteca está vinculada a módulos, los archivos de origen nunca deben incluir los archivos de encabezado habituales, <stdarg.h> y las únicas situaciones son las únicas excepciones. Solo las funciones que son realmente parte del kernel en sí pueden usarse en los módulos del kernel. Todo lo relacionado con el kernel se declara en los encabezados que se encuentran en el árbol de fuentes del kernel que ha compilado y configurado; la mayoría de los encabezados relevantes se encuentran en include/linux e include/asm , pero otros subdirectorios de include se han agregado al material del host asociado a subsistemas específicos del kernel.

Otra diferencia importante entre la programación del kernel y la programación de aplicaciones es la forma en que se maneja cada entorno de fallas; mientras que una falla de segmentación es inofensiva durante el desarrollo de la aplicación y siempre se puede usar un depurador para rastrear el error del problema en el código fuente, una falla del núcleo destruye el proceso actual como mínimo, si no todo el sistema.

2.5.3. COMPILACIÓN Y EJECUCIÓN DE MÓDULOS

Ejemplo de módulo Hello World

Un ejemplo simple para demostrar cómo se construye un módulo y se carga al sistema es el módulo “hello world”. El siguiente código es el módulo “hello world” completo:

```
// hello.c
#include <linux / init.h>
#include <linux / module.h>
MODULE_LICENSE ("Dual BSD / GPL");

static int hello_init (void)
{
    printk (KERN_ALERT "Hola Mundo \n");
    devuelve 0;
}

static void hello_exit (void)
{
    printk (KERN_ALERT "Adiós, mundo cruel \n");
}

module_init (hello_init);
module_exit (hello_exit);
```

Este módulo define dos funciones, una para invocar cuando el módulo se carga en el kernel (hello_init) y otra para cuando se elimina el módulo (hello_exit). Las líneas module_init y module_exit usan macros especiales del kernel para indicar el rol de estas dos funciones. Otra macro especial (MODULE_LICENSE) se usa para indicar al kernel que este módulo tiene una licencia gratuita; sin tal declaración, el núcleo se queja cuando se carga el módulo.

La función printk se define en el kernel de Linux y se pone a disposición de los módulos; se comporta de manera similar a la función printf de la biblioteca estándar de C. El núcleo necesita su propia función de impresión porque se ejecuta solo, sin la ayuda de las bibliotecas de C. El módulo puede llamar a printk porque, después de que insmod haya cargado el modulo, este está vinculado al kernel y puede acceder a los símbolos públicos del kernel (funciones y variables). La cadena KERN_ALERT es la prioridad del mensaje.

Se puede probar cargar el módulo con el comando insmod y liberar del kernel con el comando rmmod, como se muestra a continuación. Tener en cuenta que solo el superusuario puede cargar y descargar un módulo.

```

root# make
make [1]: ingresando al directorio `/usr/src/linux-2.6.10 '
 CC [M] /home/ldd3/src/misc-modules/hello.o
 Módulos de construcción, etapa 2.
 MODPOST
 CC /home/ldd3/src/misc-modules/hello.mod.o
 LD [M] /home/ldd3/src/misc-modules/hello.ko
make [1]: dejando el directorio `/usr/src/linux-2.6.10 '
root# insmod ./hello.ko
Hola Mundo
root# rmmod hello
Adiós mundo cruel
root#

```

De acuerdo con el mecanismo que utiliza su sistema para entregar las líneas de mensaje, su salida puede ser diferente. Si se está ejecutando insmod y rmmod desde un emulador de terminal que se ejecuta bajo el sistema de ventanas, no verá nada en su pantalla.

Compilación de un modulo

El proceso de compilación de los módulos difiere significativamente del utilizado para las aplicaciones del espacio usuario; El kernel es un programa grande e independiente con requisitos detallados y explícitos sobre cómo se juntan sus piezas. El proceso de compilación también difiere de cómo se hicieron las cosas con versiones anteriores del kernel; El nuevo sistema de compilación es más fácil de usar y produce resultados más correctos, pero se ve muy diferente de lo que venía antes. El sistema de compilación del kernel es una bestia compleja, y solo observamos una pequeña parte de él. Los archivos que se encuentran en el directorio Documentation/kbuild en la fuente del kernel son obligatorios para que todos los que quieran entender todo lo que realmente está sucediendo debajo de la superficie.

Crear un makefile para un módulo es sencillo. De hecho, para el ejemplo de “hello world” que se mostró anteriormente, con una sola línea será suficiente:

```
obj-m := hello.o
```

Es probable que los lectores que están familiarizados con make, pero no con el sistema de compilación del kernel 2.6, se estén preguntando cómo funciona este makefile. La línea anterior no es cómo se ve un makefile tradicional, después de todo. La respuesta, por supuesto, es que el sistema de compilación del núcleo se encarga del resto. La asignación anterior (que aprovecha la sintaxis extendida proporcionada por GNU make) indica que hay un módulo que se construirá a partir del archivo objeto hello.o. El módulo resultante se llama hello.ko después de compilarse desde el archivo objeto.

Si, en cambio, tiene un módulo llamado module.ko que se genera a partir de dos archivos de origen (llamados, por ejemplo, file1.c y file2.c), el conjunto de líneas correcto sería:

```
obj-m := module.o
module-objs := file1.o file2.o
```

Para que un makefile como los que se muestran arriba funcione, debe invocarse dentro del contexto del sistema de compilación más grande del kernel. Si su árbol de fuentes del núcleo está ubicado en, digamos, su directorio ~/kernel-2.6, el comando requerido para construir su módulo (escrito en el directorio que contiene la fuente del módulo y el makefile) sería:

```
make -C ~/kernel-2.6 M=`pwd` modules
```

Este comando comienza cambiando su directorio por el que se proporciona con la opción -C (es decir, el directorio de origen de su núcleo). Allí encuentra el makefile de nivel superior del kernel. La opción M= hace que el makefile se mueva de nuevo al directorio de origen de su módulo antes de intentar construir el modulo

destino (modules). Este objetivo, a su vez, se refiere a la lista de módulos que se encuentran en la variable obj-m, que hemos establecido en módulo.o de nuestro ejemplo.

Escribir el comando make anterior puede volverse aburrido después de un tiempo, por lo que los desarrolladores del kernel han desarrollado una especie de lenguaje makefile, que hace la vida más fácil para aquellos que construyen módulos fuera del árbol del kernel. El truco es escribir el makefile como sigue:

```
# Si se define KERNELRELEASE, hemos sido invocados desde
# el sistema de compilación del kernel y puede utilizar su Lenguaje.
ifeq ($(KERNELRELEASE),)
    obj-m := hello.o

# De lo contrario nos llamaron directamente desde La Línea de comandos;
# Se invoca el sistema de compilación del kernel.
else

KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD    := $(shell pwd)

default:
    $(MAKE) -C $(KERNELDIR) M=$(PWD) modules

endif
```

Una vez más, estamos viendo la prolongada sintaxis GNU make en acción. Este makefile se lee dos veces en una compilación típica. Cuando se invoca el makefile desde la línea de comando, se da cuenta de que la variable KERNELRELEASE no se ha establecido. Localiza el directorio de origen del kernel aprovechando el hecho de que el enlace simbólico construido en el directorio de módulos instalados apunta al árbol de compilación del kernel. Si en realidad no está ejecutando el kernel para el que está compilando, puede proporcionar una opción KERNELDIR= en la línea de comandos, establecer la variable KERNELDIR de entorno o volver a escribir la línea que se establece KERNELDIR en el archivo make. Una vez que se ha encontrado el árbol fuente del kernel, el makefile invoca a default: target, que ejecuta un segundo comando make (parametrizado en el makefile como \$(MAKE)) para invocar el sistema de compilación del kernel como se describió anteriormente. En la segunda lectura, los conjuntos de archivos make obj-m, y los archivos make del kernel, se encargan de construir el módulo.

Este mecanismo para construir módulos puede parecer poco manejable y oscuro. Sin embargo, una vez que se haya acostumbrado, probablemente apreciará las capacidades que se han programado en el sistema de compilación del kernel. Tenga en cuenta que lo anterior no es un makefile completo; un makefile real incluye el tipo habitual de objetivos para limpiar archivos innecesarios, instalar módulos, etc.

Por ejemplo dos formas muy utilizadas son las siguientes [10]:

```
ifeq ($(KERNELRELEASE),)
    obj-m := hello.o

else

    KDIR ?= /lib/modules/$(shell uname -r)/build
    PWD    := $(shell pwd)

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean

endif
obj-m := hello.o

KDIR ?= /lib/modules/$(shell uname -r)/build
PWD    := $(shell pwd)
```

```

default:
    $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
    $(MAKE) -C $(KDIR) M=$(PWD) clean

```

Tener en cuenta que el kernel 2.6 introduce una nueva convención de nomenclatura de archivos: los módulos del kernel ahora tienen una extensión .ko (en lugar de la antigua extensión .o) que los distingue fácilmente de los archivos de objetos convencionales. La razón de esto es que contienen una sección adicional .modinfo donde se guarda información adicional sobre el módulo.

2.5.4. CARGA Y DESCARGA DE MÓDULOS

Programa insmod

Una vez construido el módulo, el siguiente paso es cargarlo en el kernel. Como ya hemos señalado, el programa insmod hace el trabajo por usted. El programa carga el código y los datos del módulo en el kernel, que, a su vez, realiza una función similar a la de ld, ya que vincula cualquier símbolo no resuelto en el módulo a la tabla de símbolos del kernel. Sin embargo, a diferencia del enlazador, el kernel no modifica el archivo de disco del módulo, sino una copia en memoria. Insmod acepta una serie de opciones de línea de comandos y puede asignar valores a los parámetros en su módulo antes de vincularlo al núcleo actual. Por lo tanto, si un módulo está diseñado correctamente, se puede configurar en el momento de la carga; la configuración en tiempo de carga le da al usuario más flexibilidad que la configuración en tiempo de compilación, que aún se usa a veces.

Los lectores interesados pueden querer ver cómo el kernel admite insmod: se basa en una llamada al sistema definida en kernel/module.c. La función sys_init_module asigna la memoria del kernel para contener un módulo (esta memoria se asigna con vmalloc); luego copia el texto del módulo en esa región de memoria, resuelve las referencias del kernel en el módulo a través de la tabla de símbolos del kernel y llama a la función de inicialización del módulo para que todo funcione.

Si realmente busca en la fuente del kernel, encontrará que los nombres de las llamadas al sistema tienen el prefijo sys_. Esto es cierto para todas las llamadas al sistema y no para otras funciones; es útil tener esto en cuenta al grepping para las llamadas del sistema en las fuentes.

Programa modprobe

El programa modprobe, al igual que insmod , carga un módulo en el kernel. Se diferencia en que mirará el módulo que se cargará para ver si hace referencia a algún símbolo que no esté definido actualmente en el núcleo. Si se encuentra alguna de estas referencias, modprobe busca otros módulos en la ruta de búsqueda del módulo actual que definen los símbolos relevantes. Cuando modprobe encuentra esos módulos (que son necesarios para el módulo que se está cargando), también los carga en el kernel. Si se usa insmod en esta situación, en cambio, el comando falla con un mensaje de “símbolos no resueltos” que se deja en el archivo de registro del sistema.

Programa rmmod

Como se mencionó anteriormente, los módulos pueden eliminarse del kernel con la utilidad del programa rmmod. Tenga en cuenta que la eliminación del módulo falla si el kernel cree que el módulo todavía está en uso (por ejemplo, un programa todavía tiene un archivo abierto para un dispositivo exportado por los módulos), o si el kernel se ha configurado para no permitir la eliminación del módulo. Es posible configurar el kernel para permitir la eliminación "forzada" de los módulos, incluso cuando parecen estar ocupados. Sin embargo, si llega

a un punto en el que está considerando utilizar esta opción, es probable que las cosas hayan ido tan mal como para que un reinicio sea la mejor opción.

Programa lsmod

El programa lsmod produce una lista de los módulos cargados actualmente en el kernel. También se proporciona otra información, como cualquier otro módulo que haga uso de un módulo específico. lsmod funciona leyendo el archivo virtual/proc/modules . La información sobre los módulos cargados actualmente también se puede encontrar en el sistema de archivos virtual sysfs en /sys/module.

2.5.5. SÍMBOLOS DEL KERNEL

En el lenguaje de programación, un símbolo es una variable o una función. O más generalmente, podemos decir, un símbolo es un nombre que representa un espacio en la memoria, que almacena datos (variable, para leer y escribir) o instrucciones (función, para ejecutar). Para hacer la vida más fácil para la cooperación entre varias unidades de funciones del kernel, hay miles de símbolos globales en el kernel de Linux. Una variable global se define fuera de cualquier cuerpo de función. Una función global se declara sin la línea static. Todos los símbolos globales se enumeran en /proc/kallsyms [11].

Puede pensar que los símbolos del kernel son visibles en tres niveles diferentes en el código fuente del kernel:

- **static:** visible solo dentro de su propio archivo fuente.
- **extern:** potencialmente visible a cualquier otro código fuente incorporado en la compilación del módulo y del propio kernel.
- **exported:** visible y disponible para cualquier módulo cargable.

Tabla de símbolos del kernel

Hemos visto cómo insmod resuelve símbolos indefinidos contra la tabla de símbolos públicos del kernel. La tabla contiene las direcciones de los elementos globales del kernel (funciones y variables) que se necesitan para implementar controladores modularizados. Cuando se carga un módulo, cualquier símbolo exportado por el módulo se convierte en parte de la tabla de símbolos del kernel. En el caso habitual, un módulo implementa su propia funcionalidad sin la necesidad de exportar ningún símbolo. Sin embargo, debe exportar símbolos siempre que otros módulos puedan beneficiarse de su uso [9].

Los nuevos módulos pueden usar símbolos exportados por su módulo, y puede apilar nuevos módulos sobre otros módulos. El apilamiento de módulos también se implementa en las fuentes principales del kernel.

Los archivos de encabezado del kernel de Linux proporcionan una forma conveniente de administrar la visibilidad de sus símbolos, reduciendo así la contaminación del espacio de nombres (llenando el espacio de nombres con nombres que pueden entrar en conflicto con los definidos en otras partes del kernel) y promoviendo la ocultación adecuada de la información. Si su módulo necesita exportar símbolos para ser utilizados por otros módulos, se deben usar las siguientes macros.

```
EXPORT_SYMBOL(name);  
EXPORT_SYMBOL_GPL(name);
```

Cualquiera de las macros anteriores hace que el símbolo dado esté disponible fuera del módulo. La versión _GPL hace que el símbolo esté disponible solo para los módulos con licencia GPL. Los símbolos se deben exportar en la parte global del archivo del módulo, fuera de cualquier función, porque las macros se expanden a la declaración de una variable de propósito especial que se espera que sea accesible a nivel mundial. Esta variable se almacena en una parte especial del ejecutable del módulo (una "sección ELF") que utiliza el núcleo en el momento de la carga para encontrar las variables exportadas por el módulo. (Los lectores interesados

pueden consultar <linux/module.h> para conocer los detalles, aunque los detalles no son necesarios para hacer que las cosas funcionen.) Un ejemplo de la exportación de un símbolo es el siguiente:

```
#include <linux/export.h> /* IMPORTANTE */
...
/* Función que deseamos exportar */
int foo(void) {
    /** Cuerpo de La función ***/
    ...
    return 0;
}
EXPORT_SYMBOL(foo);
```

Recordar que es importante incluir la biblioteca linux/export.h para exportar adecuadamente un símbolo de un módulo.

2.5.6. PRELIMINARES

La mayoría del código del kernel termina incluyendo un número bastante grande de archivos de encabezado para obtener definiciones de funciones, tipos de datos y variables. Algunos de estos son específicos de los módulos y deben aparecer en todos los módulos cargables. Por lo tanto, casi todo el código del módulo tiene lo siguiente:

```
#include <linux/module.h>
#include <linux/init.h>
```

module.h: contiene una gran cantidad de definiciones de símbolos y funciones que necesitan los módulos cargables. Necesita init.h para especificar sus funciones de inicialización y liberación, como vimos en el ejemplo anterior “hola mundo”, y que volveremos a examinar en la siguiente sección. La mayoría de los módulos también incluyen moduleparam.h para permitir el paso de parámetros al módulo en el momento de la carga.

No es estrictamente necesario, pero su módulo realmente debe especificar qué licencia se aplica a su código. Hacerlo es solo cuestión de incluir una sola línea MODULE_LICENSE:

```
MODULE_LICENSE("GPL");
```

Las licencias específicas reconocidas por el núcleo son “GPL” (para cualquier versión de la Licencia Pública General de GNU), “GPL v2” (solo para GPL versión dos), “GPL y derechos adicionales”, “BSD/GPL doble”, Dual MPL/GPL, y “Propietario”. A menos que su módulo esté marcado explícitamente como que esté bajo una licencia libre reconocida por el kernel, se asume que es propietario, y el kernel está “contaminado” cuando el módulo está cargado.

Otras definiciones descriptivas que pueden estar contenidas dentro de un módulo incluyen MODULE_AUTHOR (indicando quién escribió el módulo), MODULE_DESCRIPTION (una declaración legible de lo que hace el módulo), MODULE_VERSION (para un número de revisión de código; consulte los comentarios en <linux/module.h> para las convenciones que se utilizarán para crear cadenas de versión), MODULE_ALIAS (otro nombre por el cual se puede conocer este módulo), y MODULE_DEVICE_TABLE (para indicar al usuario qué dispositivos admite el módulo).

Las diversas declaraciones MODULE_ pueden aparecer en cualquier lugar dentro de su archivo fuente fuera de una función. Sin embargo, una convención relativamente reciente en el código del kernel es poner estas declaraciones al final del archivo.

2.5.7. INICIALIZACIÓN Y LIMPIEZA

Como ya se mencionó, la función de inicialización del módulo registra cualquier facilidad ofrecida por el módulo. Por facilidad, nos referimos a una nueva funcionalidad, ya sea un controlador completo o una nueva abstracción de software, a la que se puede acceder mediante una aplicación. La definición real de la función de inicialización siempre se ve como:

```
static int __init initialization_function(void)
{
    /* Initialization code here */
}
module_init(initialization_function);
```

Las funciones de inicialización deben declararse static, ya que no están destinadas a ser visibles fuera del archivo específico; Sin embargo, no hay una regla estricta al respecto, ya que no se exporta ninguna función al resto del kernel a menos que se solicite explícitamente. El modificador “`__init`” en la definición puede parecer un poco extraño; es una sugerencia para el núcleo indicando que la función dada se usa solo en el momento de la inicialización. El cargador de módulos descarta la función de inicialización después de cargar el módulo, lo que hace que su memoria esté disponible para otros usos. Hay una etiqueta similar (`__initdata`) para los datos utilizados solo durante la inicialización. El uso de `__init` e `__initdata` es opcional, pero vale la pena. Solo debe asegurar de no usarlos para ninguna función (o estructura de datos) que usará después de que se complete la inicialización. También puede encontrar `__devinit` y `__devinitdata` en la fuente del núcleo; estos se traducen `__init` y `__initdata` solo si el kernel no se ha configurado para dispositivos que se pueden conectar en ejecución (hotpluggable).

El uso de `module_init` es obligatorio. Esta macro agrega una sección especial al código objeto del módulo que indica dónde se encuentra la función de inicialización del módulo. Sin esta definición, su función de inicialización nunca se llama.

Los módulos pueden registrar muchos tipos diferentes de facilidades, incluidos diferentes tipos de dispositivos, sistemas de archivos, transformaciones criptográficas y más. Para cada instalación, hay una función específica del kernel que realiza este registro. Los argumentos pasados a las funciones de registro del kernel suelen ser indicadores de las estructuras de datos que describen la nueva instalación y el nombre de la instalación que se está registrando. La estructura de datos generalmente contiene punteros de función del módulo, que es cómo se llaman las funciones en el cuerpo del módulo.

Los elementos que se pueden registrar incluyen, entre otros, puertos serie, varios dispositivos, entradas al sistema, archivos /proc, dominios ejecutables y disciplinas de línea. Muchos de esos elementos registrables admiten funciones que no están directamente relacionadas con el hardware, pero permanecen en el campo “abstracciones de software”. Esos elementos se pueden registrar, porque de todos modos se integran en la funcionalidad del controlador (como los archivos /proc y las disciplinas de línea, por ejemplo).

Existen otras instalaciones que se pueden registrar como complementos para ciertos controladores, que utilizan la técnica de apilamiento, como se describe en la sección 2.4.5. La mayoría de las funciones de registro tienen un prefijo `register_`, por lo que otra forma posible de encontrarlos es grep `register_` en la fuente del kernel.

Función de limpieza

Cada módulo no trivial también requiere una función de limpieza, que anula el registro de las interfaces y devuelve todos los recursos al sistema antes de que se elimine el módulo. Esta función se define como:

```
static void __exit cleanup_function(void)
{
    /* Cleanup code here */
}
```

```
module_exit(cleanup_function);
```

La función de limpieza no tiene valor de retorno, por lo que se declara void. El modificador “`__exit`” marca el código como solo para la descarga del módulo (al hacer que el compilador lo coloque en una sección especial de ELF). Si su módulo está integrado directamente en el kernel, o si su kernel está configurado para no permitir la descarga de módulos, las funciones marcadas `__exit` simplemente se descartan. Por esta razón, una función marcada `__exit` puede llamarse solo durante la descarga del módulo o el tiempo de apagado del sistema; Cualquier otro uso es un error. Una vez más, la declaración `module_exit` es necesaria para permitir que el kernel encuentre su función de limpieza.

Si su módulo no define una función de limpieza, el núcleo no permite que se descargue.

2.5.8. CHAR DEVICE DRIVER

Como se mencionó anteriormente, una clase de módulo es el controlador de dispositivo, que proporciona funcionalidad para hardware como una tarjeta de TV o un puerto serie. En Unix, cada pieza de hardware está representada por un archivo ubicado en `/dev` denominado archivo de dispositivo (device file) que proporciona los medios para comunicarse con el hardware. El controlador del dispositivo proporciona la comunicación en nombre de un programa de usuario. Por lo tanto, el controlador del dispositivo de la tarjeta de sonido IS1370.o puede conectar el archivo del dispositivo `/dev/sound` a la tarjeta de sonido Ensoniq IS1370. Un programa de espacio usuario como mp3blaster puede usar `/dev/sound` sin saber qué tipo de tarjeta de sonido está instalada [10].

Un controlador de caracteres (char) es una clase de módulo de dispositivos de hardware más simples. Los controladores char también son más fáciles de entender que los controladores de bloque o controladores de red [9].

La figura 2.8 muestra las diferentes interfaces existentes en la comunicación entre un programa de usuario con su hardware correspondiente.



Figura 2.8 Comunicación con char device drivers.

Los programas de usuario se comunican con los device driver utilizando los device files asociados en conjunto con llamadas al sistema. Para ellos el programa de usuario debe abrir el device file que tiene asociado el device

driver, la palabra abrir implica una llamada al sistema. Luego de realizar esta operación el programa de usuario puede acceder a las funcionalidades que el device driver ofrece.

El kernel de Linux es el encargado de recibir las llamadas al sistema realizadas por los programas de usuario y enviarlas hacia el device driver correspondiente. El device driver será el encargado de interactuar con su hardware asociado en caso de ser necesario.

Archivo de dispositivo, número mayor y número menor

Los dispositivos de carácter son accedidos mediante nombres en el sistema de archivos. Esos nombres se denominan archivos especiales o archivos de dispositivos o simplemente nodos del árbol del sistema de archivos; están convenientemente ubicados en el directorio /dev. Los archivos especiales para los controladores char se identifican con una "c" en la primera columna de la salida del comando ls -l. Los dispositivos de bloque también aparecen en /dev , pero se identifican con una "b".

Si emite el comando ls -l, verá dos números (separados por una coma) en las entradas del archivo del dispositivo antes de la fecha de la última modificación, donde normalmente aparece la longitud del archivo. Estos números son el número de dispositivo mayor y menor asociados particularmente al dispositivo.

Tradicionalmente, el número mayor identifica el controlador asociado con el dispositivo. Por ejemplo, /dev/null y /dev/zero son administrados por el controlador.

El núcleo utiliza el número menor para determinar exactamente a qué dispositivo se está haciendo referencia. Dependiendo de cómo esté escrito su controlador (como veremos más adelante), puede obtener un puntero directo a su dispositivo desde el kernel, o puede usar el número menor usted mismo como un índice en una tabla local de dispositivos.

Una de las primeras cosas que un controlador tendrá que hacer al configurar un dispositivo char es obtener uno o más números de dispositivo para trabajar. Para esto se puede utilizar un mecanismo de asignación manual o un mecanismo de asignación dinámica. A su vez es necesaria la creación del archivo de dispositivo asociado al driver utilizando estos números, aquí también se puede crear el archivo de manera manual desde el espacio usuario o dinámicamente desde el kernel. Tanto para la asignación de los números mayor y menor como para la creación de los archivos de dispositivo asociados a un driver es recomendable utilizar los métodos dinámicos ya que desligan al usuario de pasos previos para el uso del driver.

Estructura file_operations

La mayor parte de las operaciones fundamentales de controladores implican tres importantes estructuras de datos del núcleo, llamadas file_operations, file e inode. Se requiere una familiaridad básica con estas estructuras para poder hacer algo interesante, aquí explicaremos la estructura de operaciones file_operations.

Hasta ahora, hemos reservado solo números de dispositivo para nuestro uso, pero aún no hemos conectado ninguna de las operaciones de un controlador con esos números. La estructura file_operations es cómo, un controlador char, configura esta conexión. La estructura, definida en <linux/fs.h>, es una colección de punteros de función. Cada archivo abierto (representado internamente por una estructura file) está asociado con su propio conjunto de funciones (al incluir un campo llamado f_op que apunta a una estructura file_operations). Las operaciones se encargan principalmente de implementar las llamadas al sistema y, por lo tanto, se denominan open, read, write y así con todas. Podemos considerar que el archivo es un "objeto" y que las funciones que operan en él son sus "métodos", utilizando la terminología de programación orientada a objetos para denotar las acciones declaradas por un objeto para que actúen sobre sí mismas.

Convencionalmente, una estructura file_operations o un puntero a una estructura, se llama fops (o alguna variación similar). Cada campo en la estructura debe apuntar a una función implementada en el controlador de una operación específica, o debe dejarse NULL para operaciones no admitidas. El comportamiento exacto del kernel cuando se especifica un puntero NULL es diferente para cada función.

La siguiente lista presenta todas las operaciones más importantes que una aplicación puede invocar en un dispositivo. Tratamos de mantener la lista breve para que pueda usarse como referencia, simplemente resumiendo cada operación y el comportamiento predeterminado del kernel cuando se usa un puntero NULL.

A medida que lea la lista de métodos file_operations, observará que varios parámetros incluyen la cadena `_user`. Esta anotación es una forma de documentación, señalando que un puntero es una dirección de espacio usuario que no puede ser directamente referenciada. Para la compilación normal, `_user` no tiene ningún efecto, pero puede ser utilizado por un software de verificación externo para encontrar el uso incorrecto de las direcciones de espacio usuario.

struct module *owner;

El primer campo file_operations no es una operación en absoluto; Es un puntero al módulo que "posee" la estructura. Este campo se usa para evitar que el módulo se descargue mientras sus operaciones están en uso. Casi todo el tiempo, simplemente se inicializa a THIS_MODULE, una macro definida en <linux / module.h> .

ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);

Se utiliza para recuperar datos del dispositivo. Un puntero nulo en esta posición hace que la llamada al sistema de lectura falle con -EINVAL ("Argumento no válido"). Un valor de retorno no negativo representa el número de bytes leídos correctamente (el valor de retorno es un tipo de "tamaño con signo", generalmente el tipo de entero nativo para la plataforma de destino).

ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);

Envía datos al dispositivo. Si es NULL, se devuelve -EINVAL al programa llamando a la llamada al sistema de escritura. El valor de retorno, si no es negativo, representa el número de bytes escritos correctamente.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

La llamada al sistema ioctl ofrece una forma de emitir comandos específicos del dispositivo (como formatear una pista de un disquete, que no es de lectura ni de escritura). Además, el kernel reconoce algunos comandos ioctl sin hacer referencia a la tabla fops. Si el dispositivo no proporciona un método ioctl, la llamada al sistema devuelve un error para cualquier solicitud que no esté predefinida (-ENOTTY, "No existe tal ioctl para dispositivo").

int (*open) (struct inode *, struct file *);

Aunque esta es siempre la primera operación realizada en el archivo del dispositivo, el controlador no está obligado a declarar el método correspondiente. Si esta entrada es NULL, la apertura del dispositivo siempre tiene éxito, pero su controlador no recibe una notificación.

int (*release) (struct inode *, struct file *);

Esta operación se invoca cuando se libera la estructura file. Al igual que open, el lanzamiento puede ser NULL.

Sistema de archivo /proc

En Linux, existe un mecanismo adicional para que el kernel y los módulos del kernel envíen información a los procesos, este es el sistema de archivos /proc. Originalmente diseñado para permitir un fácil acceso a la información sobre los procesos (de ahí su nombre), ahora es utilizado por cada bit del kernel que tiene algo interesante que reportar, como /proc/modules que proporciona la lista de módulos y /proc/meminfo proporciona las estadísticas de uso de memoria [10].

El método para usar el sistema de archivos proc es muy similar al que se usa con los controladores de dispositivo: se crea una estructura con toda la información necesaria para el archivo /proc, incluidos los punteros a cualquier

función del controlador. Luego, al cargar un módulo la función init_module registra la estructura con el kernel dinámicamente y cleanup_module la anula.

La razón por la que se utiliza un registro dinámico (es decir en tiempo de ejecución) sobre el directorio /proc, se debe a que no queremos establecer números MAJOR y MINOR fijos de antemano para el inodo, sino permitir que el kernel determine el número en tiempo de ejecución y así evitar cualquier conflicto con el sistema. Los sistemas de archivos normales se encuentran en un disco, en lugar de la memoria principal (que es en donde se encuentra el sistema de archivos /proc), y en ese caso el número de inodo es un puntero a una ubicación de disco donde se encuentra el índice de nodo del archivo (i-nodo). El i-nodo contiene información sobre el archivo, por ejemplo los permisos del archivo, junto con un puntero a la posición del disco donde los datos del archivo se pueden encontrar.

En los archivos de dispositivo sobre el directorio /proc también podemos leer y escribir utilizando un módulo asociado a dicho archivo. En la lectura se lee el buffer desde el modo kernel y se lo mostramos al usuario por ejemplo utilizando la función copy_to_user. Para la escritura, como los datos que provienen del usuario, se tiene que importar datos desde el espacio de usuario al espacio kernel, por lo que se puede utilizar copy_from_user.

Como se verá en la programación de los módulos usando el sistema de archivos /proc hay varias formas de operar con los archivos de dispositivo asociados al módulo, una es utilizando una estructura proc_entry como una entrada al sistema de archivos /proc (utilizando las operaciones sobre el archivo a través de esta entrada) y la segunda forma es como se vio en los módulos de dispositivos de carácter asociando el archivo de dispositivo (en este caso en el directorio /proc) a una interfaz de operaciones file_operations, esto se logra utilizando el método proc_create encargado de crear el archivo de dispositivo en el sistema /proc y asociarlo a la interfaz de operaciones además de establecer los números mayor y menor dinámicamente.

En Linux, hay un mecanismo estándar para el registro de sistema de archivos. Debido a que cada sistema de ficheros tiene que tener sus propias funciones para manejar operaciones de inodos y archivos, hay una estructura especial para mantener los punteros a todas estas funciones, conocida como struct inode_operations, que incluye un puntero a una struct file_operations. En /proc, cuando registramos un nuevo fichero, se nos permite especificar qué estructura inode_operations se utilizará para acceder a ella. Con este es el mecanismo, una estructura inode_operations incluye un puntero a una estructura file_operations que incluye punteros a funciones procfs_read y procfs_write por ejemplo.

Otro punto interesante aquí es la función module_permission. Esta función es llamada cuando un proceso intenta hacer algo con el archivo del /proc, y puede decidir si se debe permitir o no el acceso. En este momento sólo se basa en la operación y el identificador de usuario del usuario actual (como disponibles en el actual, un puntero a una estructura que incluye información sobre el proceso actualmente en ejecución), pero podría basarse en lo que queramos, como otros procesos que estén utilizando el mismo archivo, la hora del día o la última entrada que recibimos.

Métodos de lectura y escritura

Ambos métodos de lectura y escritura realizan tareas similares, es decir, copiando datos desde y hacia el programa de la aplicación. Por lo tanto, sus prototipos son bastante similares, y vale la pena presentarlos al mismo tiempo:

```
ssize_t read(struct file *filp, char __user *buff,  
            size_t count, loff_t *offp);  
ssize_t write(struct file *filp, const char __user *buff,  
             size_t count, loff_t *offp);
```

Para ambos métodos, filp es el puntero de file y count es el tamaño de la transferencia de datos solicitada. El argumento buff apunta al búfer del usuario que contiene los datos escritos por el usuario o al búfer vacío donde se deben colocar los datos que leerá el usuario. Finalmente, offp es un puntero a un objeto de "tipo de

desplazamiento largo" que indica la posición del archivo al que el usuario está accediendo. El valor de retorno es un "tipo de tamaño con signo";

Se reitera que el argumento buff de los métodos de lectura y escritura es un puntero de espacio usuario. Por lo tanto, no puede ser directamente referenciado por el código del kernel. Hay algunas razones para esta restricción:

- Según la arquitectura en la que se esté ejecutando el controlador y la forma en que se configuró el kernel, es posible que el puntero del espacio usuario no sea válido mientras se ejecuta en modo kernel. Puede que no haya una asignación para esa dirección, o podría apuntar a otros datos aleatorios.
- Incluso si el puntero significa lo mismo en el espacio kernel, la memoria del espacio usuario está paginada y la memoria en cuestión puede no residir en la RAM cuando se realiza la llamada al sistema. Intentar hacer referencia a la memoria del espacio usuario directamente podría generar un error de página, que es algo que el código del kernel no puede hacer. El resultado sería un "oops", que daría como resultado la muerte del proceso que realizó la llamada al sistema.
- El puntero en cuestión ha sido suministrado por un programa de usuario, que podría tener errores o ser malicioso. Si el controlador hace una referencia ciega a un indicador suministrado por el usuario, proporciona una puerta abierta que le permite a un programa de espacio usuario acceder o sobrescribir la memoria en cualquier parte del sistema. Si no desea ser responsable de comprometer la seguridad de los sistemas de sus usuarios, nunca podrá anular la referencia directamente de un puntero del espacio usuario.

Obviamente, su controlador debe poder acceder al búfer del espacio usuario para poder realizar su trabajo. Este acceso siempre debe realizarse mediante funciones especiales proporcionadas por el núcleo, sin embargo, para que sea seguro. Introducimos algunas de esas funciones (que se definen en <asm/uaccess.h>) aquí; utilizan alguna magia especial, dependiente de la arquitectura, para garantizar que las transferencias de datos entre el kernel y el espacio usuario se realicen de forma segura y correcta.

El código para leer y escribir de un controlador necesita copiar un segmento completo de datos hacia o desde el espacio de direcciones del usuario. Esta función es ofrecida por las siguientes funciones del kernel, que copian un conjunto arbitrario de bytes y se ubican en el corazón de la mayoría de las implementaciones de lectura y escritura:

```
unsigned long copy_to_user(void __user *to,
                           const void *from,
                           unsigned long count);
unsigned long copy_from_user(void *to,
                            const void __user *from,
                            unsigned long count);
```

Aunque estas funciones se comportan como las funciones normales de memcpy, se debe tener un poco más de cuidado al acceder al espacio usuario desde el código del kernel. Es posible que las páginas de usuarios que se están dirigiendo no estén presentes actualmente en la memoria, y el subsistema de memoria virtual puede poner el proceso en suspensión mientras la página se está transfiriendo a su lugar. Esto sucede, por ejemplo, cuando la página debe recuperarse del espacio de intercambio. El resultado neto para el escritor del controlador es que cualquier función que acceda al espacio usuario debe ser reentrante, debe poder ejecutarse simultáneamente con otras funciones del controlador y, en particular, debe estar en una posición en la que pueda dormir legalmente.

La operación de las dos funciones no se limita a copiar datos desde y hacia el espacio de usuario: también comprueban si el puntero del espacio usuario es válido. Si el puntero no es válido, no se realiza ninguna copia; si se encuentra una dirección no válida durante la copia, por otra parte, solo se copia una parte de los datos. En ambos casos, el valor de retorno es la cantidad de memoria que aún no se ha copiado.

Vale la pena señalar que si no necesita marcar el puntero del espacio usuario, puede invocar __copy_to_user y __copy_from_user en su lugar. Esto es útil, por ejemplo, si sabe que ya verificó el argumento. Tener cuidado,

sin embargo; Si, de hecho, no verifica el puntero del espacio usuario que pasa a estas funciones, puede crear bloqueos del kernel y/o agujeros de seguridad.

En lo que respecta a los métodos del dispositivo real, la tarea del método de lectura es copiar los datos del dispositivo al espacio usuario (usando `copy_to_user`), mientras que el método de escritura debe copiar los datos del espacio usuario al dispositivo (usando `copy_from_user`). Cada llamada del sistema de lectura o escritura solicita la transferencia de un número específico de bytes, pero el controlador es libre de transferir menos datos;

Independientemente de la cantidad de datos que transfieran los métodos, generalmente deben actualizar la posición `*off` del archivo para representar la posición actual del archivo después de completar con éxito la llamada al sistema. El kernel luego propaga el cambio de posición del archivo a la estructura `file` cuando sea apropiado. Los llamadas al sistema `pread` y `pwrite` tienen una semántica diferente, sin embargo; operan a partir de un desplazamiento de archivo dado y no cambian la posición del archivo como se ve en ninguna otra llamada del sistema. Estas llamadas pasan en un puntero a la posición suministrada por el usuario y descartan los cambios que realiza el controlador.

La figura 2.9 representa cómo una implementación de lectura típica utiliza sus argumentos.

Tanto los métodos de lectura como los de escritura devuelven un valor negativo si se produce un error. Un valor de retorno mayor o igual a 0, en cambio, le dice al programa que llama cuántos bytes se han transferido con éxito. Si algunos datos se transfieren correctamente y luego ocurre un error, el valor de retorno debe ser el conteo de bytes transferidos con éxito, y el error no se informa hasta la próxima vez que se llame a la función. La implementación de esta convención requiere, por supuesto, que su controlador recuerde que el error ha ocurrido para que pueda devolver el estado de error en el futuro.

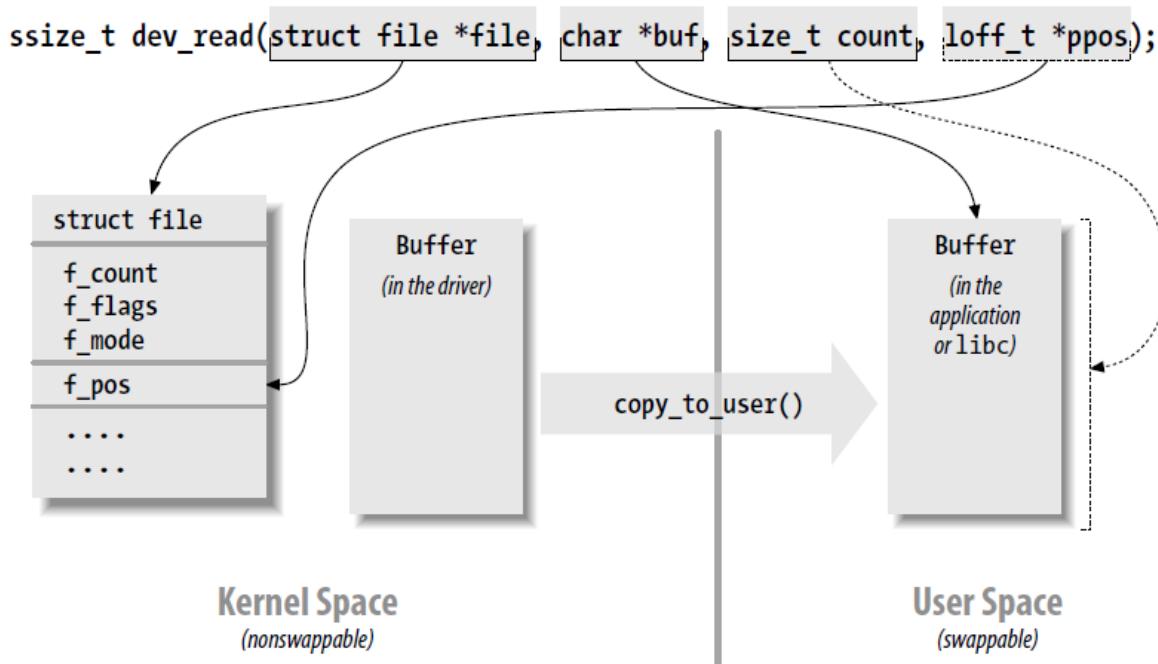


Figura 2.9 Argumentos del método lectura.

Aunque las funciones del kernel devuelven un número negativo para señalar un error, y el valor del número indica el tipo de error que se produjo, los programas que se ejecutan en el espacio usuario siempre se ven -1 como el valor de retorno de error. Necesitan acceder a la variable `errno` para averiguar qué sucedió. El comportamiento de espacio de usuario está dictado por el estándar POSIX, pero ese estándar no establece requisitos sobre cómo funciona el kernel internamente.

2.6. ASIGNACIÓN DE MEMORIA EN EL KERNEL DE LINUX

Linux proporciona una variedad de API para la asignación de memoria en el kernel. Se pueden asignar pequeños fragmentos utilizando las familias kcalloc o kmem_cache_alloc, grandes áreas prácticamente contiguas utilizando vmalloc y sus derivados, o puede solicitar directamente las páginas del asignador de páginas con alloc_pages. También es posible utilizar asignadores más especializados, por ejemplo, cma_alloc o zs_malloc [12].

La mayoría de las API de asignación de memoria utilizan indicadores GFP para expresar cómo debe asignarse esa memoria. El acrónimo de GFP significa “get free page” (traducido como "obtener páginas libres"), en la función de asignación de memoria subyacente.

La diversidad de las API de asignación combinada con los numerosos indicadores de GFP hace que la pregunta "¿Cómo debo asignar memoria?" No sea tan fácil de responder, aunque es muy probable que deba usarla.

Esta sección, muestra las formas de usar la memoria en los controladores de dispositivos y cómo optimizar los recursos de memoria de un sistema. Los módulos no están involucrados en temas de segmentación, paginación, etc., ya que el kernel ofrece una interfaz de administración de memoria unificada para los controladores [9].

2.6.1. LA FAMILIA DE FUNCIONES KMALLOC

La forma más sencilla de asignar memoria es usar una función de la familia kcalloc(). Y, para estar en un tamaño seguro, es mejor usar rutinas que configuren la memoria a cero, como kzalloc(). Si se necesita asignar memoria para un vector, existe kcalloc_array() y kcalloc() [12].

El tamaño máximo de un fragmento que se puede asignar con kcalloc es limitado. El límite real depende del hardware y la configuración del kernel, pero es una buena práctica usar kcalloc para objetos más pequeños que el tamaño de una página.

Kmalloc

La función kcalloc() es similar a la del espacio usuario malloc(), con la excepción del parámetro de indicadores adicionales. La función kcalloc() es una interfaz simple para obtener memoria del kernel en trozos de tamaño de bytes. Si se desea gestionar memoria en tamaño de páginas completas, las interfaces de obtención de páginas disponibles en el kernel de Linux podrían ser una mejor opción, estas se declaran en <linux/gfp.h>. Para la mayoría de las asignaciones de kernel, kcalloc() es la interfaz preferida [6].

La función se declara en <linux/slab.h> (include /linux/slab.h).

```
void * kcalloc ( size_t size , gfp_t flags )
```

La función kcalloc() devuelve un puntero a una región de memoria que tiene al menos size bytes de longitud. Puede asignar más de lo que pidió, aunque no tiene forma de saber cuánto más. Debido a que el asignador de kernel está basado en páginas, algunas asignaciones se pueden redondear para que quepan dentro de la memoria disponible. El kernel nunca devuelve menos memoria de la solicitada. Si el kernel no puede encontrar al menos la cantidad solicitada, la asignación falla y la función regresa NULL. La región de memoria asignada es físicamente contigua. Las asignaciones de kernel siempre tienen éxito, a menos que haya una cantidad insuficiente de memoria disponible. Por lo tanto, debe verificar NULL después de todas las llamadas kcalloc() y manejar el error adecuadamente. Por ejemplo:

```
struct dog *p;
p = kcalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* handle error ... */
```

Si la llamada kmalloc() es exitosa, p ahora apunta a un bloque de memoria que tiene al menos el tamaño solicitado. La bandera GFP_KERNEL especifica el comportamiento del asignador de memoria al tratar de obtener la memoria para regresar al usuario que llama kmalloc().

Límites de tamaño en la asignación con kmalloc

El kernel gestiona el sistema de memoria físico, que está disponible solo en porciones de tamaño de página. Como resultado, kmalloc se ve bastante diferente de una implementación típica de malloc del espacio usuario. Una técnica de asignación simple, orientada al almacenamiento dinámico, se encontraría rápidamente en problemas; tendría dificultades para trabajar alrededor de los límites de la página. Por lo tanto, el kernel utiliza una técnica especial de asignación orientada a la página para obtener el mejor uso de la memoria RAM del sistema [9].

Linux maneja la asignación de memoria creando un conjunto de conjuntos de objetos de memoria de tamaños fijos. Las solicitudes de asignación se manejan yendo a un grupo que contiene objetos lo suficientemente grandes y devolviendo una porción de memoria completa al solicitante. El esquema de administración de memoria es bastante complejo, y los detalles de la misma no suelen ser tan interesantes para los escritores de controladores de dispositivos.

Sin embargo, lo único que los desarrolladores de controladores deben tener en cuenta es que el kernel puede asignar solo ciertas vectores de bytes predefinidos y de tamaño fijo. Si solicita una cantidad arbitraria de memoria, es probable que obtenga un poco más de lo que solicitó, hasta el doble. Además, los programadores deben recordar que la asignación más pequeña que puede manejar kmalloc es tan grande como 32 o 64 bytes, dependiendo del tamaño de página utilizado por la arquitectura del sistema.

Hay un límite superior al tamaño de los fragmentos de memoria que pueden asignarse por kmalloc. Ese límite varía según la arquitectura y las opciones de configuración del kernel. Si su código debe ser completamente portátil, no puede contar con que sea capaz de asignar algo más de 128 KB. Sin embargo, si necesita más de unos pocos kilobytes, hay mejores maneras que kmalloc para obtener memoria como las interfaces de obtención de páginas o la familia vmalloc podrían ser una mejor opción.

Las banderas GFP

Las banderas GFP controlan el comportamiento de los asignadores. Indican qué zonas de memoria se pueden usar, qué tan difícil debe ser para el asignador encontrar memoria libre, si el espacio usuario puede acceder a la memoria, etc [12].

Todas las banderas están declaradas en <linux/gfp.h>. El archivo <Linux/slab.h> incluye este encabezado [6].

Las banderas se dividen en tres categorías:

- **Modificadores de acción:** estos modificadores especifican cómo se supone que el kernel asigna la memoria solicitada. En ciertas situaciones, solo ciertos métodos pueden emplearse para asignar memoria. Por ejemplo, los manejadores de interrupciones deben indicar al kernel que no se duerma (porque los manejadores de interrupciones no pueden reprogramarse) al asignar memoria.
- **Modificadores de zona:** son los modificadores que especifican desde dónde (desde qué zonas) asignar memoria.
- **Banderas de Tipo:** especifica una combinación de un modificador de acción y un modificador de zona para un determinado tipo de asignación de memoria. Esto simplifica la especificación de múltiples modificadores; En lugar de proporcionar una combinación de modificadores de acción y zona, puede especificar un solo indicador de tipo. El GFP_KERNEL es un indicador de tipo, que se utiliza para el código en el contexto de proceso dentro del kernel.

Modificadores de acción

La Tabla 2.2 es una lista de los modificadores de acción [6].

Bandera	Descripción
<code>__GFP_WAIT</code>	El asignador puede dormir.
<code>__GFP_HIGH</code>	El asignador puede acceder a pools (áreas fijas) de emergencia.
<code>__GFP_IO</code>	El asignador puede iniciar la I/O del disco.
<code>__GFP_FS</code>	El asignador puede iniciar la I/O del sistema de archivos.
<code>__GFP_COLD</code>	El asignador debe utilizar páginas frías en caché.
<code>__GFP_NOWARN</code>	El asignador no imprime advertencias de falla.
<code>__GFP_REPEAT</code>	El asignador repite la asignación si falla, pero la asignación puede fallar.
<code>__GFP_NOFAIL</code>	El asignador repite indefinidamente la asignación. La asignación no puede fallar.
<code>__GFP_NORETRY</code>	El asignador nunca se reintenta si la asignación falla.
<code>__GFP_NOMEMALLOC</code>	El asignador no retrocede en las reservas.
<code>__GFP_HARDWALL</code>	El asignador impone límites de cpuset de "hardwall".
<code>__GFP_RECLAIMABLE</code>	El asignador marca las páginas recuperables.
<code>__GFP_COMP</code>	El asignador agrega compuesto.

Tabla 2.2 Modificadores de acción.

Estas asignaciones se pueden especificar juntas. Por ejemplo:

```
ptr = kmalloc ( size , __GFP_WAIT | __GFP_IO | __GFP_FS );
```

Esta llamada le indica al asignador de página (en última instancia alloc_pages()) que la asignación puede bloquear, realizar I/O y realizar operaciones del sistema de archivos. Esto le da al kernel una gran libertad en cómo puede encontrar la memoria libre para satisfacer la asignación.

Modificadores de zona

Los modificadores de zona especifican de qué zona de memoria debe originarse la asignación. Aunque las asignaciones se pueden cumplir desde cualquier zona, el núcleo prefiere ZONE_NORMAL asegurándose de que las otras zonas tengan páginas libres cuando sean necesarias [6].

Solo hay tres modificadores de zona porque solo hay tres zonas ZONE_NORMAL, como en la Tabla 2.3:

Bandera	Descripción
<code>__GFP_DMA</code>	Asignaciones sólo de ZONE_DMA
<code>__GFP_DMA32</code>	Asignaciones sólo de ZONE_DMA32
<code>__GFP_HIGHMEM</code>	Asignaciones de ZONE_HIGHMEM o ZONE_NORMAL

Tabla 2.3 Arco inhibidor, lector y reset.

- La bandera `__GFP_DMA` obliga al núcleo a satisfacer la petición de ZONE_DMA.

- La bandera **`_GFP_HIGHMEM`** le indica al asignador que satisfaga la solicitud de cualquiera de las zonas, **`ZONE_NORMAL`** (preferentemente) o **`ZONE_HIGHMEM`**.

Si no se especifica ninguno de los indicadores, el kernel cumple con la asignación de cualquiera de las zonas entre `ZONE_DMA` o `ZONE_NORMAL` con una fuerte preferencia para satisfacer la asignación de `ZONE_NORMAL`.

No se puede especificar `_GFP_HIGHMEM` a cualquiera `__get_free_pages()` o `kmalloc()`, debido a que estos devuelven una dirección lógica, y no una estructura page. Es posible que estas funciones asignen memoria que no está asignada actualmente en el espacio de direcciones virtuales del kernel y, por lo tanto, no tiene una dirección lógica. Solo `alloc_pages()` puede asignar memoria alta.

La mayoría de las asignaciones no especificará un modificador de zona porque `ZONE_NORMAL` es suficiente.

Banderas de tipo

Los indicadores de tipo especifican los modificadores de acción y zona requerida para cumplir con un tipo particular de transacción. Por lo tanto, el código del kernel tiende a usar la bandera de tipo correcta y no especifica la miríada de otras banderas que pueda necesitar. Esto es más simple y menos propenso a errores [6].

La Tabla 2.4 es una lista de las banderas de tipo:

Bandera	Descripción
<code>GFP_ATOMIC</code>	La asignación es de alta prioridad y no debe dormir. Esta es la bandera que se debe utilizar en los controladores de interrupción, en las mitades inferiores, mientras se mantiene un spinlock, y en otras situaciones en las que no puede dormir.
<code>GFP_NOWAIT</code>	Al igual que <code>GFP_ATOMIC</code> , excepto que la llamada no se interrumpirá en las agrupaciones de memoria de emergencia. Esto aumenta la probabilidad de que la asignación de memoria falle.
<code>GFP_NOIO</code>	Esta asignación puede bloquear, pero no debe iniciar la I/O del disco. Este es el indicador que se debe utilizar en el código de I/O de bloque cuando no puede causar más I/O de disco, lo que podría provocar una recursión desagradable.
<code>GFP_NOFS</code>	Esta asignación puede bloquear e iniciar la I/O del disco, si es necesario, pero no iniciará una operación del sistema de archivos. Esta es la bandera para usar en el código del sistema de archivos cuando no puede iniciar otra operación del sistema de archivos.
<code>GFP_KERNEL</code>	Esta es una asignación normal y podría bloquear. Esta es la bandera para usar en el código de contexto de proceso cuando es seguro dormir. El kernel hará lo que tenga que hacer para obtener la memoria solicitada por el llamante. Esta bandera debe ser su elección por defecto.
<code>GFP_USER</code>	Esta es una asignación normal y podría bloquear. Este indicador se utiliza para asignar memoria a los procesos del espacio usuario.
<code>GFP_HIGHUSER</code>	Esta es una asignación de <code>ZONE_HIGHMEM</code> y podría bloquear. Este indicador se utiliza para asignar memoria a los procesos del espacio usuario.
<code>GFP_DMA</code>	Esta es una asignación de <code>ZONE_DMA</code> . Los controladores de dispositivos que necesitan memoria compatible con DMA utilizan este indicador, generalmente en combinación con uno de los indicadores anteriores.

Tabla 2.4 Banderas de tipo.

La Tabla 2.5 muestra qué modificadores están asociados con cada marca de tipo:

Bandera	Banderas modificadoras
<code>GFP_NOFS</code>	(<code>__GFP_WAIT</code> <code>__GFP_IO</code>)
<code>GFP_KERNEL</code>	(<code>__GFP_WAIT</code> <code>__GFP_IO</code> <code>__GFP_FS</code>)
<code>GFP_USER</code>	(<code>__GFP_WAIT</code> <code>__GFP_IO</code> <code>__GFP_FS</code>)

GFP_HIGHUSER	(__GFP_WAIT __GFP_IO __GFP_FS __GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

Tabla 2.5 Asociacion banderas de tipo con modificadores.

La mayoría del código usa GFP_KERNEL o GFP_ATOMIC. Independientemente del tipo de asignación, se debe verificar y manejar las fallas.

A continuación, en la Tabla 2.6 se muestra una lista de las situaciones comunes y las banderas a utilizar.

Situación	Solución
Proceso de contexto, puede dormir.	Utilizar GFP_KERNEL.
Procesar el contexto, no puedo dormir	Use GFP_ATOMIC, o realice sus asignaciones con GFP_KERNEL un punto anterior o posterior cuando pueda dormir.
Manejador de interrupciones	Utilizar GFP_ATOMIC.
Softirq	Utilizar GFP_ATOMIC.
Tasklet	Utilizar GFP_ATOMIC.
Necesito memoria DMA-able, puede dormir	Utilizar (GFP_DMA GFP_KERNEL).
Necesito memoria DMA-able, no puedo dormir	Use (GFP_DMA GFP_ATOMIC), o realice su asignación en un punto anterior cuando pueda dormir.

Tabla 2.6 Situaciones y banderas recomendadas.

Ksize

Al igual que la función kmalloc la función se declara en <linux/slab.h> (include /linux/slab.h) [12].

```
size_t ksize( const void * objp )
```

La función kmalloc puede redondear internamente las asignaciones y devolver más memoria de la solicitada. La función ksize() se puede utilizar para determinar la cantidad real de memoria asignada. La persona que llama puede usar esta memoria adicional, aunque inicialmente se especificó una cantidad menor de memoria con la llamada kmalloc. La persona que llama debe garantizar que el parámetro objp apunta a un objeto válido asignado previamente con la función kmalloc() o kmem_cache_alloc(). El objeto no debe ser liberado durante la duración de la llamada.

Kzalloc

Al igual que la función kmalloc la función se declara en <linux/slab.h> (include /linux/slab.h) [12].

```
void * kzalloc( size_t size , gfp_t flags )
```

Esta función asigna memoria igual que kmalloc con la diferencia que se inicializa a cero.

Kmalloc_array y kcalloc

Al igual que la función kmalloc las funciones kmalloc_array y kcalloc se declaran en <linux/slab.h> (include /linux/slab.h) [12]. El primer caso:

```
void * kmalloc_array( size_t n , size_t size , gfp_t flags )
```

Esta función asigna memoria para un vector. De acuerdo a los parámetros:

- **size_t n:** número de elementos.
- **size_t size:** tamaño del elemento
- **gfp_t flags:** el tipo de memoria a asignar (ver las banderas GFP).

El segundo caso

```
void * kcalloc( size_t n , size_t size , gfp_t flags )
```

Esta función es igual que kmalloc_array solo que la memoria asignada se inicializa a cero.

Kfree

La contraparte de la función kmalloc() es kfree(), declarada en <linux/slab.h>:

```
void kfree( const void * ptr )
```

El método kfree() libera un bloque de memoria previamente asignado con kmalloc() o cualquiera de las funciones de asignación de la familia kmalloc.

No llame a esta función en la memoria no asignada previamente con kmalloc() o funciones derivadas, o en la memoria que ya se ha liberado. Si lo hace, se trata de un error que provoca un mal comportamiento, como liberar memoria que pertenece a otra parte del kernel.

Al igual que en el espacio usuario, tenga cuidado de equilibrar sus asignaciones con sus liberaciones para evitar pérdidas de memoria y otros errores. kfree(NULL) se comprueba explícitamente y es seguro.

2.6.2. LA FAMILIA DE FUNCIONES VMALLOC

Para asignaciones grandes, se puede usar vmalloc() y vzalloc(), o solicitar directamente las páginas del asignador de páginas. La memoria asignada por vmalloc no es físicamente contigua y lo mismo sucede para las funciones de la familia vmalloc [7].

Vmalloc, vzalloc y vfree

La función vmalloc() funciona de manera similar a kmalloc(), excepto que vmalloc() asigna memoria que es virtualmente contigua y no necesariamente contigua físicamente. Esto es similar al espacio usuario con malloc(), donde las páginas devueltas por las cuales son contiguas dentro del espacio de direcciones virtuales, pero no necesariamente contiguas en la RAM física [6].

La función vmalloc() garantiza que las páginas sean físicamente contiguas al asignar trozos potencialmente no contiguos de memoria física y al "arreglar" las tablas de la página para asignar la memoria a una parte contigua del espacio de direcciones lógicas.

- Por lo general, solo los dispositivos de hardware requieren asignaciones de memoria físicamente contiguas, porque viven en el otro lado de la unidad de administración de memoria y no entienden las direcciones virtuales.
- Los bloques de memoria utilizados solo por software (p. ej., Buffers relacionados con el proceso) están bien si se utiliza una memoria que es virtualmente contigua. En la programación, nunca se sabe la diferencia. Toda la memoria aparece en el núcleo como lógicamente contigua.

Aunque solo se requiere memoria física contigua en ciertos casos, la mayoría de los códigos del kernel utilizan kmalloc() y no vmalloc() para obtener memoria principalmente por el rendimiento. La función vmalloc(), para hacer contiguas las páginas físicas no contiguas en el espacio de direcciones virtuales, debe configurar específicamente las entradas de la tabla de páginas. Peor aún, las páginas obtenidas a través de vmalloc() deben ser asignadas por sus páginas individuales (porque no son físicamente contiguas), lo que resulta en una TLB thrashing (traducido a golpe de TLB ver dicha sección para más detalle) mucho mayor que la que se ve cuando se usa la memoria asignada directamente. Debido a estas preocupaciones, vmalloc() se utiliza solo cuando es absolutamente necesario (normalmente, para obtener grandes regiones de memoria). Por ejemplo, cuando los módulos se insertan dinámicamente en el kernel, se cargan en la memoria creada a través de vmalloc().

Las funciones de la familia vmalloc() se declara en <linux/vmalloc.h> y se define en mm/vmalloc.c . Su uso es idéntico al de malloc() en el espacio usuario.

```
void *vmalloc(unsigned long size)
```

- La función devuelve un puntero a al menos size bytes de memoria virtualmente contigua.
- En caso de error, la función devuelve NULL.
- La función puede estar inactiva y, por lo tanto, no se puede llamar desde un contexto de interrupción u otras situaciones en las que el bloqueo no está permitido.

Para asignar memoria e inicializarla en cero se debe usar:

```
void *vzalloc(unsigned long size)
```

Para liberar una asignación obtenida vía vmalloc() o cualquier derivado de la familia, se debe usar:

```
void vfree ( const void * addr )
```

- Esta función libera el bloque de memoria a partir de la dirección addr que se asignó previamente a través de vmalloc().
- La función puede dormir y, por lo tanto, no se puede llamar desde el contexto de interrupción.
- No tiene valor de retorno.

Golpe de TLB

Un golpe de TLB se da cuando el búfer de traducción (TLB), que actúa como una caché para la unidad de administración de memoria (MMU) encargada de traducir las direcciones virtuales a direcciones físicas, es demasiado pequeño para el conjunto de páginas de trabajo. El golpe de TLB puede ocurrir incluso si la caché de instrucciones o la caché de datos no están actuando, porque se almacenan en la caché en diferentes tamaños. Las instrucciones y los datos se almacenan en caché en bloques pequeños (líneas de caché), no en páginas completas, pero la búsqueda de direcciones se realiza a nivel de página. Por lo tanto, incluso si los conjuntos de trabajo de código y datos se ajustan a la memoria caché, si los conjuntos de trabajo están fragmentados en muchas páginas, es posible que el conjunto de trabajo de la dirección virtual no se ajuste a la TLB, lo que provoca e golpe de TLB [13].

2.6.3. BUENAS PRÁCTICAS DE ASIGNACIÓN DE MEMORIA

Siempre que se reserve memoria de forma dinámica con cualquiera de las funciones de la familia kmalloc o vmalloc (kmalloc, kzalloc, vmalloc, etc.) se debe verificar que no haya habido errores (verificando que el puntero no sea NULL). Cuando se trata de verificar el valor de un puntero (y sólo en ese caso), se puede usar de forma indistinta 0 ó NULL. Usar uno u otro es cuestión de estilo [14].

Como ya se vio, las funciones de asignación dinámica de memoria devuelven un puntero void. Las reglas de C establecen que un puntero void se puede convertir automáticamente a un puntero de cualquier otro tipo, por lo que no es necesario, pero si recomendado, hacer una conversión de casteo (cast), como en el siguiente ejemplo:

```
/* El puntero void devuelto por malloc es convertido explícitamente a puntero int*/
int *i = (int *)malloc(sizeof(int));
```

Aunque no hay un consenso, muchos programadores prefieren omitir la conversión anterior porque la consideran menos segura.

Tratar de utilizar un puntero cuyo bloque de memoria ha sido liberado con free puede ser sumamente peligroso. El comportamiento del programa queda indefinido: puede terminar de forma inesperada, sobrescribir otros datos y provocar problemas de seguridad. Liberar un puntero que ya ha sido liberado también es fuente de errores.

Para evitar estos problemas, se recomienda que después de liberar un puntero siempre se establezca su valor a NULL.

```
int *i;
i = malloc(sizeof(int));
...
free(i);
i = NULL;
```

2.6.4. KERNEL MEMORY LEAKS

Una pérdida de memoria (más conocido por el término inglés memory leak) es un error de software que ocurre cuando un bloque de memoria reservado no es liberado en un programa de computación. Comúnmente ocurre porque se pierden todas las referencias a esa área de memoria antes de haberse liberado [15].

Dependiendo de la cantidad de memoria perdida y el tiempo que el programa siga en ejecución, este problema puede llevar al agotamiento de la memoria disponible en la computadora.

Este problema se da principalmente en aquellos lenguajes de programación en los que el manejo de memoria es manual (C o C++ principalmente), y por lo tanto es el programador el que debe saber en qué momento exacto puede liberar la memoria. Otros lenguajes utilizan un recolector de basura o conteo de referencias que automáticamente efectúa esta liberación. Sin embargo todavía es posible la existencia de pérdidas en estos lenguajes si el programa acumula referencias a objetos, impidiendo así que el recolector llegue a considerarlos en desuso.

Si la pérdida de memoria está en el kernel, es probable que el sistema operativo falle. Las computadoras sin una administración de memoria sofisticada, como los sistemas integrados, también pueden fallar completamente debido a una pérdida de memoria persistente.

De igual modo, un programa del espacio usuario (o capa de aplicación), un programa del espacio kernel o un programa de un sistema embebido que se codifiquen en C o C++ y gestionen la memoria de manera manual, estarán todos afectados al riesgo de que se existan pérdidas de memoria.

2.7. SINCRONIZACIÓN EN EL KERNEL DE LINUX

En una aplicación de memoria compartida, los desarrolladores deben asegurarse de que los recursos compartidos estén protegidos contra el acceso simultáneo. El kernel no es una excepción. Los recursos compartidos requieren protección contra el acceso simultáneo porque, si varios subprocessos de ejecución acceden y manipulan los datos al mismo tiempo, los subprocessos pueden sobrescribir los cambios de los demás o acceder a los datos mientras se encuentran en un estado incoherente. El acceso simultáneo de datos compartidos a menudo resulta en inestabilidad y es difícil de rastrear y depurar [6].

El término “hilos de ejecución” implica cualquier instancia de ejecución de código. Por ejemplo, esto incluye cualquiera de los siguientes:

- Una tarea en el núcleo.
- Un manejador de interrupciones.
- Una mitad inferior.
- Un hilo del núcleo.

Esta sección puede acortar “subprocesos de ejecución” a simplemente subprocessos. Tener en cuenta que este término describe cualquier código en ejecución.

El soporte de multiprocesamiento simétrico (SMP) se introdujo en el kernel 2.0. El soporte de multiprocesamiento implica que el código del kernel puede ejecutarse simultáneamente en dos o más procesadores. En consecuencia, sin protección, el código en el kernel, que se ejecuta en dos procesadores diferentes, puede acceder simultáneamente a los datos compartidos exactamente al mismo tiempo. Con la introducción del kernel 2.6, el kernel de Linux es preventivo (preemptive). Esto implica que (en ausencia de protección) el programador puede anticipar el código del kernel en prácticamente cualquier punto y reprogramar otra tarea. Hoy, varios escenarios permiten la concurrencia dentro del kernel, y todos ellos requieren protección.

En esta sección se trata los problemas de concurrencia y sincronización, tal como existen en cualquier kernel de un sistema operativo. También se detallan algunos de los mecanismos e interfaces específicos que proporciona el kernel de Linux para resolver problemas de sincronización y prevenir condiciones de carrera.

2.7.1. REGIONES CRÍTICAS Y CONDICIONES DE CARRERA

- Las rutas de código que acceden y manipulan datos compartidos se denominan regiones críticas (también denominadas secciones críticas). Por lo general, no es seguro que varios subprocessos de ejecución accedan al mismo recurso simultáneamente.
- Para evitar el acceso simultáneo durante regiones críticas, el programador debe asegurarse de que el código se ejecute atómicamente, lo que significa que las operaciones se completan sin interrupción, como si toda la región crítica fuera una instrucción indivisible.
- Es un error si es posible que dos subprocessos de ejecución se ejecuten simultáneamente dentro de la misma región crítica. Cuando esto ocurre, se denomina condición de carrera, llamada así porque los subprocessos corrieron para llegar primero. La depuración de las condiciones de la carrera suele ser difícil porque no son fácilmente reproducibles.
- Asegurarse de que se evita la concurrencia insegura y de que las condiciones de carrera no se producen se denomina sincronización.

2.7.2. BLOQUEO

Supongamos que se tiene una cola de solicitudes que necesita servicio y la implementación es una lista enlazada, en la que cada nodo representa una solicitud. Dos funciones manipulan la cola:

- Una función agrega una nueva solicitud a la cola de la lista.
- Una función elimina una solicitud de la cabeza de la cola y la solicita de servicio.

Las solicitudes se agregan, eliminan y atienden continuamente, ya que varias partes del kernel invocan estas dos funciones. Manipular las colas de solicitud ciertamente requiere múltiples instrucciones. Si un hilo intenta leer de la cola mientras otro está manipulando, el hilo de lectura encontrará la cola en un estado inconsistente. Debería ser evidente el tipo de daño que podría ocurrir si el acceso a la cola ocurre simultáneamente. A menudo, cuando el recurso compartido es una estructura de datos compleja, el resultado de una condición de carrera es la corrupción de la estructura de datos.

¿Cómo se puede evitar que un procesador lea de la cola mientras otro procesador lo está actualizando? Aunque es factible que una arquitectura particular implemente instrucciones simples, como aritmética y comparación, atómicamente es absurdo que las arquitecturas proporcionen instrucciones para respaldar las regiones críticas de tamaño indefinido como las que existen en el ejemplo. Lo que se necesita es una forma de asegurarse de que solamente un hilo manipule la estructura de datos a la vez, un mecanismo para evitar el acceso a un recurso mientras que otro hilo de ejecución se encuentra en la región marcada.

Un bloqueo (o cerradura) proporciona tal mecanismo. Los hilos mantienen los bloqueos; Los bloqueos protegen los datos.

- Siempre que hubiera una nueva solicitud para agregar a la cola, el primer hilo obtendría el bloqueo del recurso. Entonces podría agregar la solicitud a la cola de forma segura y, en última instancia, liberar el bloqueo.
- Cuando un hilo quería eliminar una solicitud de la cola, también obtendría el bloqueo. Luego podría leer la solicitud y eliminarla de la cola. Finalmente, liberaría el bloqueo.

Cualquier otro acceso a la cola necesitaría igualmente obtener el bloqueo. Debido a que el bloqueo solo puede mantenerse por un hilo a la vez, solo un hilo puede manipular la cola a la vez. Si aparece un subproceso mientras otro ya lo está actualizando, el segundo subproceso debe esperar a que el primero libere el bloqueo antes de poder continuar. El bloqueo evita la concurrencia y protege la cola de las condiciones de carrera.

Cualquier código que acceda a la cola primero debe obtener el bloqueo relevante. Si aparece otro hilo de ejecución, el bloqueo evita la concurrencia:

Hilo 1	Hilo 2
intenta bloquear la cola	intenta bloquear la cola
conseguido: bloqueo adquirido	fallado: esperando ...
cola de acceso ...	esperando ...
desbloquear la cola	esperando ...
...	conseguido: bloqueo adquirido
	cola de acceso ...
	desbloquear la cola

Tabla 2.7 Bloqueo.

Observe que los bloqueos son consultivos y voluntarios. Los bloqueos son completamente una construcción de programación que el programador debe aprovechar. Nada le impide escribir código que manipule la cola ficticia sin el bloqueo apropiado, pero tal práctica eventualmente resultará en una condición de carrera y corrupción.

Los bloqueos vienen en varias formas y tamaños. Solo Linux implementa un puñado de mecanismos de bloqueo diferentes. La diferencia más significativa entre los diversos mecanismos es el comportamiento cuando el bloqueo no está disponible porque otro hilo ya lo contiene:

- Algunas variantes de bloqueo están ocupadas en espera (gira en un bucle cerrado, se verifica el estado del bloqueo una y otra vez, esperando que el bloqueo esté disponible).
- Otros bloqueos ponen la tarea actual en suspensión hasta que el bloqueo esté disponible.

Como puede observar, el bloqueo no resuelve el problema; simplemente reduce la región crítica a solo el código de bloqueo y desbloqueo: probablemente mucho más pequeño, pero sigue siendo una carrera potencial. Afortunadamente, los bloqueos se implementan mediante operaciones atómicas que garantizan que no exista ninguna carrera. Una sola instrucción puede verificar si se tomó el bloqueo y, si no, aprovecharlo. La forma en que se hace es específica de la arquitectura, pero casi todos los procesadores implementan una prueba atómica y una instrucción de conjunto que prueba el valor de un número entero y lo establece en un nuevo valor solo si es cero. Un valor de cero significa desbloqueado. En la popular arquitectura x86, los bloqueos se implementan utilizando una instrucción similar llamada comparar e intercambiar (compare and exchange).

Causas de la concurrencia

En el espacio usuario, los programas se programan de forma preventiva a voluntad del programador. Debido a que se puede anular un proceso en cualquier momento y se puede programar otro proceso en el procesador, se puede anular involuntariamente un proceso en medio de acceder a una región crítica. Si el proceso recién programado ingresa a la misma región crítica (por ejemplo, si los dos procesos manipulan la misma memoria compartida o escriben en el mismo descriptor de archivo), puede ocurrir una carrera. El mismo problema puede ocurrir con múltiples procesos de un solo hilo que comparten archivos, o dentro de un solo programa con señales, porque las señales pueden ocurrir de forma asíncrona. Este tipo de concurrencia en la que dos cosas no ocurren realmente al mismo tiempo, sino que se intercalan entre sí se denomina “pseudo concurrencia”.

Si tiene una máquina de multiprocesamiento simétrico, dos procesos pueden ejecutarse en una región crítica al mismo tiempo. Eso se llama “concurrencia real”. Aunque las causas y la semántica de la pseudo concurrencia y la real son diferentes, ambas resultan en las mismas condiciones de carrera y requieren el mismo tipo de protección.

El kernel tiene causas similares de concurrencia:

- **Interrupciones:** Una interrupción puede ocurrir de forma asincrónica en casi cualquier momento, interrumpiendo el código que se está ejecutando actualmente.
- **Softirqs y tasklets:** El kernel puede generar o programar un softirq o tasklet en casi cualquier momento, interrumpiendo el código que se está ejecutando actualmente.
- **Kernel preemptive:** Debido a que el kernel es preventivo, una tarea en el kernel puede adelantarse a otra.
- **Dormir y sincronizar con espacio usuario:** Una tarea en el kernel puede dormir y, por lo tanto, invocar al programador, lo que resulta en la ejecución de un nuevo proceso.
- **Multiprocesamiento simétrico:** Dos o más procesadores pueden ejecutar el código del kernel exactamente al mismo tiempo.

Los desarrolladores de kernel necesitan comprender y prepararse para estas causas de concurrencia:

- Es un error importante si se produce una interrupción en medio del código que está manipulando un recurso y el controlador de interrupciones puede acceder al mismo recurso.
- Del mismo modo, es un error si el código del kernel es preventivo mientras accede a un recurso compartido.
- Del mismo modo, es un error si el código en el núcleo duerme mientras se encuentra en medio de una sección crítica.
- Finalmente, dos procesadores nunca deben acceder simultáneamente a la misma pieza de datos.

Con una imagen clara de qué datos necesita protección, no es difícil proporcionar el bloqueo para mantener el sistema estable. Más bien, la parte difícil es identificar estas condiciones y darse cuenta de que necesita algún tipo de protección para evitar la concurrencia.

Diseñar el bloqueo adecuado desde el principio

Implementar el bloqueo real en su código para proteger los datos compartidos no es difícil, especialmente cuando se realiza desde el principio durante la fase de diseño del desarrollo. La parte difícil es identificar los datos compartidos reales y las secciones críticas correspondientes. Esta es la razón por la que el diseño de bloqueo en su código es de suma importancia desde el primer momento, y no como una idea de último momento. Puede ser difícil ingresar, sobreponer e identificar regiones críticas y reconvertir el bloqueo en el código existente. El código resultante a menudo tampoco es bonito. La conclusión de esto es diseñar siempre el bloqueo adecuado en su código desde el principio.

Definiciones de términos seguros de concurrencia

- El código que está a salvo del acceso simultáneo desde un controlador de interrupciones se dice que es seguro contra interrupciones (interrup-safe).
- El código que está a salvo de la concurrencia en máquinas de multiprocesamiento simétrico es seguro para SMP (SMP-safe) .
- El código que está a salvo de la concurrencia con la preferencia del kernel es seguro preventivo o preempt-safe (salvo algunas excepciones, estar seguro para SMP implica estar seguro anticipadamente).

Saber que proteger

Identificar qué datos necesitan protección específica es vital. Dado que cualquier información a la que se puede acceder al mismo tiempo casi con seguridad necesita protección, a menudo es más fácil identificar qué datos no necesitan protección y trabajar desde allí:

- Obviamente, cualquier dato que sea local para un hilo particular de ejecución no necesita protección, porque solo ese hilo puede acceder a los datos. Por ejemplo, las variables automáticas locales (y las estructuras de datos asignadas dinámicamente cuya dirección se almacena solo en la pila) no necesitan ningún tipo de bloqueo porque existen únicamente en la pila del subproceso en ejecución.
- Del mismo modo, los datos a los que solo se accede mediante una tarea específica no requieren bloqueo (porque un proceso puede ejecutarse en un solo procesador a la vez).

¿Qué necesita bloquearse?

La mayoría de las estructuras de datos globales del kernel requieren bloqueo. Una buena regla general es que si otro hilo de ejecución puede acceder a los datos, los datos necesitan algún tipo de bloqueo; Si alguien más puede verlo, entonces necesita bloquearse. Recordar bloquear los datos, no el código.

2.7.3. INTERBLOQUEO

Un interbloqueo es una condición que involucra uno o más subprocesos de ejecución y uno o más recursos, de manera que cada subprocesso espera uno de los recursos, pero todos los recursos ya están retenidos. Todos los hilos se esperan unos a otros, pero nunca avanzan hacia la liberación de los recursos que ya tienen. Por lo tanto, ninguno de los hilos puede continuar, lo que resulta en un interbloqueo.

Una buena analogía es una parada de tráfico de cuatro vías. Si cada automóvil en la parada decide esperar a los otros automóviles antes de ir, ningún automóvil lo hará, y tenemos un punto muerto en el tráfico.

El ejemplo más simple de un interbloqueo es el autobloqueo. Si un hilo de ejecución intenta adquirir un bloqueo que ya tiene, tiene que esperar a que se libere el bloqueo. Pero nunca liberará el bloqueo, porque está ocupado esperando el bloqueo y el resultado es un interbloqueo:

```
adquirir bloqueo  
adquirir bloqueo, de nuevo  
espere a que el bloqueo esté disponible ...
```

Algunos núcleos previenen este tipo de interbloqueo al proporcionar bloqueos recursivos. Estos son bloqueos que un solo hilo de ejecución puede adquirir varias veces. Linux no proporciona bloqueos recursivos. Esto es ampliamente considerado como algo bueno. Si bien los bloqueos recursivos pueden aliviar el problema del autobloqueo, conducen muy fácilmente a una semántica de bloqueo descuidada.

Del mismo modo, considere n hilos y n bloqueos. Si cada hilo mantiene un bloqueo que el otro desea, todos los hilos se bloquean mientras esperan que sus respectivos bloqueos estén disponibles. El ejemplo más común es con dos hilos y dos bloqueos, que a menudo se llama el abrazo mortal (deadly embrace) o interbloqueo de ABBA:

Hilo 1	Hilo 2
adquirir bloqueo A	adquirir bloqueo B
tratar de adquirir bloqueo B	tratar de adquirir un bloqueo A
esperar el bloqueo B	esperar para bloqueo A

Tabla 2.8 Interbloqueo.

Cada hilo está a la espera del otro, y ninguno de los dos liberará su bloqueo original; por lo tanto, ningún bloqueo estará disponible.

La prevención de los escenarios de interbloqueo es importante. Aunque es difícil probar que el código no tiene interbloqueos, puede escribir código sin interbloqueo siguiendo las reglas a continuación:

- Implementar un orden de bloqueo. Los bloqueos anidados siempre deben obtenerse en el mismo orden. Esto evita el abrazo mortal. Documente el orden de bloqueo para que otros lo sigan.
- Prevenir la inanición. Pregúntese:
 - ¿Este código siempre termina?
 - Si no ocurre foo, ¿esperará la barra para siempre?
- No adquirir dos veces el mismo bloqueo.
- Diseñar con simplicidad. La complejidad en su esquema de bloqueo invita a los interbloqueos.

El primer punto es el más importante y vale la pena destacar. Si se adquieren dos o más bloqueos al mismo tiempo, siempre se deben adquirir en el mismo orden.

Cuando los bloqueos se anidan dentro de otros bloqueos, se debe obedecer un orden específico. Es una buena práctica colocar el pedido en un comentario sobre el bloqueo.

El orden de desbloqueo no importa con respecto al interbloqueo, aunque es una práctica común liberar los bloqueos en un orden inverso al que se adquirió.

2.7.4. CONTENCIÓN Y ESCALABILIDAD

El término contención de bloqueo, o simplemente contención, describe un bloqueo actualmente en uso, pero que otro subproceso está intentando adquirir. Un bloqueo que es altamente contendido a menudo tiene subprocesos esperando para adquirirlo. Se puede producir una alta contención porque con frecuencia se obtiene un bloqueo, se mantiene durante mucho tiempo después de que se obtiene, o ambos. Debido a que el trabajo de un bloqueo es serializar el acceso a un recurso, pueden ralentizar el rendimiento del sistema. Un bloqueo altamente contendido puede convertirse en un cuello de botella en el sistema, lo que limita rápidamente su rendimiento. Sin embargo, una solución a alta disputa debe continuar proporcionando la protección de concurrencia necesaria, ya que también se requieren bloqueos para evitar que el sistema se rompa en pedazos.

La escalabilidad es una medida de qué tan bien puede expandirse un sistema. En sistemas operativos, hablamos de la escalabilidad con una gran cantidad de procesos, una gran cantidad de procesadores o grandes cantidades de memoria. Podemos analizar la escalabilidad en relación con prácticamente cualquier componente de una computadora a la que podemos adjuntar una cantidad. Idealmente, duplicar el número de procesadores debería resultar en una duplicación del rendimiento del procesador del sistema, lo que, por supuesto, nunca es el caso.

La escalabilidad de Linux en un gran número de procesadores ha aumentado dramáticamente con el tiempo desde que se introdujo el soporte de multiprocesamiento en el kernel 2.0:

- En los primeros días del soporte de multiprocesamiento de Linux, solo una tarea podía ejecutarse a la vez en el kernel.
- Durante la versión 2.2, esta limitación se eliminó a medida que los mecanismos de bloqueo se hacían más precisos.
- A través de 2.4 y en adelante, el bloqueo del kernel se hizo aún más fino.
- Hoy en día, en el kernel 2.6 de Linux, el bloqueo del kernel es muy preciso y la escalabilidad es buena.

La granularidad del bloqueo es una descripción del tamaño o la cantidad de datos que protege un bloqueo:

- Un bloqueo muy grueso protege una gran cantidad de datos, por ejemplo, un conjunto completo de estructuras de datos de un subsistema.
- Por otro lado, un bloqueo de grano muy fino protege una pequeña cantidad de datos, por ejemplo, solo un elemento en una estructura más grande.

En realidad, la mayoría de los bloqueos se ubican entre estos dos extremos, protegiendo ni un subsistema completo ni un elemento individual, sino quizás una estructura única o una lista de estructuras. La mayoría de los bloqueos se inician de forma bastante aproximada y se hacen más detallados, ya que la contención del bloqueo resulta ser un problema.

La mejora de la escalabilidad generalmente es algo bueno porque mejora el rendimiento de Linux en sistemas más grandes y más potentes. Sin embargo, las "mejoras" de escalabilidad rampantes pueden llevar a una disminución en el rendimiento en máquinas SMP y UP más pequeñas, ya que las máquinas más pequeñas pueden no necesitar este tipo de bloqueo de grano fino pero, sin embargo, tendrán que soportar la mayor complejidad y la sobrecarga.

No obstante, la escalabilidad es una consideración importante. Es importante diseñar bien el bloqueo desde el principio hasta la escala. El bloqueo general de los principales recursos puede convertirse fácilmente en un cuello de botella incluso en máquinas pequeñas. Hay una línea delgada entre el bloqueo demasiado grueso y el bloqueo demasiado fino.

- El bloqueo que es demasiado grueso da como resultado una escalabilidad deficiente si hay una contención de bloqueo alta.
- El bloqueo que es demasiado fino da como resultado una sobrecarga innecesaria si hay poca contención de bloqueo.

Ambos escenarios equivalen a un desempeño bajo. Se debe comenzar de manera simple y crecer en complejidad solo cuando sea necesario. La simplicidad es la clave.

2.7.5. SPIN LOCKS

Como se vio anteriormente, aunque sería bueno si cada región crítica consistiera en un código que no hiciera nada más complicado que incrementar una variable, la realidad es mucho más cruel. En la vida real, las regiones críticas pueden abarcar múltiples funciones. Por ejemplo, a menudo ocurre que los datos deben eliminarse de una estructura, formatearse y analizarse, y agregarse a otra estructura. Toda esta operación debe ocurrir atómicamente; no debe ser posible que otro código lea o escriba en ninguna de las estructuras antes de que se complete la actualización. Debido a que las operaciones atómicas simples son claramente incapaces de proporcionar la protección necesaria en un escenario tan complejo, se necesita un método más general de sincronización: los bloqueos.

El bloqueo más común en el kernel de Linux es el spin lock (bloqueo de giro). Un bloqueo de giro es un bloqueo que se puede mantener como máximo en un subproceso de ejecución. Si un subproceso de ejecución intenta adquirir un bloqueo de giro mientras ya está retenido, lo que se denomina contenido, el subproceso está ocupado, gira, esperando a que el bloqueo esté disponible. Si el bloqueo no se disputa, el hilo puede adquirir inmediatamente el bloqueo y continuar. El giro evita que más de un hilo de ejecución ingrese a la región crítica en un momento dado. El mismo bloqueo se puede utilizar en múltiples ubicaciones, por lo que todos los accesos a una estructura de datos determinada, por ejemplo, pueden protegerse y sincronizarse.

Un spin lock es una analogía de una puerta con llave, los spin locks son similares a sentarse afuera de la puerta, esperando que salga el sujeto que está dentro y te entregue la llave. Si llegas a la puerta y no hay nadie dentro, puedes agarrar la llave y entrar a la habitación. Si llega a la puerta y hay alguien dentro, debes esperar afuera por la llave, verificando su presencia repetidamente. Cuando la habitación está desocupada, puedes agarrar la llave y entrar. Gracias a la llave (lectura de: bloqueo de giro), solo se permite una persona (lectura de: un hilo de ejecución) dentro de la sala (lectura de: región crítica) al mismo tiempo.

El hecho de que un spin lock en disputa provoque que los hilos giren (esencialmente desperdiциando el tiempo del procesador) mientras se espera que el bloqueo esté disponible es relevante. Este comportamiento es el punto del spin lock. No es prudente mantener un spin lock durante mucho tiempo. Esta es la naturaleza del spin lock: un bloqueo liviano de un solo soporte que debe mantenerse por períodos cortos. Un comportamiento alternativo cuando se sostiene el bloqueo es poner el hilo actual en suspensión y activarlo cuando esté disponible. Entonces el procesador puede apagarse y ejecutar otro código. Esto conlleva un poco de sobrecarga, sobre todo los dos interruptores de contexto requeridos para cambiar fuera y regresar al hilo de bloqueo, que es ciertamente mucho más código que el puñado de líneas utilizadas para implementar un spin lock. Por lo tanto, es aconsejable mantener los spin locks por menos de la duración de dos cambios de contexto. Debido a que la mayoría de nosotros tenemos mejores cosas que hacer que medir los cambios de contexto, se debe mantener el bloqueo el menor tiempo posible. Los semáforos, proporcionan un bloqueo que hace que el hilo en espera se duerma, en lugar de girar, cuando se contiene.

Métodos de Spinlocks

Los spin lock dependen de la arquitectura y se implementan en el ensamblaje. El código dependiente de la arquitectura se define en `<asm/spinlock.h>`. Las interfaces utilizables reales se definen en `<linux/spinlock.h>`. El uso básico de un spin lock es:

```
DEFINE_SPINLOCK (mr_lock);  
  
spin_lock (& mr_lock);  
/* región crítica ... */  
spin_unlock (& mr_lock);
```

El bloqueo se puede mantener simultáneamente a lo sumo por un solo hilo de ejecución. En consecuencia, solo se permite un hilo en la región crítica a la vez. Esto proporciona la protección necesaria contra la concurrencia en máquinas de multiprocesamiento. En las máquinas con un solo procesador, los bloqueos se compilan y no existen; simplemente actúan como marcadores para deshabilitar y habilitar la preferencia del kernel. Si se desactiva el kernel preemptive, los bloqueos se compilan completamente.

Se deberá tener sumo cuidado con los spin locks de Linux, ya que a diferencia de las implementaciones de spin lock en otros sistemas operativos y bibliotecas de subprocesamiento, los spin locks del kernel de Linux no son recursivos. Esto significa que si intenta adquirir un bloqueo que ya tiene, girará, esperando a que lo libere. Pero debido a que está ocupado girando, nunca liberará el bloqueo y se generará un interbloqueo.

Los spin locks se pueden usar en los controladores de interrupción, mientras que los semáforos no se pueden usar porque están en reposo. Si se utiliza un bloqueo en un controlador de interrupciones, también debe desactivar las interrupciones locales (solicitudes de interrupción en el procesador actual) antes de obtener el bloqueo. De lo contrario, es posible que un controlador de interrupciones interrumpa el código del kernel mientras se mantiene el bloqueo e intente recuperar el bloqueo. El manejador de interrupciones gira, esperando que el bloqueo esté disponible. Sin embargo, el soporte de bloqueo no se ejecuta hasta que se completa el controlador de interrupciones. Este es un ejemplo del interbloqueo de doble adquisición analizado en la sección 2.6.3. Tener en cuenta que se debe deshabilitar las interrupciones solo en el procesador actual. Si se produce una interrupción en un procesador diferente y gira en el mismo bloqueo, no impide que el soporte del bloqueo (que está en un procesador diferente) eventualmente libere el bloqueo.

El kernel proporciona una interfaz que deshabilita convenientemente las interrupciones y adquiere el bloqueo.

```
DEFINE_SPINLOCK (mr_lock);  
unsigned long flags;  
  
spin_lock_irqsave (& mr_lock, flags);  
/* región crítica ... */  
spin_unlock_irqrestore (& mr_lock, flags);
```

La rutina `spin_lock_irqsave()` guarda el estado actual de las interrupciones, las deshabilita localmente y luego obtiene el bloqueo dado. Por el contrario, `spin_unlock_irqrestore()` desbloquea el bloqueo dado y devuelve las interrupciones a su estado anterior. De esta manera, si las interrupciones se deshabilitaron inicialmente, su código no las habilitaría por error, sino que las mantendría deshabilitadas. Tenga en cuenta que la variable `flags` aparentemente se pasa por valor. Esto se debe a que las rutinas de bloqueo se implementan parcialmente como macros.

Si siempre sabe antes que las interrupciones están habilitadas inicialmente, no es necesario restaurar su estado anterior. Se puede habilitar incondicionalmente en el desbloqueo. En esos casos, `spin_lock_irq()` y `spin_unlock_irq()` son óptimos:

```
DEFINE_SPINLOCK (mr_lock);  
  
spin_lock_irq (& mr_lock);  
/* sección crítica ... */
```

```
spin_unlock_irq (& mr_lock);
```

A medida que el kernel crece en tamaño y complejidad, es cada vez más difícil garantizar que las interrupciones siempre estén habilitadas en cualquier ruta de código dada en el kernel. El uso de spin_lock_irq() no se recomienda, por lo tanto. Si lo usa, será mejor que esté seguro de que las interrupciones se activaron originalmente o la gente se molestará cuando esperan que las interrupciones estén desactivadas.

Otros métodos de Spinlocks

Puede usar el método spin_lock_init() para inicializar un spin lock creado dinámicamente (un indicador spinlock_t al que no tiene referencia directa, solo es un puntero).

El método spin_trylock() intenta obtener el spin lock dado. Si se opone el bloqueo, en lugar de girar y esperar a que se libere el bloqueo, la función devuelve cero inmediatamente. Si logra obtener el bloqueo, devuelve un valor distinto de cero. Similarmente, spin_is_locked() devuelve un valor distinto de cero si el bloqueo dado está actualmente adquirido. De lo contrario, devuelve cero. En ninguno de los dos casos se spin_is_locked() obtiene realmente el bloqueo.

La Tabla 2.9 muestra una lista completa de los métodos estándar de spin locks.

Método	Descripción
spin_lock()	Adquiere el bloqueo dado.
spin_lock_irq()	Desactiva las interrupciones locales y adquiere el bloqueo dado.
spin_lock_irqsave()	Guarda el estado actual de las interrupciones locales, deshabilita las interrupciones locales y adquiere el bloqueo dado.
spin_unlock()	Libera el bloqueo dado.
spin_unlock_irq()	Libera el bloqueo dado y habilitan las interrupciones locales.
spin_unlock_irqrestore()	Libera el bloqueo dado y restaura las interrupciones locales al estado anterior.
spin_lock_init()	Se inicializa dinámicamente spinlock_t dado.
spin_trylock()	Intenta adquirir un bloqueo dado; Si no está disponible, devuelve distinto de cero.
spin_is_locked()	Devuelve distinto de cero si el bloqueo dado se adquiere actualmente, de lo contrario devuelve cero.

Tabla 2.9 Métodos de spinlocks.

2.7.6. SPIN LOCKS LECTOR-ESCRITOR

A veces, el uso del bloqueo se puede dividir claramente en vías de lectura y escritura. Por ejemplo, considere una lista que se actualiza y se busca. Cuando la lista se actualiza (se escribe en), es importante que ninguna otra secuencia de ejecución escriba o lea simultáneamente de la lista. La escritura exige la exclusión mutua. Por otro lado, cuando se busca (se lee) la lista, solo es importante que nada más escriba en la lista. Múltiples lectores concurrentes están seguros siempre y cuando no haya escritores. Los patrones de acceso de la lista de tareas se ajustan a esta descripción. No es sorprendente que un spin lock lector-escritor proteja la lista de tareas.

Cuando una estructura de datos se divide claramente en patrones de uso de lector-escritor o consumidor-productor, tiene sentido usar un mecanismo de bloqueo que proporcione una semántica similar. Para satisfacer este uso, el kernel de Linux proporciona spin locks lector-escritor. Los spin locks lector-escritor proporcionan variantes de lectura y escritura separadas de bloqueo. Uno o más lectores pueden mantener simultáneamente el

bloqueo del lector. El bloqueo del escritor, por el contrario, puede ser mantenido por a lo sumo un escritor sin lectores concurrentes. Los bloqueos de lector-escritor a veces se llaman bloqueos compartidos-exclusivos o concurrentes-exclusivos porque el bloqueo está disponible en forma compartida (para lectores) y exclusiva (para escritores).

El uso es similar que los spin locks. El spin lock lector-escritor se inicializa a través de

```
DEFINE_RWLOCK(mr_rwlock);
```

Luego, en la ruta del código del lector:

```
read_lock (& mr_rwlock);
/* sección crítica (solo lectura) ... */
read_unlock (& mr_rwlock);
```

Finalmente, en la ruta del código del escritor:

```
write_lock (& mr_rwlock);
/* sección crítica (lectura y escritura) ... */
write_unlock (& mr_rwlock);
```

Normalmente, los lectores y los escritores están en rutas de código completamente separadas, como en este ejemplo.

Tener en cuenta que no puede "actualizar" un bloqueo de lectura a un bloqueo de escritura. Por ejemplo, considere este fragmento de código:

```
read_lock (& mr_rwlock);
write_lock (& mr_rwlock);
```

La ejecución de estas dos funciones como se muestra se interrumpirá, ya que el bloqueo de escritura gira, a la espera de que todos los lectores liberen el bloqueo compartido, incluido usted mismo. Si alguna vez necesita escribir, obtenga el bloqueo de escritura desde el principio. Si la línea entre sus lectores y escritores está confusa, podría ser una indicación de que no necesita utilizar los bloqueos lector-escritor. En ese caso, un bloqueo de giro normal es óptimo.

Es seguro que varios lectores obtengan el mismo bloqueo. De hecho, es seguro que el mismo hilo obtenga recursivamente el mismo bloqueo de lectura. Esto se presta a una optimización útil y común. Si solo tiene lectores en controladores de interrupción pero no escritores, puede combinar el uso de los bloqueos de "desactivación de interrupción". Se puede utilizar read_lock() en lugar de read_lock_irqsave() para la protección del lector. Aún debe deshabilitar las interrupciones para el acceso de escritura, a la función write_lock_irqsave(), de lo contrario, un lector en una interrupción podría bloquearse en el bloqueo de escritura retenido. Ver la siguiente tabla para obtener una lista completa de los métodos de spin lock lector-escritor.

Método	Descripción
read_lock()	Adquiere el bloqueo dado para lectura.
read_lock_irq()	Desactiva las interrupciones locales y adquiere un bloqueo dado para lectura.
read_lock_irqsave()	Guarda el estado actual de las interrupciones locales, deshabilita las interrupciones locales y adquiere el bloqueo dado para lectura.
read_unlock()	Libera el bloqueo dado para lectura.
read_unlock_irq()	Libera el bloqueo dado y habilita las interrupciones locales.
read_unlock_irqrestore()	Libera el bloqueo dado y restaura las interrupciones locales al estado anterior.

write_lock()	Adquiere el bloqueo dado para escritura.
write_lock_irq()	Desactiva las interrupciones locales y adquiere el bloqueo dado para escritura.
write_lock_irqsave()	Guarda el estado actual de las interrupciones locales, deshabilita las interrupciones locales y adquiere el bloqueo dado para escritura.
write_unlock()	Libera el bloqueo dado para escritura.
write_unlock_irq()	Libera el bloqueo dado y habilita las interrupciones locales.
write_unlock_irqrestore()	Libera el bloqueo dado y restaura las interrupciones locales al estado anterior.
write_trylock()	Intenta adquirir un bloqueo dado para la escritura; Si no está disponible, devuelve un valor distinto de cero.
rwlock_init()	Inicializa el bloqueo rwlock_t dado.

Tabla 2.10 Métodos spinlock lector-escritor.

Una última consideración importante al usar los spin locks lector-escritor de Linux es que favorecen a los lectores sobre los escritores. Si se mantiene el bloqueo de lectura y un escritor está esperando un acceso exclusivo, los lectores que intenten adquirir el bloqueo continuarán teniendo éxito. El escritor que gira no adquiere el bloqueo hasta que todos los lectores liberan el bloqueo. Por lo tanto, un número suficiente de lectores puede matar de inanición a los escritores pendientes. Es importante tener esto en cuenta al diseñar su bloqueo. A veces este comportamiento es beneficioso; A veces es catastrófico.

Los spin locks proporcionan un bloqueo rápido y sencillo. El comportamiento de giro es óptimo para tiempos de espera cortos y código que no se puede dormir (por ejemplo, el manejador de interrupciones). En los casos en que el tiempo que duerme un subproceso sea prolongado o si es posible que necesite dormir mientras mantiene el bloqueo, para estos casos el uso de semáforos es una solución.

2.8. PATRONES DE DISEÑO EN EL KERNEL DE LINUX

A pesar del hecho de que el núcleo de Linux está escrito principalmente en C, hace un uso amplio de algunas técnicas del campo de la programación orientada a objetos. Los desarrolladores que desean utilizar estas técnicas orientadas a objetos reciben poco apoyo u orientación por parte del lenguaje y, por lo tanto, deben defenderse por sí mismos. Como suele ser el caso, esta es una espada de doble filo. El desarrollador tiene la flexibilidad suficiente para hacer cosas realmente geniales, e igualmente la flexibilidad para hacer cosas no tan geniales, y no siempre queda claro a primera vista cuál es cuál, o más exactamente: dónde se encuentra un enfoque particular en el espectro [16].

En lugar de mirar el lenguaje para proporcionar orientación, un ingeniero de software debe buscar la práctica establecida para descubrir qué funciona bien y qué es mejor evitar. Interpretar la práctica establecida no siempre es tan fácil como a uno le gustaría y el esfuerzo, una vez realizado, vale la pena preservarlo. Para preservar ese esfuerzo por parte de su autor, se presentan a continuación Patrones de Diseño del Kernel de Linux exponiendo, con ejemplos, los patrones de diseño en el Kernel de Linux que afectan a un estilo de programación orientado a objetos.

2.8.1. PASO DE MÉTODOS

A continuación se analizan patrones del área de paso de métodos. A pesar de su aparente sencillez se conduce por diseños valiosos para la investigación.

La gran variedad de estilos de herencia y reglas para su uso en los lenguajes actuales parece sugerir que no existe una comprensión uniforme de lo que realmente significa "orientado a objetos". El término es un poco como "amor": todos piensan que saben lo que significa, pero cuando se entra en detalles, la gente puede descubrir que tienen ideas muy diferentes. Si bien lo que significa estar "orientado" puede no estar claro, lo que entendemos por "objeto" parece estar acordado de manera uniforme. Es simplemente una abstracción que comprende tanto el estado como el comportamiento. Un objeto es como un registro (Pascal) o una estructura (C), con la excepción de que algunos de las declaraciones de campos se refieren a funciones que actúan en los otros campos del objeto. Estos campos de función a veces se refieren a "métodos" [16].

Los patrones de diseño orientados a objetos referidos al paso de métodos para cada objeto utilizado en el kernel se agrupan de acuerdo con los siguientes puntos principales:

- Tabla de funciones virtuales o vtable.
- Punteros de función nulos.
- Vtable con campos que no son punteros de función.
- Combinación de métodos para diferentes objetos.
- Punteros de función incrustados en el mismo objeto.

Cada uno de los patrones que aparecen en los diferentes puntos no son los únicos, sino que se decidió clasificarlos de una manera general de forma que cada uno de ellos pueda derivar un subconjunto de patrones asociados.

La forma más obvia de implementar objetos en C es declarar una "estructura" donde algunos campos son punteros de funciones que toman un puntero de la estructura en sí como su primer argumento. Por ejemplo, la convención de llamada para el método "foo" en el objeto "bar" sería simplemente: bar->foo(bar, ... args); Si bien este patrón se usa en el kernel de Linux, no es el patrón dominante, por lo que dejaremos la discusión hasta analizar los "punteros de función incrustados en el mismo objeto".

Tabla de función virtual o vtable

Como los métodos (a diferencia del estado) normalmente no se cambian por objeto, un enfoque más común y un poco menos obvio es recopilar todos los métodos para una clase particular de objetos en una estructura separada, a veces conocida como tabla de "función virtual" o vtable . El objeto tiene un solo puntero a esta tabla en lugar de un puntero separado para cada método y, en consecuencia, se utiliza menos memoria.

Esto conduce a nuestro primer patrón: una vtable pura es una estructura que contiene solo punteros de función donde el primer argumento de cada uno es un puntero a la estructura del objeto que a su vez contiene un puntero a esta vtable. Algunos ejemplos simples de esto en el kernel de Linux son la estructura file_lock_operations que contiene dos punteros de función, cada uno de los cuales utiliza un puntero a una estructura file_lock como parámetro (que sería la estructura del objeto), lo mismo pasa con la vtable seq_operations que contiene cuatro punteros de función que operan con una estructura struct seq_file enviada por parámetro. Estos dos ejemplos muestran un patrón de nomenclatura obvio: la estructura que contiene un vtable recibe el nombre de la estructura que contiene el objeto (posiblemente abreviado) seguido de "_operations". Por ejemplo si nuestro objeto se declara en la estructura automóvil, la estructura vtable será automóvil_operations.

Veamos un resumen de cómo se conformaría el objeto file_lock con su estructura vtable asociada.

```
/* Archivo: include/linux/fs.h */

/* Estructura de Objeto: file_lock */
struct file_lock {
    .
    .
    . /*CAMPOS DE CLASE DE OBJETO file_lock !!!*/
    .

    /* Puntero a estructura VTABLE */
    const struct file_lock_operations *fl_ops;      /* Callbacks for filesystems */
};

/* Estructura VTABLE de file_lock */
struct file_lock_operations {
    void (*fl_copy_lock)(struct file_lock *, struct file_lock *);
    void (*fl_release_private)(struct file_lock *);
};
```

Para mayor detalle de las estructuras visite los siguientes enlaces:

- Estructura vtable file_lock_operations: [enlace externo](#).
- Estructura vtable seq_operations: [enlace externo](#).

Vtable con campos que no son punteros de función

Mientras que la mayoría de las estructuras de tipo vtable en el kernel contienen exclusivamente punteros de función, hay una minoría significativa que tiene campos de puntero que no son de función. Muchos de estos aparecen en la superficie de manera bastante arbitraria y algunas inspecciones más cercanas sugieren que algunos de ellos tienen un diseño deficiente o un poco pobre y su eliminación mejoraría el código.

Hay una excepción al patrón de "solo funciones" que ocurre repetidamente y proporciona un valor real, por lo que vale la pena analizarlo. Este patrón se ve en su forma más general en struct md़_personality que proporciona operaciones para un nivel RAID de software en particular. En particular, esta estructura contiene un "owner" (propietario o poseedor), un "name" (nombre) y un "list" (lista). El modulo "owner" es el que proporciona la implementación. El identificador "name" es un identificador simple: algunos vtables tienen nombres de cadenas, algunos tienen nombres numéricos, y a menudo se le llama algo diferente como "versión", "familia", "nombre de drv" o "nivel". Pero conceptualmente sigue siendo un nombre. En el presente ejemplo hay dos nombres, una cadena y un número de nivel "int level".

La "list", aunque es parte de la misma funcionalidad, es menos común. La estructura md़_personality tiene una estructura list_head, al igual que struct ts_ops. La estructura file_system_type tiene un puntero simple a la siguiente struct file_system_type. La idea subyacente aquí es que para que cualquier implementación particular de una interfaz (o la definición "final" de una clase) sea utilizable, debe registrarse de alguna manera para que pueda encontrarse. Además, una vez que se ha encontrado, debe ser posible garantizar que el módulo que contiene la implementación no se elimine mientras está en uso.

Parece que hay casi tantos estilos de registro contra una interfaz en Linux como interfaces para registrarse, por lo que encontrar patrones fuertes sería una tarea difícil. Sin embargo, es bastante común que una "vtable" sea tratada como el controlador primario en una implementación particular de una interfaz y tenga un puntero "owner" que se puede usar para obtener una referencia en el módulo que proporciona la implementación.

Por lo tanto, el patrón que encontramos aquí es que una estructura de punteros de función utilizada como "vtable" para el paso de métodos a objetos normalmente debe contener solo punteros de función. Las excepciones requieren una justificación clara. Una excepción común permite un puntero de módulo y otros campos posibles, como un nombre y un puntero de lista. Estos campos se utilizan para admitir el protocolo de registro para una interfaz particular. Cuando no hay un puntero de lista, es muy probable que toda la vtable se trate como de solo lectura. En este caso, la vtable a menudo se declarará como una estructura const y, por lo tanto, podría almacenarse en la memoria de solo lectura.

Para mayor detalle de las estructuras visite los siguientes enlaces:

- Estructura mdk_personality : [enlace externo](#).
- Estructura file_system_type: [enlace externo](#).

Conclusiones de los patrones analizados

Si combinamos todos los patrones que se encuentran en el código del kernel de Linux, encontramos que los punteros de función que operan en un tipo particular de objeto normalmente se recopilan en una tabla asociada directamente con ese objeto, aunque también pueden aparecer:

- En una vtable mixin que recopila funciones relacionadas que pueden seleccionarse independientemente del tipo base del objeto.
- En la vtable para un objeto hecho "padre", se evita la necesidad de un puntero vtable en un objeto muy cargado.
- Directamente en el objeto cuando hay pocos punteros de métodos, se deben adaptar individualmente al objeto en particular.

Estas vtables rara vez contienen algo más que punteros a función, aunque los campos necesarios para registrar la clase de objeto pueden ser apropiados. Permitir que estos punteros de función sean NULOS es una técnica común pero no necesariamente ideal para manejar los valores predeterminados.

Entonces, al explorar el código del Kernel de Linux, hemos encontrado que aunque no está escrito en un lenguaje orientado a objetos, ciertamente contiene objetos, clases, métodos (representados por las vtables) e incluso mezclas y combinaciones caracteres. También contiene conceptos que normalmente no se encuentran en lenguajes orientados a objetos, como delegar métodos de objetos a un objeto "principal".

Es de esperar que la comprensión de estos patrones diferentes y las razones para elegirlos puedan llevar a una aplicación más uniforme de los patrones en el núcleo, y por lo tanto facilitar que un principiante entienda qué patrones se están siguiendo.

2.8.2. ARCHIVOS DE CABECERA

Los archivos de cabecera, también conocidos por header file, en español fichero/archivo (de) cabecera, o include file, en español fichero de inclusión, en ciencias de computación, especialmente en el ámbito de los lenguajes de programación C y C++, al archivo, normalmente en forma de código fuente, que el compilador incluye de forma automática al procesar algún otro archivo fuente. Típicamente los programadores especifican la inclusión de los header files por medio de pragmas al comienzo (head o cabecera) de otro archivo fuente [17].

Un header file contiene, normalmente, una declaración directa de clases, subrutinas, variables u otros identificadores. Aquellos programadores que desean declarar identificadores estándares en más de un archivo fuente pueden colocar esos identificadores en un único header file, que se incluirá cuando el código que contiene sea requerido por otros archivos.

Para minimizar la posibilidad de errores, C adoptó la convención de usar archivos de cabecera para contener las declaraciones. Realice las declaraciones en un archivo de cabecera y luego use la directiva #include en cada archivo .c o de otro archivo de cabecera .h que requiere la declaración. La directiva de inclusión #include es una copia del archivo de cabecera directamente en el archivo de desarrollo .c antes de la compilación.

Protección de inclusión

Normalmente, los archivos de encabezado .h tienen una directiva #include guard (también denominada protección de inclusión) o #pragma once para asegurarse de que no se inserte varias veces el mismo archivo de encabezado en un solo archivo .c. Esta es una directiva de pre compilación y la podemos considerar un patrón de diseño a seguir ya que brinda organización, flexibilidad y soluciona errores de compilación. A continuación se presenta un ejemplo:

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

struct matrix_struct
{
    int var;

    void (*set_var)(matrix_o *, int);
};

#endif /* MY_CLASS_H */
```

Utilizando la protección de inclusión se puede utilizar el archivo de cabecera en cualquier programa de desarrollo .c sea un programa abuelo, padre e hijo, la inclusión se realizará una única vez independientemente del número de archivos fuente que lo utilice.

2.8.3. ESTRUCTURAS OPACAS

Una declaración typedef introduce un nombre que, dentro de su ámbito, se convierte en un sinónimo del tipo especificado por la “declaración de tipo” parte de la declaración [18].

Se puede utilizar declaraciones typedef para construir nombres más cortos o más significativos para tipos ya definidos por el lenguaje o para tipos que se han declarado. Los nombres de typedef permiten encapsular detalles de la implementación que pueden cambiar.

Por el contrario a las declaraciones struct, unión, y enum, la declaración typedef no introduce nuevos tipos, introduce nuevos nombres para tipos existentes.

Los nombres declarados con typedef ocupan el mismo espacio de nombres que otros identificadores (excepto las etiquetas de instrucciones).

Es posible declarar cualquier tipo con typedef, incluidos los tipos de puntero, función y arreglos. Se puede declarar un nombre de typedef para un puntero a un tipo de estructura o de unión antes de definir el tipo de estructura o de unión, siempre y cuando la definición tenga la misma visibilidad que la declaración.

La declaración typedef a menudo se combina con la declaración struct para declarar y dar nombre a tipos definidos por el usuario. Por ejemplo:

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H
```

```

typedef struct matrix_struct matrix_o;

struct matrix_struct
{
    int var;

    void (*set_var)(matrix_o *, int);
    int (*get_var)(void);
};

#endif /* MY_CLASS_H */

// main.c
#include "my_class.h"

int main()
{
matrix_o m1;

m1->set_var(&m1, 10);

printf_s("El valor es: %d \n", m1->get_var());
}

```

De esta forma es posible declarar y utilizar un tipo de estructura o puntero a una estructura sin definir la estructura misma. Con este mecanismo se pueden crear estructuras opacas [19].

En el fichero de cabecera (extensión .h) define un tipo de estructura o puntero a una estructura:

```

// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

typedef struct matrix_struct matrix_o; // alternative 1
typedef struct *matrix_struct matrix_o; // alternative 2

#endif /* MY_CLASS_H */

```

La definición de la estructura puede estar dentro de un fichero de implementación (*.c).

```

// my_class.c

struct matrix_struct
{
    int var;

    void (*set_var)(matrix_o *, int);
    int (*get_var)(void);
};

```

La interfaz en el fichero de cabecera define una estructura o un puntero, que es un tipo de un tamaño conocido. Por eso el compilador puede compilar un programa que usa un puntero aunque el tamaño del elemento a que apunta es desconocido. El usuario de matrix_o accede a la biblioteca meramente con el puntero.

Este encapsulamiento de datos es incluso mejor que el de C++ ya se define la estructura de datos en el fichero de definición. Este fichero no tiene por qué ser visible para el usuario de la estructura opaca, ya que se puede meramente publicar los ficheros binarios correspondientes. En otras palabras, se puede esconder por completo la estructura de datos. Esto no es posible con una clase en C++, ya que la declaración de la clase en el fichero de cabecera debe contener todos los campos de datos, en C se pueden seguir ambos enfoques.

3. ESTUDIO DEL SISTEMA

3.1. ESPECIFICACIÓN DE REQUERIMIENTOS

Esta sección representa la Especificación de requerimientos de software (ERS) del sistema desarrollado. El propósito de esta sección es hacer conocer a los lectores todas las características del sistema desde diferentes puntos de vista y ayudar a entender el sistema para diferentes propósitos particulares, como por ejemplo que el sistema pueda seguir extendiéndose por nuevos proyectos futuros. Todos los lectores que conozcan el marco teórico del presente documento tendrán los fundamentos necesarios para entender la presente ERS del sistema.

3.1.1. ÁMBITO DEL SISTEMA

El sistema de software desarrollado es un device driver de Linux. Su nombre es MatrixmodG. De acuerdo con el marco teórico, el origen del nombre es el siguiente:

- **Matrix:** se debe a que el driver en desarrollo utiliza un conjunto de matrices bases para la estructura de Redes de Petri Generalizadas (RDPG) en el kernel de Linux.
- **mod:** hace referencia a modulo o driver, que es el tipo de sistema en desarrollo dentro del presente proyecto.
- **G:** ya que esta versión del driver es una mejora de un trabajo anterior, denominado Matrixmod el cual solo permitía gestionar Redes de Petri Ordinarias (RDP), en contraparte esta versión permite gestionar RDPG que incluye a las RDP.

El driver MatrixmodG puede gestionar cualquier Red de Petri Generalizada en el kernel de Linux que no supere las 1000 plazas y 1000 transiciones. En esta versión del driver se cubren las extensiones para las RDPG sobre sus arcos, incorporando los arcos inhibidores, arcos lectores y arcos reset, también se cubren algunas extensiones sobre las transiciones como el uso de guardas, las extensiones relacionadas a eventos y semánticas temporales sobre las transiciones no son abarcadas en esta versión del driver, para contar con disponibilidad de la semántica temporal sobre las RDPG, se trabajó en conjunto con el espacio usuario, siendo una alternativa para su simulación. Se tiene que tener en cuenta, de acuerdo con el marco teórico del presente documento, que estas redes preservan el modelo original de las Redes de Petri Ordinarias.

El kernel brinda un rendimiento óptimo independiente de cualquier lenguaje de programación. De esta forma el driver funciona como un conjunto de lógica (creación y gestión de Redes de Petri Generalizadas) cargada en el kernel que puede ser accedida por cualquier lenguaje de programación desde el espacio usuario. Esto es una ventaja significativa, ya que la lógica está en un solo lugar, el kernel, y cualquier lenguaje puede hacer uso de dicha lógica mediante llamadas al sistema aprovechando el rendimiento del kernel.

La gestión de una RDPG en el kernel de Linux a través del driver MatrixmodG, se logra mediante el uso de un conjunto de comandos enviados desde el espacio usuario. Usar los comandos desde una terminal es poco práctico y empeora a medida que una RDPG es de mayor tamaño. Para solucionar este inconveniente se provee de una librería en lenguaje C/C++ que permite gestionar las RDPG de una manera automatizada y totalmente transparente para un usuario final.

Proveer de una librería para cada lenguaje de programación es mucho menos trabajo que realizar la lógica completa de las RDPG en cada lenguaje de programación en particular, manteniéndose de esta forma un óptimo rendimiento desde el kernel de Linux.

Trabajar sobre el kernel de Linux tiene sus riesgos y estos pueden dañar el sistema operativo en cuestión principalmente en las primeras fases de desarrollo del sistema. Para reducir los posibles daños que podrían ocurrir por estos riesgos, se trabajó sobre una máquina virtual (VM) cuyas características son las siguientes:



Figura 3.1 Características de máquina virtual.

De esta forma al final de los procesos de desarrollo y pruebas se garantiza un producto final de calidad y con menos riesgos de dañar el sistema operativo, por lo que luego de estos procesos ya será más recomendable utilizar el sistema desarrollado en un sistema operativo Linux sobre una maquina física real.

3.1.2. CARACTERÍSTICAS DE LOS USUARIOS

El driver MatrixmodG se desarrolla para que pueda ser utilizado por cualquier usuario que conozca la teoría de las Redes de Petri Generalizadas y tenga los conocimientos básicos de programación en la capa de aplicación o espacio usuario. De acuerdo con los objetivos específicos se busca que los programadores del lenguaje C/C++ tengan una herramienta para gestionar RDPG en sus programas simplemente instalando el driver a su sistema y adicionando la librería que funciona como la interfaz necesaria para conectar un usuario con el driver MatrixmodG. A un futuro esta misma idea se puede llevar a los programadores de cualquier lenguaje del espacio usuario como JAVA, Python, Perl, etc. De una manera simple y eficiente adicionando solamente la librería correspondiente.

3.1.3. DIAGRAMAS DE CASOS DE USO

El diagrama de casos de uso actúa como un foco para la descripción de los requerimientos funcionales del usuario. Describe las relaciones entre los requerimientos funcionales, los usuarios y los componentes principales [20].

¿Cuál será el comportamiento del driver matrixmodG? La respuesta a esta pregunta define los casos de uso del sistema a desarrollar y brindará una ayuda para la especificación de los requerimientos funcionales.

Diagrama general de casos de uso

El diagrama general de los casos de uso del driver MatrixmodG se muestra a continuación.

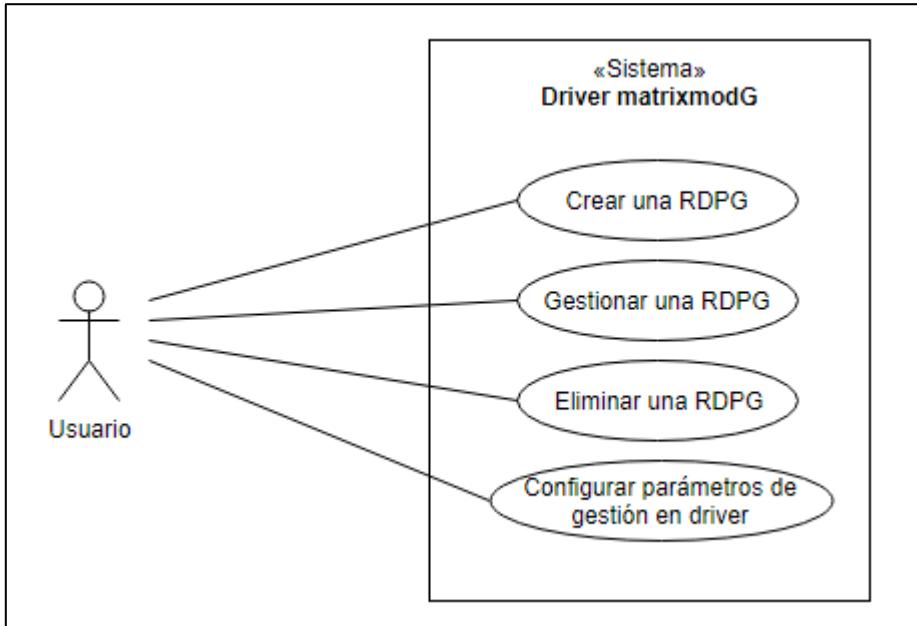


Figura 3.2 Diagrama general de los casos de uso de matrixmodG.

Para entender los casos de uso del driver matrixmodG en la interacción con los actores del sistema se proporcionan un conjunto de tablas complementarias por cada caso de uso.

Tabla 3.1 - CU1: Crear una RDPG

Actores	Usuario.
Descripción	Un usuario puede crear una RDPG en el kernel de Linux con el driver matrixmodG. Para la creación de la RDPG es necesario indicar el número de plazas y transiciones para reservar solo la memoria necesaria. Además se deben asignar los valores iniciales de los componentes bases de la RDPG (mII, mIH, mIR, mIRe y vMI).
Datos de entrada	Número de plazas y transiciones. Valores iniciales de los componentes bases para la creación de una RDPG.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Para finalizar la creación de una RDPG en el kernel de Linux mediante el driver se debe enviar un comando de confirmación para que se establezcan correctamente el resto de componentes de la RDPG. Ver los diagramas específicos de casos de uso para entender los detalles.

Tabla 3.2 - CU2: Gestionar una RDPG

Actores	Usuario.
Descripción	Un usuario puede gestionar una RDPG cargada en el kernel de Linux con el driver matrixmodG. Las gestiones en una RDPG que puede realizar un usuario son el disparo de cualquiera de las transiciones, la solicitud del estado actual de la RDPG y solicitar información asociada a la RDPG cargada en el kernel.
Datos de entrada	De acuerdo a cualquiera de las alternativas de gestión en una RDPG los datos requeridos pueden ser el número de transición a disparar, los componentes que conforman el estado completo de la RDPG y características sobre la información necesaria sobre la RDPG.
Evento	Para indicar el tipo de gestión sobre la RDPG se envía un comando de escritura al archivo de dispositivo del driver matrixmodG. Para obtener información desde el driver hacia el

	espacio usuario se envía un comando de lectura al archivo de dispositivo. En ambos casos los comandos son emitidos por el usuario.
Respuesta	Ante un evento de escritura al archivo de dispositivo el driver notifica en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito. Ante un evento de lectura al archivo de dispositivo el driver devuelve en la salida estándar (STDIN) o en una cadena de caracteres del espacio usuario los datos solicitados. Ante cualquier error de lectura será notificado por la salida de diagnóstico del kernel.
Comentarios	Ver los diagramas específicos de casos de uso para entender los detalles.

Tabla 3.3 - CU3: Eliminar una RDPG

Actores	Usuario.
Descripción	Un usuario puede eliminar una RDPG en el kernel de Linux con el driver matrixmodG. La eliminación de la RDPG elimina de manera completa todos los componentes que conforman la misma y libera la memoria reservada en el kernel.
Datos de entrada	No hay datos de entrada para eliminar una RDPG.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Ninguno.

Tabla 3.4 - CU4: Configurar parámetros de gestión en driver

Actores	Usuario.
Descripción	Un usuario puede configurar parámetros de las formas en que el driver matrixmodG gestiona las RDPG en el kernel de Linux. Estas configuraciones se relacionan a las diferentes alternativas de reservar memoria y al cambio de la presentación de los datos del estado de la RDPG.
Datos de entrada	De acuerdo a las alternativas de configuración los datos de entrada pueden ser un número identificativo del método de asignación de memoria o el número de plaza o transición desde el que se desea presentar el estado de la RDPG. Se puede enviar un número entero cuando se desea cambiar la cantidad de plazas o transiciones a ver de un componente.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Las diferentes configuraciones al driver para la gestión de la RDPG se corresponderán con un comando de escritura diferente asociado a cada operación. Ver los diagramas específicos de casos de uso para entender los detalles.

Diagramas específicos de casos de uso

Algunos casos de uso del diagrama general permiten incluir otros casos de uso que nos muestran más detalles. A continuación se muestran los detalles de los casos de uso que describen pasos o alternativas que un actor puede realizar sobre un caso de uso general.

En el diagrama de casos de usos de la figura 3.3, se detalla el caso de uso “Crear una RDPG” con un conjunto de casos de uso particulares. En contraparte para el caso de uso “Eliminar una RDPG” no se detallan casos de uso particulares.

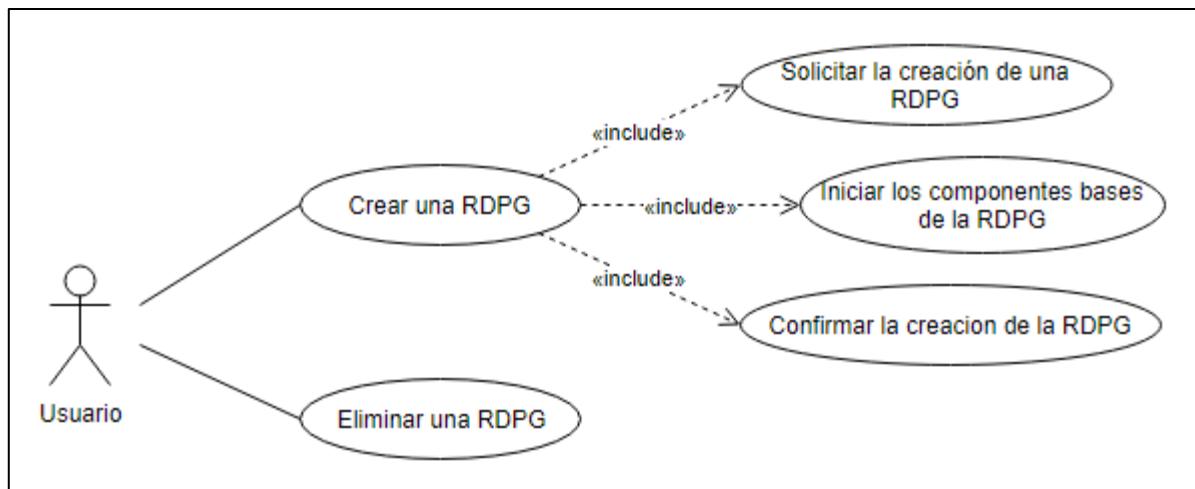


Figura 3.3 Diagrama específico de los casos CU1 y CU3.

Para los casos de uso “Gestionar una RDPG” y “Configurar parámetros asociados a una RDPG” se describen los siguientes casos de uso particulares en las figuras 3.4 y 3.5.

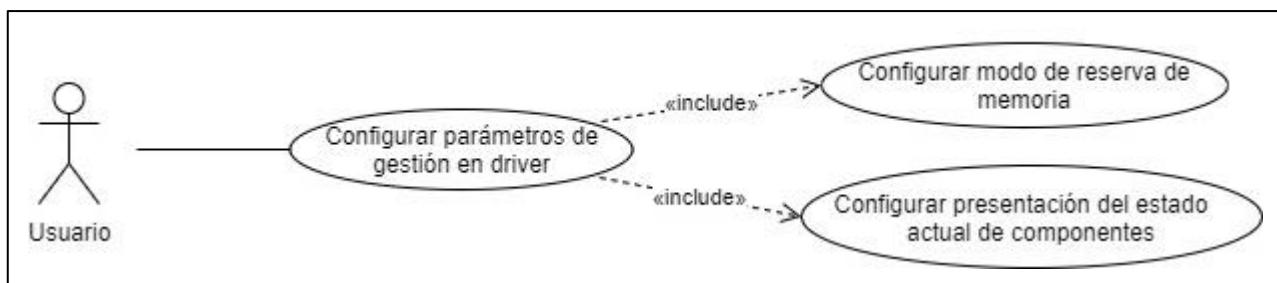


Figura 3.4 Diagrama específico de los casos de uso particulares para CU4.

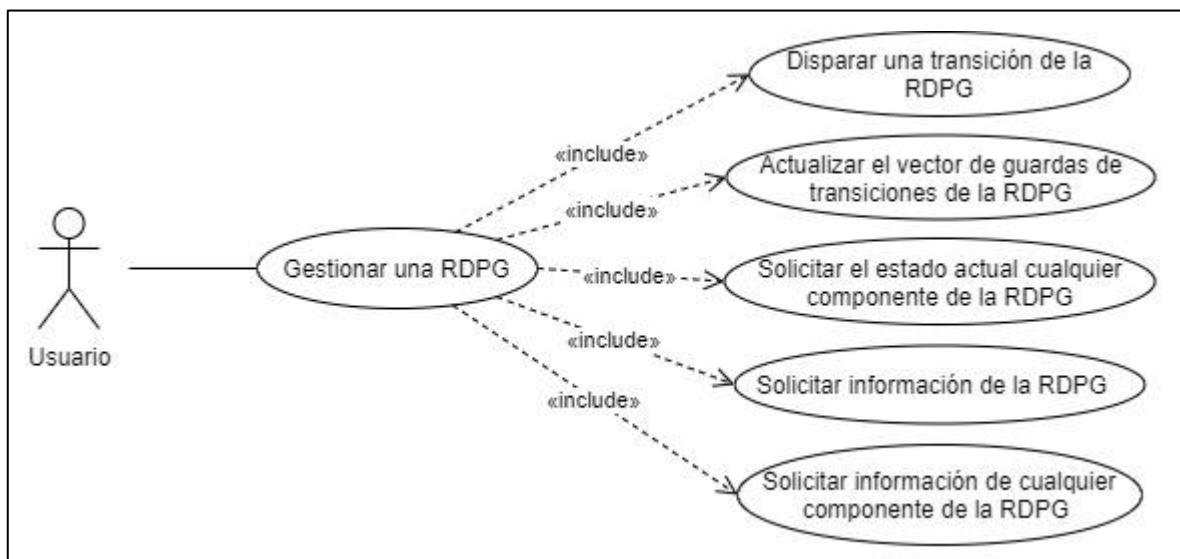


Figura 3.5 Diagrama específico de los casos de uso particulares para CU2.

Se proporcionan las tablas complementarias para el entendimiento de cada caso de uso.

Tabla 3.5 - CU1.1: Solicitar la creación de una RDPG

Actores	Usuario.
Descripción	Para la creación de una RDPG en el kernel, el usuario debe solicitar la creación al driver matrixmodG especificando el número de plazas y transiciones que contiene la red.
Datos de entrada	Número de plazas y transiciones de la RDPG a crear en el kernel.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Después de esta operación ya es posible iniciar los componentes base de la RDPG.

Tabla 3.6 - CU1.2: Iniciar los componentes bases de la RDPG

Actores	Usuario.
Descripción	El usuario debe asignar cada uno de los valores de los componentes base de la RDPG para que esta tenga su estado inicial correspondiente.
Datos de entrada	Cada uno de los valores iniciales de los componentes bases de la RDPG (mII, mIH, mIR, mIRe y vMI).
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Después de esta operación el estado inicial de la RDPG está configurado pero hará falta dar un aviso de confirmación para que se inicialicen correctamente el resto de componentes de la RDPG.

Tabla 3.7 - CU1.3: Confirmar la creación de la RDPG

Actores	Usuario.
Descripción	El usuario debe confirmar el fin de asignación de los componentes base de la RDPG. De esta forma la red sabe cuándo iniciar el resto de los componentes de la misma y es una forma más eficiente ya que se utilizan pocas escrituras en el driver matrixmodG.
Datos de entrada	No son necesarios datos de entrada.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Luego de esta operación la RDPG del kernel de Linux ya está lista para ser gestionada desde el espacio usuario por medio del driver matrixmodG.

Tabla 3.8 - CU2.1: Disparar una transición de la RDPG

Actores	Usuario.
Descripción	Una alternativa en la gestión de la RDPG es que el usuario puede disparar cualquiera de las transiciones de la red donde será necesario especificar el número de la transición a disparar.
Datos de entrada	Número de la transición a disparar.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.

Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Si el disparo de la transición tiene éxito el estado actual de la RDPG cambiara por lo que podrá ser consultado mediante el caso de uso CU2.3.

Tabla 3.9 - CU2.2: Actualizar el vector de guardas de transiciones de la RDPG

Actores	Usuario.
Descripción	En la gestión de una RDPG mediante el driver MatrixmodG el usuario puede actualizar el vector de guardas asociado a las transiciones de la RDPG donde será necesario especificar el número de la transición a actualizar su guarda y el valor que se desea configurar sobre la guarda.
Datos de entrada	Número de la transición sobre la que se desea configurar la guarda y valor en el que se desea configurar la guarda (habilitada o deshabilitada).
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Si el disparo de la transición tiene éxito el estado actual de la RDPG cambiara por lo que podrá ser consultado mediante el caso de uso CU2.3.

Tabla 3.10 - CU2.3: Solicitar el estado actual de cualquier componente de la RDPG

Actores	Usuario.
Descripción	El usuario puede consultar el estado actual de cada componente de la RDPG. Para esto se deberá especificar al driver matrixmodG cada uno de los componentes sobre los cuales se desea conocer su estado. El estado actual completo equivale a solicitar el estado de todos los componentes de la RDPG.
Datos de entrada	El componente sobre el que se desea conocer su estado actual.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG en conjunto con el comando de lectura emitidos secuencialmente por el usuario.
Respuesta	Ante un evento de escritura al archivo de dispositivo el driver notifica en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito. Ante un evento de lectura al archivo de dispositivo del drive, este devuelve una cadena de caracteres con los valores actuales del componente solicitado a la salida estándar o una cadena del espacio usuario.
Comentarios	Cuando se solicitan datos al driver y no existe una RDPG en el kernel o este está en estado inconsistente se devolverá por la salida estándar o cadena de caracteres del espacio usuario “MATRIXMODG: Estado Desconocido.”.

Tabla 3.11 - CU2.4: Solicitar información de la RDPG

Actores	Usuario.
Descripción	El usuario puede solicitar información sobre la RDPG cargada en el kernel. Para esto se deberá especificar el tipo de información requerida como el nombre de la RDPG, la memoria reservada por la RDPG, el número de plazas, número de transiciones, modo de reserva de memoria y el resultado del ultimo disparo efectuado en la red.
Datos de entrada	Tipo de información requerida de la RDPG.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG en conjunto con el comando de lectura emitidos secuencialmente por el usuario.

Respuesta	Ante un evento de escritura al archivo de dispositivo del driver, este notificara en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito. Ante un evento de lectura al archivo de dispositivo del driver, este devolverá la información solicitada en la salida estándar o una cadena de caracteres del espacio usuario.
Comentarios	Ídem CU2.3.

Tabla 3.12 - CU2.5: Solicitar información cualquier componente de la RDPG

Actores	Usuario.
Descripción	El usuario puede solicitar información sobre cualquier componente de la RDPG. Para esto se deberá especificar el componente sobre el que desea obtener información. La información que brindará el driver será nombre del componente, memoria reservada por el componente, número de filas y columnas o número de elementos dependiendo si el componente es un vector o matriz, y el modo de reserva de memoria.
Datos de entrada	Tipo de información requerida de la RDPG.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG en conjunto con el comando de lectura emitidos secuencialmente por el usuario.
Respuesta	Ante un evento de escritura al archivo de dispositivo del driver, este notificara en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito. Ante un evento de lectura al archivo de dispositivo del driver, este devolverá la información solicitada en la salida estándar o una cadena de caracteres del espacio usuario.
Comentarios	Ídem CU2.3.

Tabla 3.13 - CU4.6: Configurar modo de reserva de memoria

Actores	Usuario.
Descripción	Un usuario puede configurar el modo de asignación de memoria que utiliza el driver matrixmodG. Para esto se deberá especificar el número identificativo del modo requerido.
Datos de entrada	Número identificativo del modo de reserva de memoria a usar por el driver matrixmodG.
Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Por defecto el modo de reserva de memoria del driver matrixmodG será el que tiene mayor rendimiento en el kernel de Linux.

Tabla 3.14 - CU4.7: Configurar presentación del estado actual de componentes

Actores	Usuario.
Descripción	Un usuario puede configurar el número de la plaza o la transición desde la cual se desea ver los datos en la presentación de un estado actual de cada uno de los componentes de la RDPG y también puede configurar la cantidad de elementos que desea visualizar sobre ese componente brindado por el driver matrixmodG. Esto se debe a que para redes de gran tamaño mostrar todos los datos no será práctico por lo que la solución es indicar ver los datos desde la plaza y transición requerida y la cantidad adecuada de elementos de un componente.
Datos de entrada	Número de la plaza o transición desde los que se desea obtener los datos en la presentación del estado actual de los componentes de la RDPG. Número de la cantidad de elementos que se desean mostrar sobre los componentes solicitados de la RDPG.

Evento	Envío de comando de escritura al archivo de dispositivo del driver matrixmodG emitido por el usuario.
Respuesta	Notificación del driver en la salida de diagnóstico del kernel de Linux si la operación tuvo o no éxito.
Comentarios	Por defecto los datos de estado de cada componente de la RDPG proporcionados por el driver se presentan con una cantidad fija de plazas y transiciones y con una cantidad de elementos mínima. Es por lo cual es útil modificar el número de inicio de plaza y transición a pesar de mostrar siempre una cantidad fija como así también variar la cantidad de elementos que se desean ver sobre los componentes de la RDPG.

3.1.4. REQUERIMIENTOS FUNCIONALES

¿Qué debe hacer el device driver MatrixmodG? La respuesta a esta pregunta define cada uno de los requerimientos funcionales que posee el desarrollo del driver. Los requerimientos están muy vinculados al tipo de software que se construye, a los usuarios que usaran el sistema y al enfoque utilizado para la definición de los mismos [1].

El enfoque utilizado para la definición de los requerimientos funcionales se basa en dos niveles de requerimientos, estos son:

- Requerimientos funcionales del usuario.
- Requerimientos funcionales del sistema.

Los diferentes niveles de requerimientos son útiles ya que informan sobre el sistema a diferentes lectores. Un nivel tiene más detalles que el otro y esto permite que los diferentes lectores lo usen de diferentes formas.

Requerimientos funcionales del usuario

A continuación se enlistan los requerimientos funcionales de usuario del sistema en desarrollo.

Requerimientos	Descripción
	RF1 El driver debe permitir crear cualquier RDPG en el kernel, que no supere las 1000 plazas y 1000 transiciones.
	RF2 El driver debe permitir eliminar una RDPG previamente cargada en el kernel.
	RF3 El driver debe permitir conocer el estado actual de la RDPG cargada en el kernel.
	RF4 El driver debe permitir operar sobre cualquiera de las transiciones y plazas de la RDPG cargada en el kernel e informar el resultado para las operaciones que lo requieran.
	RF5 El driver debe permitir configurar parámetros relacionados con la gestión de las RDPG en el kernel.
	RF6 El driver debe permitir conocer información general de la RDPG cargada en el kernel.
	RF7 El driver debe permitir conocer información particular de cada uno de los componentes de la RDPG cargada en el kernel.
	RF8 El driver deberá contar con un módulo de control de errores al procesar los comandos enviados por un usuario e informar cuando un comando no es reconocido.
	RF9 El driver deberá contar con un módulo de extracción de datos para los comandos libre de errores filtrados y recibidos desde el módulo de control de errores.

Tabla 3.15 Requerimientos funcionales del usuario.

Requerimientos funcionales del sistema

A continuación se enlistan los requerimientos funcionales de sistema. Cada uno de estos representa el detalle de los requerimientos funcionales del usuario.

Tabla 3.16 - RF1: El driver permite crear cualquier RDPG en el kernel.

Requerimiento	Descripción
RF1.1	La creación de una RDPG se realizará mediante un comando de escritura interpretado por el driver donde se especificara la cantidad de plazas y transiciones que compone la red.
RF1.2	La carga de una RDPG se realizará mediante un comando de escritura interpretado por el driver donde se especificara que componente cargar, posición del vector o posiciones de la matriz, de acuerdo al componente en cuestión y el valor a cargar en dicho vector o matriz.
RF1.3	Se deberá utilizar un aviso de confirmación al driver mediante un comando de escritura interpretado por el mismo para finalizar correctamente la creación de la RDPG cargada en el kernel.
RF1.4	Crear una RDPG cuando ya existe otra RDPG cargada en el kernel, no será posible. Esto lo informara el driver mediante la salida de diagnóstico de errores del kernel de Linux.

Tabla 3.17 - RF2: El driver permite eliminar una RDPG previamente cargada en el kernel.

Requerimiento	Descripción
RF2.1	La eliminación de una RDPG se realizará mediante un comando de escritura interpretado por el driver para eliminar la red del kernel.
RF2.2	Eliminar una RDPG cuando no se cargó previamente en el kernel no será posible. Esto lo informara el driver mediante la salida de diagnóstico de errores del kernel de Linux.

Tabla 3.18 - RF3: El driver permite conocer el estado actual de una RDPG previamente cargada en el kernel.

Requerimiento	Descripción
RF3.1	El driver debe permitir conocer el estado de cada uno de los componentes de la RDPG cargada en el kernel mediante un comando de escritura interpretado por el driver que especificara el componente sobre el cual conocer su estado y en conjunto se usara el comando de lectura.
RF3.2	Conocer el estado de un componente de la RDPG cuando no hay cargada ninguna RDPG en el kernel será informado por el driver mediante la salida de diagnóstico de errores del kernel.
RF3.3	Conocer el estado actual completo de una RDPG implicara llevar a cabo el RF3.1 para cada uno de los componentes de la red.

Tabla 3.19 - RF4: El driver debe permitir operar sobre cualquiera de las transiciones y plazas de la RDPG cargada en el kernel e informa el resultado para las operaciones que lo requieran.

Requerimiento	Descripción
RF4.1	El driver permitirá disparar cualquiera de las transiciones de una RDPG mediante un comando de escritura interpretado por el mismo donde se especificara el número de transición a disparar, informando por la salida de diagnóstico de errores del kernel si el disparo se efectuó o no exitosamente.
RF4.2	El driver no disparara ninguna transición de una RDPG si el comando de escritura interpretado especifica una transición que no existe en la red. Esto será informado por el driver en la salida de diagnóstico de errores del kernel.
RF4.3	El driver permitirá conocer el resultado del último disparo efectuado en la RDPG cargada en el kernel mediante un comando de escritura interpretado por el mismo en conjunto con

	el comando de lectura encargado de brindar la información o también realizando la lectura del vector de vUDT de resultado de ultimo disparo sobre transición.
RF4.4	El driver permitirá actualizar el vector de guardas asociado a cada una de las transiciones de la RDPG cargada en el kernel de Linux mediante un comando de escritura interpretado por el mismo, donde se especificara el número de la transición sobre la que se desea actualizar la guarda y el valor a configurar en la guarda para detectar si se desea habilitar o deshabilitar la transición en cuestión. Se informara por la salida de diagnóstico de errores del kernel si la operación se efectuó o no exitosamente.
RF4.5	El driver permitirá conocer el número de tokens existentes en cualquiera de las plazas por las que se compone la RDPG cargada en el kernel de Linux mediante un comando de escritura interpretado por el mismo donde se especificara el número de la plaza sobre la que se desea conocer el número de tokens. Se informara por la salida de diagnóstico de errores del kernel si la operación se efectuó o no exitosamente.

Tabla 3.20 - RF5: El driver permite configurar parámetros relacionados con la gestión de las RDPG en el kernel.

Requerimiento	Descripción
RF5.1	El driver permitirá el cambio del modo de asignación de memoria para la creación de una nueva RDPG en el kernel mediante un comando de escritura interpretado por el mismo el cual especificara el numero identificador del modo a utilizar en las asignaciones. El cambio se efectúa siempre que no exista ninguna red cargada en el kernel. Es decir que si ya se cargó una red en el kernel no se puede cambiar el modo de asignación hasta que se elimine la red.
RF5.2	El driver permitirá configurar el número de plaza y transición desde la que se desean ver los datos del estado de un componente de la RDPG cargada en el kernel. Mediante el uso de un comando de escritura interpretado por el driver, se especificara el número de la plaza o transición desde la que se desean ver los datos de los componentes de la RDPG.
RF5.3	El driver permitirá configurar el número de datos máximos que se desean ver sobre el estado de un componente de la RDPG cargada en el kernel. Mediante el uso de un comando de escritura interpretado por el driver, se especificara el número de datos que se desean ver para la presentación del estado de cada componente de la RDPG.

Tabla 3.21 - RF6: El driver permite conocer información general de la RDPG cargada en el kernel.

Requerimiento	Descripción
RF6.1	El driver permitirá conocer la información de las características de la RDPG cargada en el kernel como el número de plazas, número de transiciones, nombre de la red y memoria reservada por la red. Cada uno de estos casos de información se solicitará mediante un comando de escritura diferente interpretado por el driver que especificara la característica a ser informada. Conjuntamente se debe utilizar el comando de lectura para la obtención de la información.
RF6.2	Al solicitar la información de las características una RDPG y no se encuentra cargada ninguna red en el kernel se informara el error mediante la salida de diagnóstico de errores del kernel.

Tabla 3.22 - RF7: El driver permite conocer información particular de cada uno de los componentes de la RDPG cargada en el kernel.

Requerimiento	Descripción
RF7.1	El driver permitirá conocer la información de las características de cada uno de los componentes de una RDPG como el nombre del componente y la memoria reservada por dicho componente mediante un comando de escritura interpretado por el driver que especificara la característica a ser informada. Conjuntamente se debe utilizar el comando de lectura para la obtención de la información.

RF7.2	Al solicitar la información de las características de un componente de una RDPG y no se encuentra cargada ninguna red en el kernel se informara el error mediante la salida de diagnóstico de errores del kernel.
--------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Tabla 3.23 - RF8: El driver deberá contar con un módulo de control de errores al procesar los comandos enviados por un usuario e informar cuando un comando no es reconocido.

Requerimiento	Descripción
RF8.1	El módulo de control de errores del driver deberá informar por la salida de diagnóstico del kernel, cuando detecte errores en el formato de datos del comando enviado por un usuario.

Tabla 3.24 - RF9: El driver deberá contar con un módulo de extracción de datos para los comandos libere de errores filtrados y recibidos desde el módulo de control de errores.

Requerimiento	Descripción
RF9.1	El módulo de extracción de datos deberá proporcionar una función para extraer enteros de acuerdo a un formato predefinido valido para la extracción del número de disparo, la posición de elemento de un vector, el seteo de alguna de las variables enteras de la RDPG, etc. Por ejemplo “_100” o “100”.
RF9.2	El módulo de extracción de datos deberá proporcionar una función para extraer dos enteros de acuerdo a un formato predefinido valido para la extracción de los números de plazas y transiciones que tendrá una RDPG. Por ejemplo “10_33”.
RF9.3	El módulo de extracción de datos deberá proporcionar una función para extraer tres enteros de acuerdo a un formato predefinido válido para la extracción de la posición de fila, columna y valor a asignar en una matriz. Por ejemplo “0_200_-1”.

3.1.5. REQUERIMIENTOS NO FUNCIONALES

Los requerimientos no funcionales se clasifican de manera general en requerimientos del producto, requerimientos de la organización y requerimientos externos [21]. Dentro de cada una de estas clasificaciones existen otras clasificaciones, para definir los tipos de requerimientos no funcionales del device driver matrixmodG, de acuerdo con sus características buscadas, se eligieron las siguientes clasificaciones:

- Requerimientos de desarrollo.
- Requerimientos de rendimiento.

Requerimientos no funcionales del usuario

Requerimientos de desarrollo

A continuación se enlistan los requerimientos no funcionales de desarrollo del sistema a desarrollar.

Tabla 3.25: Requerimientos de desarrollo - Descripción

Requerimientos	RNF1	El driver se programara con el paradigma de programación orientado a objetos, se buscara mayor organización de código logrando una alta portabilidad, mantenibilidad y escalabilidad en el código ya que será utilizado en el espacio kernel como también en el espacio usuario.
	RNF2	El driver deberá proteger la RDPG cargada en el kernel con algún mecanismo de exclusión mutua.
	RNF3	La librería del driver se programara con POO tratando la conexión con el driver como un objeto instanciado en espacio usuario. Esto permitirá explotar la característica SMPs que proporciona el driver.

RNF4	La codificación de todas las funciones de los programas fuentes que conforman el driver y su librería de espacio usuario se documentarán con una cabecera que defina su descripción, los parámetros y el retorno de la función.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Requerimientos de rendimiento

A continuación se enlistan los requerimientos no funcionales de rendimiento del sistema a desarrollar.

		Descripción
Requerimientos	RNF4	El driver permitirá crear en el kernel solamente una RDPG.
	RNF5	La RDPG a crear en el kernel no podrá superar las mil plazas y tampoco mil transiciones.
	RNF6	El driver deberá contar con soporte al multiprocesamiento simétrico seguro (SMPs) para la protección de la RDPG como recurso crítico. Esto permitirá que el driver MatrixmodG sea consultado por más de un hilo a la vez (simultánea o paralelamente).
	RNF7	El driver presentará el estado de los componentes de la RDPG (vectores o matrices) por bloques con un máximo de 35 plazas y 35 transiciones.

Tabla 3.26 Requerimientos de rendimiento.

Requerimientos no funcionales del sistema

Requerimientos de desarrollo

Para los requerimientos que lo requieran a continuación se enlistan los requerimientos no funcionales de desarrollo detalladamente.

Tabla 3.27 - RNF1: El driver se programara con el paradigma de programación orientado a objetos.	
Requerimiento	Descripción
RNF1.1	Las RDPG se deberán gestionar como un objeto. Se usarán estructuras que encapsulen sus atributos y funciones disponibles para su posterior configuración y uso durante la instancia del objeto.
RNF1.2	De la misma forma que las RDPG los componentes que hacen a una RDPG deberán gestionarse como objetos. Los componentes de las RDPG a nivel de código son todos los vectores y matrices por lo que se conforma una RDPG. Estos componentes deberán encapsular su conjunto de atributos y funcionalidades.
RNF1.3	Los objetos RDPG y sus objetos componentes deberán utilizar un constructor para su inicialización al momento de instanciarse. Al conocer el número de plazas y transiciones ya se podrán crear los objetos de manera dinámica.
RNF1.4	El método de asignación de memoria dinámica para cada uno de los objetos se podrá cambiar solo antes de la creación de los objetos.

Tabla 3.28 - RNF2: El driver deberá proteger la RDPG cargada en el kernel con algún mecanismo de exclusión mutua.	
Requerimiento	Descripción
RNF2.1	El driver deberá utilizar cualquiera de los mecanismos de exclusión mutua en el kernel de la familia spinlock para proteger las RDPG del multiprocesamiento.
RNF2.2	El driver deberá proporcionar un conjunto de métodos propios de las RDPG que soporten SMPs, haciendo uso de un componente spinlock para la gestión de las mismas de manera segura en el kernel.

3.1.6. INTERFACES EXTERNAS

Para explotar todo el rendimiento y las funcionalidades del driver MatrixmodG es necesario contar con una interfaz de usuario que permita automatización de operaciones entre un usuario y el driver. Proporcionando una interfaz de usuario todo es más simple y sencillo para utilizar el driver.

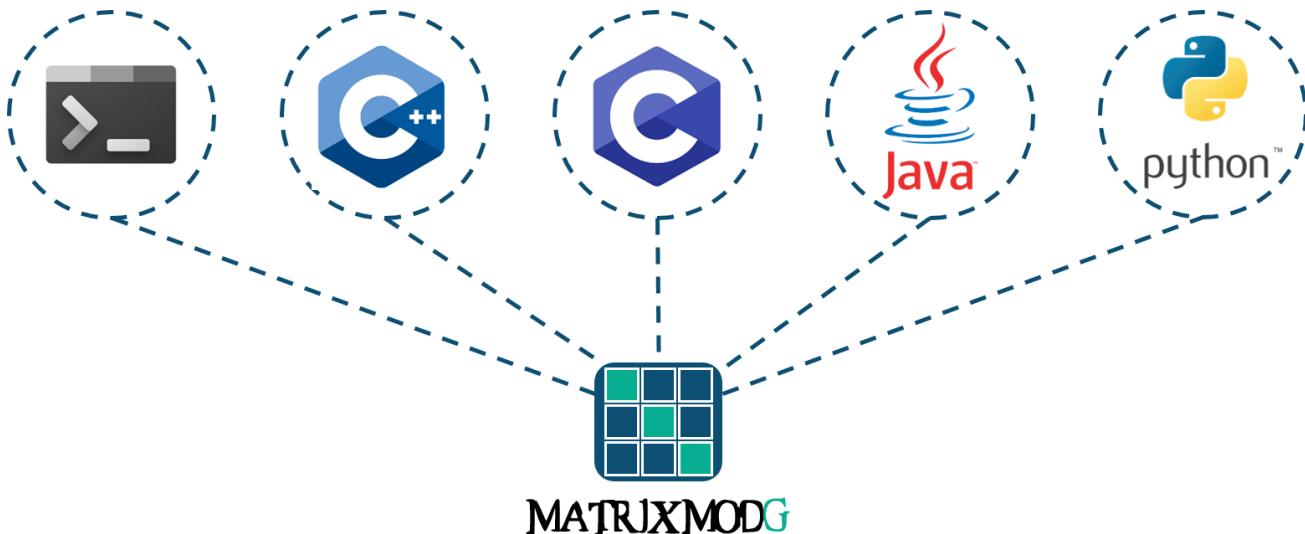


Figura 3.6 Interfaces externas de driver MatrixmodG.

En el presente proyecto se busca gestionar al driver desde el espacio usuario de una manera simple y fácil, por esto, el driver se podrá gestionar desde una terminal a través de la combinación de comandos del sistema y comandos propios del driver. Por otro lado para obtener automatización y eficiencia en la gestión del driver, será necesario realizar una librería de espacio usuario propia de cada lenguaje particular como se muestra por ejemplo en la figura 3.6 para los lenguajes C++, C, JAVA, Python, etc.

Requerimientos funcionales del usuario

Lista de los requerimientos funcionales de usuario de la interfaz de usuario del driver.

Tabla 3.29: Requerimientos funcionales de interfaces externas - Descripción	
Requerimientos	<p>RFI1.1</p> <p>La librería de espacio usuario del driver permitirá importar cualquier RDPG a través de archivos de extensión “.txt” para los componentes bases de una RDPG que son mII, mIH, mIR, mIRE y vMI, los archivos a importar se enviarán como parámetros a una función que recibirá el nombre del archivo de texto.</p> <p>Si se desea importar una RDP solo se deberá importar el componente mII y vMI, el resto de los componentes se deberán indicar con NULL o enviar un string vacío dentro de los parámetros de la función de inicialización del driver.</p> <p>En caso de no ser necesario la importación de la RDPG mediante archivos se permitirá gestionar el driver enviando todos los parámetros de archivos con NULL o un string vacío.</p>
	<p>RFI1.2</p> <p>La librería del driver permitirá configurar cualquiera de los parámetros del driver relacionados a su modo de reserva en el kernel y a su modo de presentación de los elementos de cada componente de la RDPG.</p>
	<p>RFI1.3</p> <p>La librería del driver permitirá realizar todas las operaciones disponibles por el driver ya sean de lectura o escritura.</p> <p>Se permitirá crear una RDPG, eliminarla, agregar valores a los componentes base de la RDPG, confirmar la RDPG, disparar cualquiera de las transiciones de la RDPG, obtener el estado de cualquiera de los componentes de la RDPG, obtener todos los componentes de la RDPG,</p>

obtener la información de la RDPG cargada en el kernel y obtener información de cualquiera de los componentes de la RDPG cargada en el kernel.

3.1.7. ATRIBUTOS DEL SISTEMA

Los atributos que se buscan para que el sistema en desarrollo sea un sistema de software profesional y de calidad son:

- **Escalabilidad:** Su desarrollo se basa en patrones de diseño y POO esto garantiza que el sistema se extienda fácilmente a nuevas funciones y modificaciones creciendo escalablemente en trabajos futuros por la simple lectura de su código y complementando con documentación normalizada.
- **Portabilidad:** Se utilizan patrones de diseño y el paradigma POO para que la lógica del código sea fácil de adaptar a diferentes entornos en los que se tenga interés utilizarla. En este caso la librería de RDPG será utilizada tanto en el kernel de Linux, mediante el driver, como también en el espacio usuario mediante una aplicación de usuario. Esta característica hace al código fácil de migrar a cualquier lenguaje de programación.
- **Mantenibilidad:** Otros de los beneficios de utilizar patrones de diseño y el paradigma POO es que la mantenibilidad del proyecto será sencilla y flexible ante la aparición de diferentes inconvenientes que puedan surgir frente a nuevos escenarios, nuevos cambios y nuevas extensiones que se presenten. El uso de la trazabilidad de requerimientos y pruebas del sistema será un complemento que alimentará el mantenimiento del sistema.
- **Rendimiento:** El sistema se decidió realizar en el kernel de Linux ya que una de las características principales es la búsqueda de un alto rendimiento para la gestión de las RDPG. De acuerdo con esto se trabaja en conjunto con las herramientas que provee el kernel de Linux estudiando cuales son las que demuestran los mejores resultados para incluirlas al desarrollo del driver.
- **Aceptabilidad:** El driver debe ser aceptable al tipo de usuario para el que se diseña y compatible con otros sistemas que se crearon en proyectos previos y que siguen siendo útiles. También su característica de portabilidad y escalabilidad hacen que sea aceptable para ser incluido en trabajos futuros.
- **Eficiencia:** Se busca que el driver sea lo más eficiente posible en cuanto al uso de memoria y tiempo de procesamiento por lo que se trabaja e investiga las mejores alternativas para llevarlas a su desarrollo.

3.2. PLAN DE GESTIÓN DE RIESGOS

Como se detalla en la sección 8.4, el resultado del proceso de gestión de riesgos del proyecto se documenta en el plan de gestión de riesgos y este es el objetivo de esta sección. Se realiza el estudio de los riesgos que enfrenta el proyecto, el análisis de estos, y la información necesaria de cómo gestionar los mismos. Dentro del proceso de gestión de riesgo consideramos un riesgo como algo que es preferible que no suceda.

3.2.1. IDENTIFICACIÓN DE RIESGOS

Sabemos que los riesgos de alguna u otra forma se traslanan a los tres tipos de riesgo generales:

- Riesgos del proyecto.
- Riesgos del producto.
- Riesgos de la empresa u organización.

Tal como se plantea en la sección 8.4.1, ya que estos riesgos amenazan al proyecto, al producto de software que se desarrolla y a la organización respectivamente. Más específicamente se pueden identificar los tipos de riesgos

asociados a cada proyecto en particular. Esto es útil, porque se pueden utilizar para agregarlos a una lista de verificación, lo cual colabora de manera adecuada con la etapa de identificación de riesgos.

Para la etapa de identificación de riesgos se decidió utilizar una lista de verificación de riesgos con los tipos de riesgos que se enlistan a continuación:

- Riesgos personales o de recursos.
- Riesgos de diseño e implementación del software.
- Riesgos de herramientas.
- Riesgos de requerimientos.
- Riesgos de estimación.
- Otros riesgos.

Estos son los tipos de riesgos de lista de verificación elegida para el presente proyecto. Es decir se verifica todos los riesgos que existen de acuerdo a cada uno de los tipos, o viceversa, teniendo un riesgo, a qué tipo de riesgo se puede asociar. Ambos enfoques alimentan la etapa de identificación de riesgos.

Como resultado de la etapa de identificación de riesgos posibles según cada uno de los tipos de riesgos considerados se obtuvo la tabla 2.30.

Riesgos personales o de recursos	
Riesgo Posible (RP)	Descripción y consecuencias
RP1	Equipo de trabajo pequeño.
Riesgos de diseño e implementación del software	
RP2	Imposibilidad de cargar una RDPG en el kernel de Linux mediante un device driver.
RP3	Implementación fuera del paradigma orientada a objetos.
RP4	Fallas en la asignación de memoria dinámica al límite deseado para objetos en el kernel.
RP5	Fallas en el manejo de punteros del espacio kernel.
RP6	Falla al soporte multicore. Problemas de concurrencia en recursos compartidos.
RP7	Presencia de fugas de memoria en la gestión de objetos sobre el kernel de Linux.
RP8	Protección de recursos del kernel con uso de spinlock lector-escritor.
RP9	Ingreso y egreso de datos al driver solo como cadenas de caracteres.
Riesgos de Herramientas	
RP10	Escaso número de herramientas para pruebas unitarias en el kernel.
RP11	Uso de máquinas virtuales con bajo rendimiento.
Riesgos de Requerimientos	
RP12	Poca flexibilidad al cambio de requerimientos definidos e incorporación de nuevos requerimientos al sistema.
Otros riesgos	
RP13	Marco teórico complejo.

Tabla 3.30 Riesgos posibles que afectan el proyecto.

3.2.2. ANÁLISIS DE RIESGOS

De acuerdo con la sección 8.4.2, en la etapa de análisis de riesgo se debe priorizar a los riesgos de mayor probabilidad de ocurrencia y mayor gravedad. Para esto se debe utilizar nuestro juicio y experiencia para determinar la probabilidad y gravedad correcta a cada riesgo haciendo un análisis de las consecuencias. La tabla 3.31 representa dichas relaciones para cada riesgo.

Tabla 3.31: Análisis de Riesgos			
Riesgo Posible (RP)	Consecuencias	Probabilidad de ocurrencia	Gravedad de riesgo
RP1	Afecta la planificación ya que se requiere mayor tiempo de desarrollo.	Moderada	Tolerable
RP2	Cambio de arquitectura del software.	Baja	Tolerable
RP3	Diseño más complejo, código con mayor tendencia a errores.	Moderada	Grave
RP4	Reducción del límite deseado de asignación de memoria dinámica.	Moderada	Tolerable
RP5	Obliga a un reinicio del sistema.	Alta	Catastrófico
RP6	Gestión incorrecta de los datos. Aumento de errores de datos.	Moderada	Grave
RP7	Consumo ineficiente e injustificado de memoria.	Alta	Grave
RP8	Se prioriza los lectores sobre los escritores. Posibilidad de inanición en procesos escritores.	Baja	Grave
RP9	Necesidad de detectar y convertir los datos ASCII al tipo de dato deseado, por ejemplo enteros. Presencia de errores de datos por fallas en la detección o conversión de los estos dentro de la cadena de carácter.	Alta	Catastrófico
RP10	Baja calidad del software resultante.	Moderada	Grave
RP11	Problemas de desarrollo de sistema, baja calidad de producto resultante.	Moderada	Grave
RP12	Rediseño de implementación. Incompatibilidades con diseño definido. Incremento del número de pruebas.	Baja	Grave
RP13	Falta de material bibliográfico. Falta de experiencia.	Baja	Tolerable

Teniendo en cuenta esta tabla ya podemos analizar sobre qué bandas de probabilidad de ocurrencia y gravedad se encuentra cada riesgo. La figura 3.7 muestra esta idea.

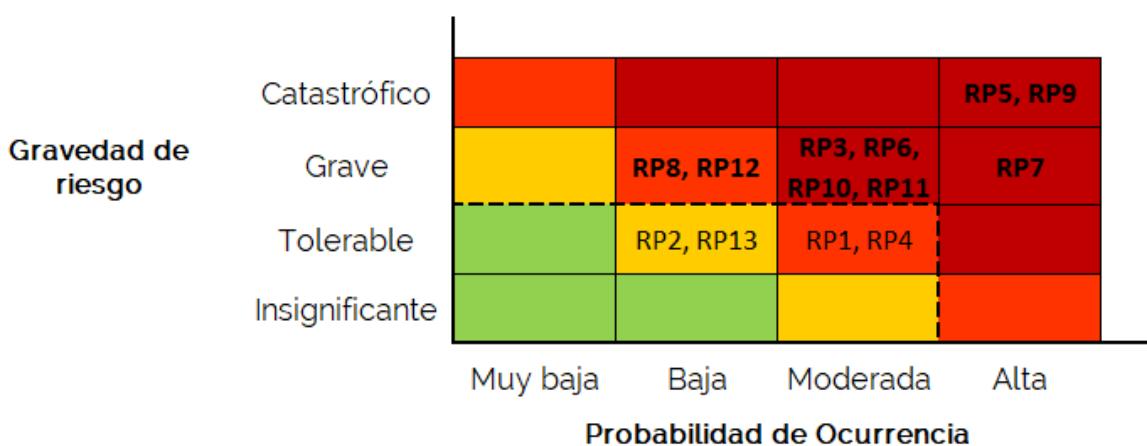


Figura 3.7 Grafico de análisis de riesgos.

Se decidió priorizar los riesgos de banda grave, dentro de la gravedad, y de banda alta, dentro de la probabilidad de ocurrencia. De esta manera podemos decir que los riesgos posibles se convierten en riesgos confirmados para la etapa de planeación del riesgo por ser los riesgos de mayor prioridad por su grado de ocurrencia y gravedad. Como se nota en la figura 3.7 los riesgos que pasan a la próxima etapa son: RP3, RP5, RP6, RP7, RP8, RP9, RP10, RP11 y RP12, el resto de riesgos son descartados. De esta forma se obtiene una nueva tabla con los riesgos confirmados priorizados por el grado de su probabilidad de ocurrencia y gravedad.

Tabla 3.32: Riesgos confirmados		
Riesgo Confirmado (RC)	Ex-Riesgo Posible (RP)	Descripción
RC1	RP5	Fallas en el manejo de punteros del espacio kernel.
RC2	RP9	Ingreso y egreso de datos al driver solo como cadenas de caracteres.
RC3	RP7	Presencia de fugas de memoria en la gestión de objetos sobre el kernel de Linux.
RC4	RP3	Implementación fuera del paradigma orientada a objetos.
RC5	RP6	Falla al soporte multicore. Problemas de concurrencia en recursos compartidos.
RC6	RP10	Escaso número de herramientas para pruebas unitarias en el kernel.
RC7	RP11	Uso de máquinas virtuales con bajo rendimiento.
RC8	RP8	Protección de recursos del kernel con uso de spinlock lector-escritor.
RC9	RP12	Poca flexibilidad al cambio de requerimientos definidos e incorporación de nuevos requerimientos al sistema.

3.2.3. PLANIFICACIÓN DE ESTRATEGIAS DE RIESGOS

En esta etapa nos encargamos de definir las estrategias para la gestión y reducción de la gravedad de cada uno de los riesgos confirmados que se obtuvieron en la etapa de análisis.

Tabla 3.33: Planificación de estrategias de riesgos	
Riesgo	Estrategia
RC1	Preparar un plan de pruebas que permita verificar todas las funcionalidades del sistema desarrollado, garantizando de esta forma que la gestión de los punteros en el espacio kernel para cada una de las funcionalidades disponibles, se realiza de manera adecuada.
RC2	Implementar en el driver un módulo de control de errores sobre los comandos recibidos como entradas al driver MatrixmodG y un módulo de extracción de datos, que será utilizado siempre que un comando esté libre de errores, es decir para todos los comandos que superan el filtro del módulo de control de errores. Para ambos casos se deberá adicionar un conjunto de pruebas que validen el correcto funcionamiento de los mismos dando garantía a la reducción del riesgo.
RC3	Incorporar al plan de pruebas un conjunto de casos de prueba necesarios para garantizar que no existe presencia de fugas de memoria.
RC4	Investigación de patrones de diseño en el kernel de Linux relacionados al paradigma POO y aplicación de estos patrones sobre el desarrollo del sistema con el objetivo de llevar todo el desarrollo a dicho paradigma.

RC5	Incorporación de un conjunto de casos de pruebas necesarios para garantizar el correcto funcionamiento en la gestión de hilos en el espacio kernel y el correcto manejo de los recursos compartido ante problemas de concurrencia.
RC6	Investigación de las diferentes herramientas disponibles para la automatización de pruebas unitarias en el kernel de Linux y formas de testear el código desarrollado. Seleccionar la herramienta que permita, de manera viable, la realización de pruebas unitarias e integrales en el kernel de Linux como su automatización. O seleccionar la forma o técnica adecuada para realizar las pruebas unitarias e integrales de manera automatizada, como por ejemplo, un estilo de trabajo similar al compilador cruzado, haciendo las pruebas en el espacio usuario y eliminando todos sus errores en este espacio dando garantía que en el espacio kernel se reduzca la probabilidad de errores.
RC7	Ánalisis e investigación de las diferentes herramientas disponibles para la gestión de máquinas virtuales y su rendimiento asociado. Utilización de la herramienta con mejor rendimiento obtenido. En caso de ser necesario se reemplazará el sistema físico para obtener uno con mayor potencia para gestionar adecuadamente las VM.
RC8	Incorporación de casos de pruebas que garanticen que no existe presencia de inanición para los procesos escritores ante cada una de las funcionalidades disponibles del sistema.
RC9	Generación de matrices de trazabilidad para gestionar de manera adecuada los requerimientos y su impacto en los diferentes elementos del proyecto ante la presencia de cambios de requerimientos e incorporación de nuevos requerimientos al sistema.

Esta última tabla muestra cuales son las estrategias planteadas para dar garantía a la fiabilidad, seguridad y protección del sistema cuando hay que evitar, tolerar o recuperarse de las posibles fallas. El objetivo en esta etapa es plantear las estrategias para enfrentar a los diferentes riesgos y tales estrategias serán las encargadas de reducir el efecto global de un riesgo en el proyecto o producto. Finalizando esta etapa queda la última etapa implícita de “Monitorización de los riesgos” que deberá llevarse a cabo durante la evolución del sistema analizando continuamente si los riesgos permanecen controlados, aparecen nuevos riesgos o los riesgos actuales incrementan su grado de probabilidad de ocurrencia y/o gravedad.

3.3. TRAZABILIDAD DE REQUERIMIENTOS

Para describir la trazabilidad de los requerimientos del proyecto, hacer el seguimiento y análisis de los cambios en su ciclo de vida se decidió aplicar la técnica de construcción de diferentes matrices de trazabilidad, tal como describe la sección 8.2, permitiendo visualizar la correlación de cada requerimiento con cada elemento del proyecto donde fue de interés mantener dicha trazabilidad.

De acuerdo con el análisis de riesgos y en particular con el riesgo RC9, las matrices de trazabilidad son la estrategia planteada para su reducción. En este proyecto es de interés mantener un fuerte control en la gestión de los casos de prueba asociados a cada requerimiento, se busca lograr una cobertura completa, a nivel de requerimientos (no de código), en número de pruebas asociados a cada uno de ellos. En este nivel se busca que cada requerimiento contenga como mínimo un caso de prueba asociado. La matriz de correlación requerimientos-casos de prueba proporciona muchos beneficios ya que permite mantener y gestionar los requerimientos del proyecto dando un incremento en la capacidad de mantenibilidad del sistema. De esta forma cualquier desarrollador que busque extender funcionalidades o modificar requerimientos sabrá qué casos de prueba se vinculan de manera rápida y directa.

Conocer el origen de cada requerimiento del proyecto es otra de las correlaciones de interés. Para esto se creó la matriz de trazabilidad de la correlación requerimiento-Casos de uso.

3.3.1. MATRICES DE TRAZABILIDAD

Como se describió al comienzo de la sección, para el análisis de la trazabilidad de requerimientos son de interés las siguientes correlaciones:

- **Requerimientos – Bancos de casos de prueba unitarios e integrales:** Esta correlación permitirá visualizar que Bancos (o grupos) de casos de prueba son los que cubren a cada requerimiento funcional. De esta forma se puede saber si todos los requerimientos del sistema están cubiertos por algún grupo de casos de prueba a nivel de prueba unitaria y/o integral.
- **Requerimientos – Casos de prueba de sistema:** Al igual que con los casos de prueba unitarios, esta correlación permitirá visualizar que casos de prueba de sistema son los que cubren a cada requerimiento funcional.
- **Requerimientos – Casos de uso:** Esta correlación permitirá visualizar en qué casos de uso son representados y especificados los requerimientos funcionales del sistema.

Con la construcción y análisis de estas matrices se podrá visualizar información de control y gestión muy importante dentro del ciclo de vida de los requerimientos. Por ejemplo:

- Ante el fallo de cualquier caso de prueba, será posible conocer que requerimientos del sistema son los que se encuentran en riesgo como así también el caso de uso afectado.
- Ante el cambio de un requerimiento será posible conocer qué casos de uso están afectados y cuáles son las pruebas afectadas para su posterior actualización y ejecución.

Correlación Requerimientos vs Casos de prueba

Correlación de Requerimientos vs casos de prueba unitarios e integrales

		Pruebas unitarias e integrales						
		BPU2	BPU3	BPU4	BPU5	BPU6	BPI1	BP2
Requerimientos	RF1	✓					✓	
	RF2	✓					✓	
	RF3							
	RF4		✓				✓	✓
	RF5					✓		
	RF6							
	RF7							
	RF8			✓				
	RF9				✓			
	RF10							

Correlación de Requerimientos vs casos de prueba de sistema

		Pruebas de sistema									
		CPS50	CPS51	CPS52	CPS53	CPS54	CPS55	CPS56	CPS57	CPS58	CPS59
Requerimientos	RF1	✓	✓								
	RF2			✓							
	RF3				✓	✓	✓				
	RN4						✓	✓			
	RF5								✓		
	RF6									✓	
	RF7										✓
	RN8										
	RF9										
	RF10	✓		✓	✓		✓	✓	✓	✓	✓

Correlación Requerimientos vs Casos de uso

A continuación se proporciona la matriz de trazabilidad de la correlación requerimientos vs casos de uso.

		Casos de uso			
		CU1	CU2	CU3	CU4
Requerimientos	RF1.1	✓			
	RF1.2	✓			
	RF1.3	✓			
	RF1.4	✓			
	RF2.1			✓	
	RF2.2			✓	
	RF3.1		✓		
	RF3.2		✓		
	RF3.3		✓		
	RF4.1		✓		
	RF4.2		✓		
	RF4.3		✓		
	RF4.4		✓		
	RF4.5		✓		
	RF5.1				✓
	RF5.2				✓
	RF5.3				✓
	RF6.1		✓		
	RF6.2		✓		
	RF7.1		✓		
	RF7.2		✓		
	RF8.1	✓	✓		✓
	RF9.1	✓	✓		✓
	RF9.2	✓	✓		✓
	RF9.3	✓	✓		✓
	RFI1.1	✓			
	RFI1.2				✓
	RFI1.3	✓	✓	✓	

3.4. ANÁLISIS DE USO DE BYTES EN MATRICES Y VECTORES

Una variable entera acepta valores positivos y negativos dentro de un rango determinado, que depende de la plataforma y del compilador, por ejemplo en MS-DOS suele estar entre -32768 y 32767 ya que se utilizan registros de 16 bits; mientras que en Linux son enteros de 32 bits. En base a esto y la teoría del anexo 8.5, podemos concluir que la relación del tamaño en bytes ocupado por la creación de matrices dinámicas, en el kernel de Linux, a medida que aumentan sus filas y columnas queda dado por la siguiente ecuación:

$$\boxed{\text{Tamaño} = (\text{filas} \times 4 \text{ bytes}) + (\text{filas} \times \text{columnas} \times 4 \text{ bytes})}$$

Esta ecuación hace referencia al arreglo de punteros que representa cada fila y que es accedido mediante un doble puntero. El espacio necesario para este arreglo es el número de filas por la cantidad de bytes que ocupan los punteros de enteros, es decir, filas x 4 bytes, considerando que cada puntero entero es de 4 bytes. Por último cada puntero, del arreglo de punteros, apunta a un arreglo de enteros para acceder a cada columna de una fila de la matriz dinámica, de esta manera el espacio necesario de cada arreglo de enteros es el número de columnas por la cantidad de bytes de un entero (4 bytes) y como existe un arreglo de enteros por fila se multiplica por el número de filas, es decir, filas x columnas x 4 bytes.

Por otro lado en los vectores no es necesario un arreglo de punteros, ya que un vector se puede entender como una matriz de una sola fila con columnas igual al número de elementos. Por estas razones de los vectores el tamaño del mismo solo es afectado por su número de elementos, lo que equivale a un arreglo de enteros de una sola fila. Teniendo en cuenta estos factores la ecuación para determinar el tamaño que ocupa un vector se resume como sigue:

$$\boxed{\text{Tamaño} = \text{elementos} \times 4 \text{ bytes}}$$

La ecuación hace referencia a un único arreglo de enteros con un número de elementos por los que se compone el vector. La memoria utilizada será el número de elementos del vector multiplicado por el tamaño de los enteros en Linux que es de 4 bytes ya que dichos enteros utilizan registros de 32 bits.

4. DISEÑO E IMPLEMENTACIÓN

Como se describió anteriormente en la sección 1.3 del presente proyecto, la metodología de trabajo utilizada es la combinación de Kanban con el modelo de desarrollo iterativo incremental. De acuerdo con la sección 8.1.5 en este capítulo se planifican las diferentes iteraciones que componen el sistema final que se obtiene como resultado, describiendo todas sus características luego del proceso de diseño e implementación logrado con la ayuda de la metodología de trabajo utilizada.

4.1. ITERACIONES

4.1.1. ITERACIÓN 0

Partiendo de la base de un simple driver hello_world para el kernel de Linux, el driver planteado como objetivo general se divide en un conjunto de iteraciones que permiten el incremento iterativo hacia el sistema final. Las iteraciones son:

Iteración	Descripción
Iteración 1	Desarrollo de driver con capacidad de crear, mostrar y eliminar la estructura de una RDPG en el kernel (redes de Petri con arco inhibidor, lector y reset) mediante el uso de comandos.
Iteración 2	Desarrollo de driver con capacidad de configurar los diferentes parámetros con respecto a la gestión de las RDPG en el kernel, capacidad de gestionar la dinámica de la RDPG e inclusión de la capacidad de gestionar guardas sobre las transiciones de la RDPG. Se adicionan los comandos asociados a las nuevas funcionalidades.
Iteración 3	Desarrollo de driver con capacidad de reportar información sobre características de la RDPG y sus componentes. Se adicionan los comandos asociados a las nuevas funcionalidades.
Iteración 4	Desarrollo de driver con capacidad de proteger la RDPG cargada en el kernel del multiprocesamiento (característica SMPs). Se actualizan las funciones que realiza cada comando del driver para dar soporte a la característica SMPs.
Iteración 5	Desarrollo de librería C y C++ para gestionar driver desde el espacio usuario. Desarrollo de aplicación de usuario con las mismas características del driver MatrixmodG, pero gestionando las RDPG completamente en el espacio usuario.

Tabla 4.1 Iteraciones del sistema desarrollado.

4.1.2. ITERACIÓN 1

Durante la fase de desarrollo de esta iteración el objetivo fue conseguir una versión del driver que tenga la capacidad de crear y eliminar una RDPG (con arcos inhibidores, lectores y reset) mediante el uso de comandos enviados desde el espacio usuario.

Gran parte del trabajo de esta iteración se enfocó en la definición de los objetos a utilizar dentro del dominio del problema, la implementación y las diferentes pruebas del funcionamiento, rendimiento, etcétera de acuerdo con el plan de pruebas del capítulo 5.

Los objetos implementados fueron las matrices, vectores y las RDPG que hacen uso de los dos primeros. Toda la implementación de esta iteración se aplicó cumpliendo con los atributos del sistema como también de algunos requerimientos no funcionales tal como se describe en la sección 3.1.

Se diseñó la composición de un objeto RDPG para obtener su estructura. Para esto se definieron sus componentes bases que son la matriz de incidencia I (mII), la matriz de incidencia H de brazos inhibidores (mIH), la matriz de incidencia R de brazos lectores (mIR), la matriz de incidencia Re de transiciones reset (mIRe) y el vector de marcado inicial (vMI).

En esta primera versión del driver, también se armó toda la estructura de las funciones de escritura y lectura (funciones write() y read() respectivamente). La función de escritura del driver es la que permite la interpretación del primer conjunto de comandos para la creación, configuración de lectura y eliminación de la estructura de una RDPG cuando el archivo de dispositivo del driver es escrito desde el espacio usuario. La función de lectura del driver es la que permite al espacio usuario obtener la información del dominio del problema gestionada por el kernel, inicialmente se necesita conocer el estado de cada componente de una RDPG de acuerdo al componente indicado en un comando de escritura previamente enviado.

De acuerdo con las secciones 3.1 y el capítulo 5, la siguiente tabla muestra los requerimientos cubiertos en la iteración como así también el conjunto de pruebas realizadas para dicho requerimientos.

Requerimientos cubiertos	Pruebas unitarias e integrales	Pruebas de sistema
RF1.1	BPU2	CPS50
RF1.2	BPU2	CPS50
RF1.3	BPU2	CPS50
RF1.4	BPU2	CPS51
RF2.1	BPU2	CPS52
RF2.2	BPU2	CPS52
RF3.1	-	CPS53
RF3.2	-	CPS54
RF3.3	-	CPS55
RF8.1	BPU4	-
RF9.1	BPU5	-
RF9.2	BPU5	-
RF9.3	BPU5	-

Tabla 4.2 Requerimientos cubiertos en iteración 1.

4.1.3. ITERACIÓN 2

Las implementaciones referidas a la segunda iteración se basaron en la incorporación de las funcionalidades para gestionar la dinámica de la RDPG cargada en el kernel. El objetivo de esta iteración fue obtener una versión del driver que permita realizar el disparo de cualquiera de las transiciones de la RDPG cargada en el kernel y la posibilidad de gestionar las guardas sobre cualquiera de las transiciones de la RDPG.

Se implementó la funcionalidad de realizar disparos sobre las transiciones de la RDPG haciendo uso del modelo matemático de las RDPG con su ecuación de estado extendida. Se adicionó el vector de guardas (vG) y se

incorporaron todos los comandos necesarios en la función de escritura del driver para la gestión de las nuevas funcionalidades.

La capacidad del arco reset tiene efecto si las condiciones y los brazos que habilitan a la transición del arco reset, están dadas. En otro caso no se realiza y se pierde la señal reset. Por lo cual, el brazo reset debe estar explícitamente indicado sobre la matriz de incidencia I.

Debe tenerse en cuenta que solicitar disparar una transición asociada a un arco reset es un evento no perenne, es decir, en el momento que se intentó realizar el reset y la transición no está sensibilizada, este evento se pierde.

Otro objetivo de la iteración fue el de obtener una versión del driver MatrixmodG que pueda ser configurado a través del envío de un conjunto de comandos de configuración de parámetros del driver desde el espacio usuario.

Se trabajó en la creación del nuevo conjunto de comandos de configuración de parámetros y las respectivas funcionalidades realizadas por el driver al recibir estos comandos. Se actualizó la función de escritura del driver para adaptarse al nuevo conjunto de comandos. La configuración de estos comandos impacta en la forma que trabaja el driver MatrixmodG con respecto a la gestión de las RDPG y la forma en que se muestran los datos al usuario al momento que se realiza una lectura. Por esta razón se realizaron las modificaciones correspondientes haciendo que las lecturas respondan a los parámetros configurados y también la forma de asignar memoria responda a la configuración previamente asignada. Todos los parámetros tienen un valor por defecto si no se configuran previamente.

Los requerimientos cubiertos por esta iteración y sus casos de pruebas asociados se muestran en la siguiente tabla:

Requerimientos cubiertos	Pruebas unitarias e integrales	Pruebas de sistema
RF4.1	BPU3	CPS56
RF4.2	BPU3	CPS56
RF4.3	BPU3	CPS55
RF5.1	BPU6	CPS57
RF5.2	BPU6	CPS57
RF5.3	BPU6	CPS57

Tabla 4.3 Requerimientos cubiertos en iteración 2.

4.1.4. ITERACIÓN 3

La iteración 3 tiene como objetivo obtener una versión del driver MatrixmodG que tenga la capacidad de informar acerca de las características de los objetos gestionados en el kernel de Linux.

Como los objetos gestionados por el driver son la RDPG y sus componentes, que son vectores o matrices, se implementó la funcionalidad de reportar información referida a la RDPG cargada en el kernel y sus componentes. La información que proporciona el driver respecto a la RDPG se vincula con la cantidad de memoria reservada, memoria real utilizada, cantidad de plazas y transiciones, método de asignación de memoria y nombre de la RDPG. Para cada componente de la RDPG también se permite obtener la información de cada uno en particular.

Para cumplir con las funcionalidades se extendió el conjunto de comandos creando comandos referidos a la extracción de cada uno de los atributos necesarios de la RDPG como también de sus componentes.

Debido a que se agregaron nuevos comandos y nuevas funcionalidades en el driver, todos los cambios impactaron en las funciones de escritura y lectura del mismo, adaptando cada una de las funciones a las nuevas funcionalidades.

La siguiente tabla muestra los requerimientos cubiertos por esta iteración y los casos de prueba realizados para dar garantía de un correcto funcionamiento.

Requerimientos cubiertos	Pruebas unitarias e integrales	Pruebas de sistema
RF6.1	-	CPS58
RF6.2	-	CPS58
RF7.1	-	CPS59
RF7.2	-	CPS59

Tabla 4.4 Requerimientos cubiertos en iteración 3.

4.1.5. ITERACIÓN 4

En la iteración 4 se tuvo como objetivo obtener una versión mejorada del driver MatrixmodG que gestione las RDPG en kernel con soporte al multiprocesamiento simétrico seguro, esto significa que la RDPG del driver pueda ser ejecutada por múltiples procesos sin que existan problemas de concurrencia en la gestión del recurso compartido.

Para lograr el objetivo de la iteración fue necesario determinar todas las funciones que realiza el driver MatrixmodG en las que es necesario brindar protección a la RDPG, como la RDPG se lee y escribe puede ser inseguro si se lee y escribe al mismo tiempo, es por lo cual todas sus funciones deben proveer exclusión mutua en cada operación de escritura y lectura. El mecanismo utilizado en el driver matrixmodG para proveer exclusión mutua en la gestión de la RDPG cargada en el kernel fue utilizar spinlock, usando su variante de spinlock lector-escritor, una variante de spinlock del kernel más eficiente para los problemas de lectura y escritura tal como sucede en la gestión de las RDPG, donde se puede leer la RDPG por múltiples lectores concurrentes siempre que no exista algún escritor que interrumpa, y solo un escritor puede operar en la RDPG a la vez, sin que otro escritor o lector interrumpa.

Se implementaron un conjunto de métodos de la RDPG con soporte SMPs en una capa de programación superior para lograr gestionar todos los métodos de lectura y escritura en la RDPG. Estos métodos hacen uso del spinlock antes de operar con la RDPG, de esta forma se provee la exclusión del recurso y además se provee eficiencia al separar las operaciones de escritura por sobre las de lectura para permitir en la RDPG múltiples procesos de lectura concurrentes como se mencionó anteriormente.

En esta iteración se actualizaron todos los comandos que leen o escriben la RDPG actualizando el uso de los métodos con soporte SMPs.

Los requerimientos cubiertos por esta iteración y las pruebas realizadas se detallan en la siguiente tabla.

Requerimientos cubiertos	Pruebas unitarias e integrales	Pruebas de sistema
RNF2.1	BPI1, BPI2, BPI3	CPS101
RNF2.2	BPI1, BPI2, BPI3	CPS101

Tabla 4.5 Requerimientos cubiertos en iteración 4.

4.1.6. ITERACIÓN 5

En la iteración 5 se tuvo como objetivo generar una librería de espacio usuario para los lenguajes C y C++, que permita gestionar el driver MatrixmodG de una manera fácil y simple.

En esta iteración se trabajó en el diseño e implementación del objeto que representa al driver desde la librería de espacio usuario, el cual se denominó DriverRDPG_o (en C) y RDPG_Driver (en C++). Este objeto al instanciarlo en el espacio usuario, permite realizar todas las operaciones del driver MatrixmodG e incluso facilita un conjunto de funcionalidades gracias a la automatización de operaciones sobre la que se trabajó para lograr simplificar su uso.

La librería permite cargar cualquier RDPG desde el espacio usuario mediante el uso de archivos de texto. Se interacciona con el driver a través de llamadas al sistema de escritura y lectura a través del file device asociado.

En esta iteración también se diseñó para la capa de usuario un monitor que gestiona la RDPG con protección al multiprocesamiento. De esta forma se pudo crear un conjunto de hilos que operan una RDPG concurrentemente de forma eficiente. La clase monitor se diseñó en el lenguaje C++ y la misma hace uso de las variables mutex y condition_variable para proveer la exclusión mutua entre procesos.

Por último se realizó el desarrollo de una aplicación de C++ con las mismas características que el driver MatrixmodG con el fin de poder realizar una comparación de rendimiento entre el driver MatrixmodG y la aplicación de C++.

Para obtener las mediciones de tiempo de las operaciones del driver y la aplicación C++ desde el espacio usuario, se utilizó la API OpenMP y la librería time.h, con el fin de tener dos referencias en la medición del tiempo.

La siguiente tabla muestra los requerimientos cubiertos y las pruebas asociadas.

Requerimientos cubiertos	Pruebas unitarias e integrales	Pruebas de sistema
RFI1.1	-	CPS50
RFI1.2	-	CPS57
RFI1.3	-	CPS59, CPS58, CPS56, CPS55, CPS53, CPS52

Tabla 4.6 Requerimientos cubiertos en iteración 5.

4.2. COMPOSICIÓN DEL SISTEMA FINAL

Cumpliendo con cada una de las iteraciones se obtuvo el sistema de software final de acuerdo con los requerimientos de sistema, sus atributos y los objetivos del proyecto. Cada iteración fue aportando su parte hasta obtener el sistema de software final. A continuación se detalla los aspectos más importantes por los que está compuesto el driver MatrixmodG.

4.2.1. COMANDOS DE DRIVER MATRIXMODG

Como se describe en los casos de uso, el driver matrixmodG tiene que gestionar una RDPG desde el kernel de Linux. Para ingresar los parámetros de creación, configuración y gestión de una RDPG desde el espacio usuario al kernel y viceversa se necesita de un conjunto de comandos que sean interpretados por el driver para llevar a cabo todas las operaciones que este provee.

Cada escritura que se realiza sobre el archivo de dispositivo del driver matrixmodG es considerada un comando. Si el comando es interpretado por el driver se realiza una operación asociada, en caso de no interpretar el comando el driver notifica el error por la salida de diagnóstico del kernel y no realiza ninguna otra operación. Los comandos que interpreta el driver se agrupan en cinco tipos diferentes relacionados con el tipo de operación que realiza cada uno de estos. Los tipos de comando que reconoce el driver son:

- Comandos de configuración de parámetros asociados a la RDPG.
- Comandos de configuración de la RDPG y sus componentes.
- Comandos de reportes de información de la RDPG y sus componentes.
- Comandos de control del comportamiento de la RDPG.
- Comandos de lectura de componentes de la RDPG.

Comandos de configuración de parámetros asociados a la RDPG

Son comandos que se utilizan para configurar parámetros que utiliza el driver matrixmodG para realizar operaciones con las RDPG en algunos casos antes de que estas sean creadas.

Comando	Descripción
RDPG config alloc_mode m Ejemplo de uso: RDPG config alloc_mode 3	Este comando indica al driver matrixmodG que deberá configurar las asignaciones de memoria para cada uno de los componentes de la RDPG con el modo m. m: Indica el modo de asignación de memoria a utilizar. Puede ser 1, 2, 3, 4 o 5 ver sección de “Creación de objetos” para más detalle.
RDPG config vdim n Ejemplo de uso: RDPG config vdim 15	Este comando indica al driver MatrixmodG configurar el parámetro de dimensión de visualización de componentes (vdim) de la RDPG con el entero n. n: Número entero que indica el valor al que se desea configurar vdim en la RDPG. Sus valores van de un mínimo a un máximo restringidos por el propio driver.
RDPG config posVP n Ejemplo de uso: RDPG config posVP 100	Este comando indica al driver MatrixmodG que debe configurar la posición de vista de plazas (posVP) con el entero n. n: Número entero que indica el valor al que se desea configurar posVP. Sus valores van de 0 a la cantidad de plazas que tenga la RDPG_o cargada en el kernel.
RDPG config posVT n Ejemplo de uso: RDPG config posVT 100	Este comando indica al driver matrixmodG que debe configurar la posición de vista de transiciones (posVT) con el entero n. n: Número entero que indica el valor al que se desea configurar posVT. Sus valores van de 0 a la cantidad de transiciones que tenga la RDPG_o cargada en el kernel.

Tabla 4.7 Comandos de configuración de parámetros.

Comandos de configuración de la RDPG y sus componentes

Los comandos de configuración de la RDPG y sus componentes se utilizan para crear la red y sus componentes, inicializar la red, modificar los componentes que lo requieran, configurar la red o eliminarla del kernel.

Comando	Descripción
create RDPG n_m Ejemplo de uso: create RDPG 10_6	Este comando indica al driver crear una nueva RDPG con un número de n plazas y m transiciones. n: número de plazas de la RDPG. m: número de transiciones de la RDPG.
RDPG add comp posn_posm_v Ejemplo de uso: RDPG add mII 1_0_1 RDPG add mIH 1_0_1 RDPG add mIR 1_0_1 RDPG add mIRe 1_0_1	Este comando indica al driver agregar un nuevo valor en el componente matriz comp de la RDPG. comp: componente de la RDPG. Puede ser: <ul style="list-style-type: none"> • mII: Matriz de incidencia I. • mIH: Matriz de incidencia H. • mIR: Matriz de incidencia R. • mIRe: Matriz de incidencia Reset. posn: número de plaza. Se utiliza como posición de filas en las matrices de incidencia y posición de columnas en los vectores de marcado. posm: número de transición. Se utiliza como posición de columna en las matrices de incidencia y en los vectores de transiciones. v: valor que se agrega en el componente de la RDPG.
RDPG add comp pos_v Ejemplo de uso: RDPG add vMI 4_1	Este comando indica al driver agregar un nuevo valor en el componente vector comp de la RDPG. comp: componente de la RDPG. Puede ser: <ul style="list-style-type: none"> • vMI: Vector de marcado inicial. pos: número de plazas. Se utiliza como posición de elementos en el vector de marcado inicial. v: valor que se agrega en el componente de la RDPG.
RDPG set vG pos_v Ejemplo de uso: RDPG set vG 4_1	Este comando indica al driver actualizar un nuevo valor en el componente vector de guardas vG de la RDPG. pos: número de transición afectada. Se utiliza como posición de elementos en el vector de guardas. v: valor que se agrega en el componente vG de la RDPG.
RDPG inc vHQCv pos Ejemplo de uso: RDPG inc vHQCv 1	Este comando indica al driver incrementar a uno el valor del componente vector de hilos en cola de variable de condición vHQCv de la RDPG. pos: número de la cola de variable de condición de la transición afectada. Es la posición del elemento a incrementar en el vector vHQCv.
RDPG dec vHQCv pos Ejemplo de uso: RDPG dec vHQCv 4	Este comando indica al driver decrementar a uno el valor del componente vector de hilos en cola de variable de condición vHQCv de la RDPG. pos: número de cola de variable de condición de la transición afectada. Es la posición del elemento a decrementar en el vector vHQCv.
RDPG confirm Ejemplo de uso:	Este comando confirma los componentes cargados en la RDPG del driver matrixmodG para que se actualicen el resto de componentes que se calculan a partir de los componentes previamente cargados.

RDPG confirm	Luego de este comando la RDPG del kernel esta lista en su estado inicial.
RDPG delete	Este comando indica al driver borrar la RDPG cargada en el kernel.

Tabla 4.8 Comandos de configuración de componentes de una RDPG.

Comandos de reportes de información de la RDPG y sus componentes

Para conocer la información de los objetos cargados en el kernel se utilizan los siguientes comandos de reporte de información.

Comando	Descripción
RDPGinfo name Ejemplo de uso: RDPGinfo name	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar el nombre de la RDPG.
RDPGinfo places Ejemplo de uso: RDPGinfo places	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar información de las plazas de la RDPG.
RDPGinfo transitions Ejemplo de uso: RDPGinfo transitions	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar información de las transiciones de la RDPG.
RDPGinfo shots Ejemplo de uso: RDPGinfo shots	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar información de los disparos disponibles en la RDPG.
RDPGinfo shot_result Ejemplo de uso: RDPGinfo shot_result	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar información del resultado del último disparo de transición efectuado en la RDPG.
RDPGinfo memory Ejemplo de uso: RDPGinfo memory	Este comando indica al driver matrixmodG cambiar el modo de lectura para mostrar información asociada a la memoria de la RDPG.
RDPGinfo comp Ejemplo de uso: RDPGinfo vMA	Este comando indica al driver matrixmodG que debe cambiar el modo lectura del driver a. comp: componente de la RDPG. Puede ser: <ul style="list-style-type: none"> • mII: Matriz de incidencia I, • mIH: Matriz de incidencia H, • mIR: Matriz de incidencia R, • mIRe: Matriz de incidencia Re, • mD: Matriz de disparos D, • vMI: Vector de marcado inicial. • vMA: Vector de marcado actual. • vMN: Vector de marcado nuevo.

	<ul style="list-style-type: none"> • vE: Vector de transiciones sensibilizadas. • vQ: Vector de función Q. • vW: Vector de función W. • vB: Vector de transiciones desensibilizadas B. • vL: Vector de transiciones desensibilizadas L. • vG: Vector de transiciones desensibilizadas G. • vA: Vector de transiciones reset A. • vUDT: Vector de resultado de último disparo de transición. • vEx: Vector de transiciones sensibilizadas extendido. • vHQCV: Vector de hilos en cola de variable de condición. • vHD: Vector de hilos a despertar.
RDPG get tokens place Ejemplo de uso: RDPG get tokens 2	Este comando indica al driver matrixmodG que retorne el valor de tokens de la plaza indicada de la RDPG. place: número de la plaza sobre la que se desea conocer el número de tokens.
RDPG get vHD element Ejemplo de uso: RDPG get vHD 4	Este comando indica al driver matrixmodG que retorne el valor del vector vHD para el elemento indicado. element: número del elemento del vector vHD que se desea conocer. El elemento indicado siempre está dentro del dominio de transiciones de la RDPG.

Tabla 4.9 Comandos de reporte de información de la RDPG.

Comandos de control del comportamiento de la RDPG

Con los comandos de control del comportamiento de la RDPG se puede ejecutar disparos en transiciones de la RDPG para que cambie su estado actual.

Comando	Descripción
RDPG shoot nt Ejemplo de uso: RDPG shoot 2	Este comando indica al driver matrixmodG disparar la transición nt. nt: número de transición a disparar en la RDPG.

Tabla 4.10 Comandos de control del comportamiento de la RDPG.

Comandos de lectura de componentes de la RDPG

Los comandos de lectura de componentes son utilizados por el driver MatrixmodG para saber que componente de la RDPG se desea leer al momento de realizar una lectura sobre el driver.

Comando	Descripción
RDPG cat comp Ejemplo de uso: RDPG cat mII	Este comando indica al driver matrixmodG que deberá configurar las asignaciones de memoria para cada uno de los componentes de la RDPG con el modo m. comp: componente de la RDPG. Igual que comando RDPGinfo comp.

Tabla 4.11 Comandos de lectura de componentes de la RDPG.

4.2.2. OBJETOS DE DRIVER MATRIXMODG

La inclusión de POO ayuda a que el diseño del sistema utilice un conjunto de objetos para simplificarlo y permitir la resolución del problema de una manera más simple y eficiente a la vez. A continuación se detallan los objetos que se diseñaron e implementaron en el driver MatrixmodG, estos serán los objetos instanciados en el kernel de Linux al momento de su ejecución.

Objetos matrix_o y vector_o

De acuerdo con la sección 2.8.3 del capítulo 2, se logra encapsular los datos asociados a los objetos con el uso de estructuras y resumiendo los nombres con estructuras opacas. De esta forma el objeto matrix_o presenta la estructura de la figura 4.1 donde también se detallan los métodos que el objeto dispone.

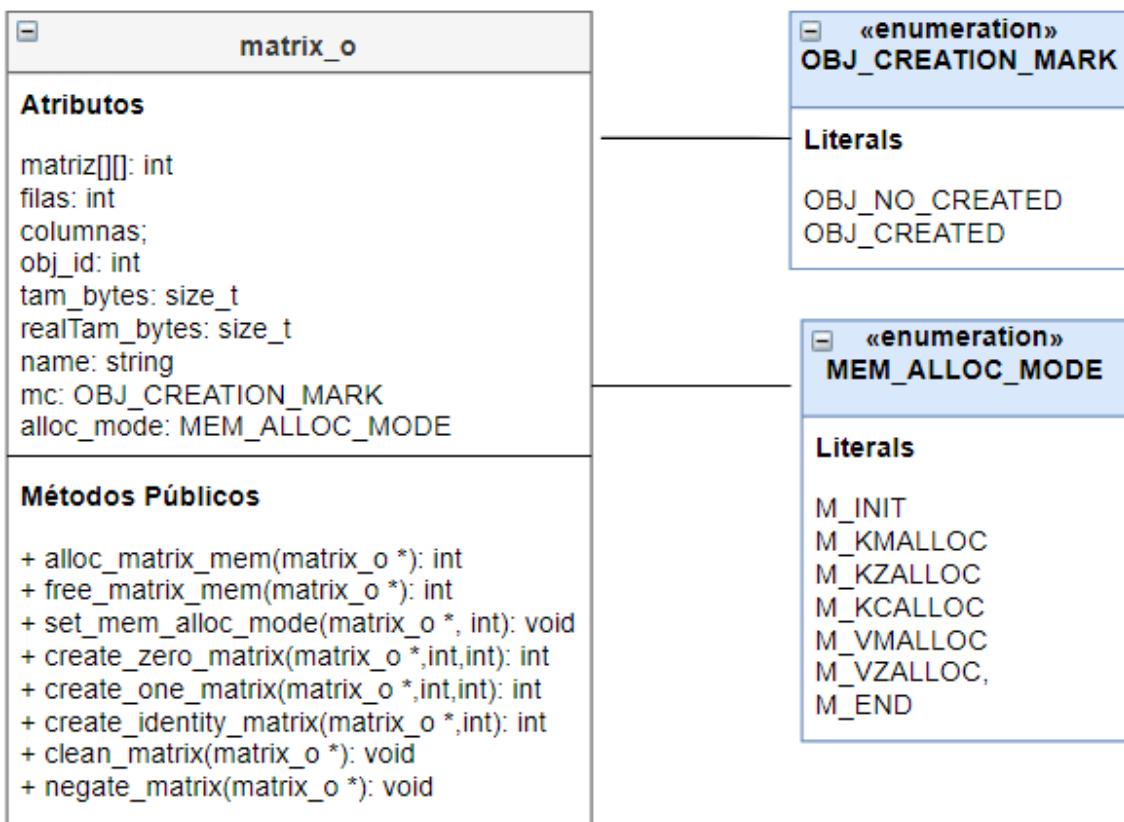


Figura 4.1 Diseño de objeto matrix_o.

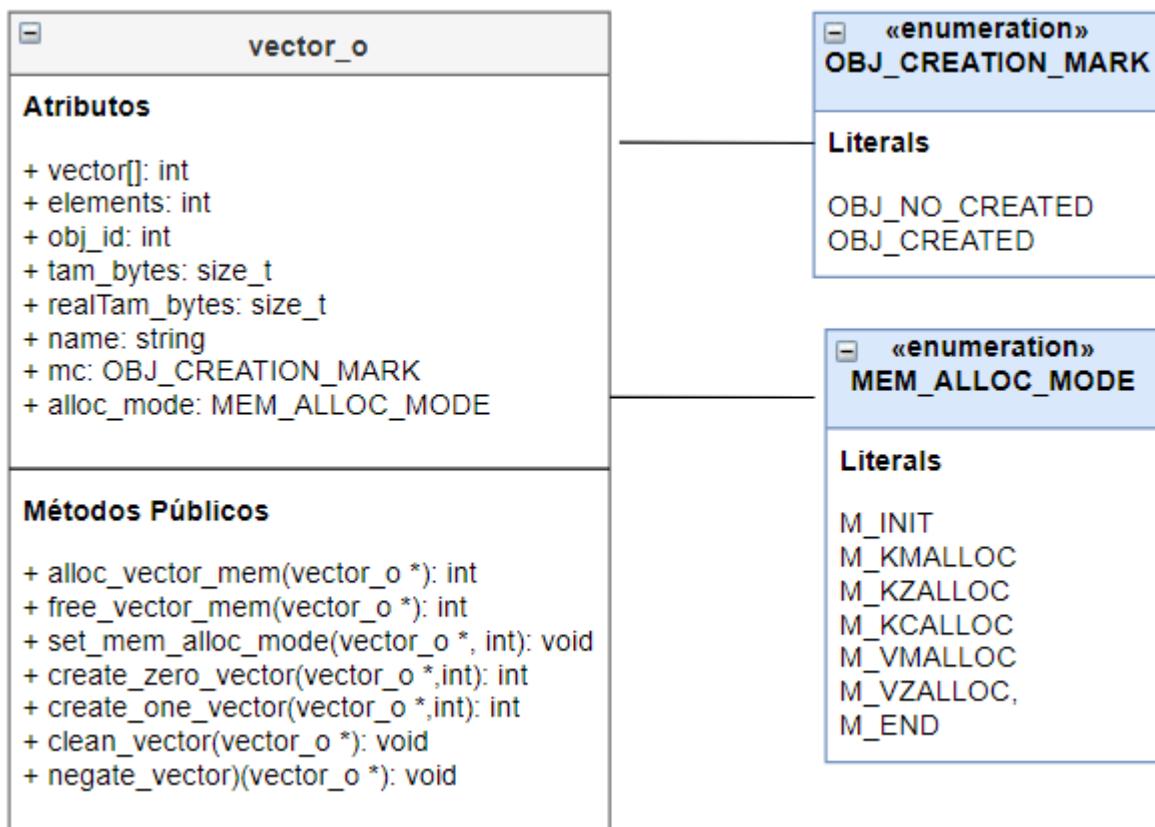


Figura 4.2 Diseño de objeto `vector_o`.

Para el objeto de tipo `vector_o` se realizó lo mismo tal como se muestra en la figura 4.2.

El objeto `matrix_o` de la figura 4.1 cuenta con los siguientes atributos:

- ****matriz:** es un doble puntero de tipo entero, el cual se utiliza para mantener la relación de una matriz con respecto a sus filas y columnas. Se utiliza para proporcionar asignación dinámica adaptándose a cada caso en particular.
- **filas:** atributo que almacena el número de filas de la matriz.
- **columnas:** Filas: atributo que almacena el número de columnas de la matriz.
- **obj_id:** atributo que almacena el número identificador único de la matriz.
- **tam_bytes:** atributo que almacena el tamaño en bytes reservado por la matriz.
- **realTam_bytes:** atributo que almacena el tamaño real en bytes necesario por la matriz.
- **name:** atributo que almacena el nombre de la matriz.
- **mc:** atributo que indica si el objeto fue creado dinámicamente en el kernel o no.
- **alloc_mode:** atributo que indica el modo de reserva a utilizar en el kernel.

De la misma manera el objeto `vector_o` cuenta con los siguientes atributos:

- ***vector:** es un puntero de tipo entero, el cual se utiliza para mantener la relación del vector con respecto a sus elementos. Se utiliza para proporcionar asignación dinámica adaptándose a cada caso en particular.
- **elements:** atributo que almacena el número de elementos del vector.
- **obj_id:** atributo que almacena el número identificador único del vector.
- **tam_bytes:** atributo que almacena el tamaño en bytes reservado por el vector.
- **realTam_bytes:** atributo que almacena el tamaño real en bytes necesario por el vector.
- **name:** atributo que almacena el nombre del vector.
- **mc:** atributo que indica si el objeto fue creado dinámicamente en el kernel o no.
- **alloc_mode:** atributo que indica el modo de reserva a utilizar en el kernel.

Las enumeraciones utilizadas por los objetos `matrix_o` y `vector_o` son `OBJ_CREATION_MARK` y `MEM_ALLOC_MODE`. En el primer caso se utiliza la enumeración para indicar si un objeto `matrix_o` o

vector_o esta creado en el kernel indicando que reservo memoria para su creación. En el segundo caso se utiliza la enumeración para indicar el tipo de algoritmo a utilizar para la reserva de memoria asociada a la creación de un objeto matrix_o o vector_o, se detalla esta característica en la sección 6.1.1 del capítulo 6.

El detalle de cada uno de los métodos se realiza en la sección 4.5.3 donde se describe la responsabilidad que tiene cada método en particular sobre los objetos matrix_o y vector_o implementados en la librería MV_object.h.

Para conocer los detalles de las relaciones entre cada uno de los objetos se puede ver la sección 4.3.2 sobre la vista lógica del sistema.

Objeto RDPG_o

La estructura diseñada del objeto que representa las RDPG y sus métodos se presenta en varias figuras, ya que por su gran tamaño se decidió dividirlo de acuerdo a conjuntos de atributos particulares de la siguiente forma.

- La figura 4.3 muestra los atributos relacionados con la identificación de la RDPG y sus atributos de estado.
- La figura 4.4 muestra los atributos relacionados con los componentes de la RDPG del tipo matrix_o y vector_o.
- Por último la figura 4.5 muestra todos los métodos que dispone un objeto RDPG.

Los detalles de los atributos de la figura 4.3 son los siguientes:

- **lock_RDPG:** Bloqueo de RDPG, variable spinlock proveedora de exclusión mutua a la RDPG dentro del kernel de Linux.
- **name:** nombre de la RDPG.
- **obj_id:** identificador de la RDPG.
- **select_comp:** número del componente seleccionado para una lectura de componente en la RDPG.
- **shot_result:** resultado del último disparo realizado.
- **posVP:** número de plaza inicial en la cual posicionarse para la visualización del estado de los componentes de la RDPG.
- **posVP:** número de la transición inicial en la cual posicionarse para la visualización del estado de los componentes de la RDPG.
- **vdim:** cantidad de plazas y transiciones a visualizar cuando se muestra el estado de los componentes.
- **mc:** marca de creación dinámica del objeto sobre el kernel de Linux.
- **read_mode:** modo de lectura configurado en el objeto RDPG_o cuando este se lee desde su file device asociado.
- **sread_mode:** sub-modo de lectura configurado en el objeto RDPG_o cuando este se lee desde su file device asociado.
- **s_plazas:** string con el número de plazas.
- **s_transiciones:** string con el número de transiciones.
- **s_size:** string con el número de tamo en bytes de la RDPG.
- **s_realSize:** string con el número del tamaño real en bytes de la RDPG.
- **s_allocMode:** string con el modo de reserva de memoria utilizado por la RDPG.
- **s_header_transiciones:** string con la cabecera de transiciones.
- **s_header_plazas:** string con la cabecera de plazas.
- **s_header_disparos:** string con la cabecera de disparos.

RDPG_o
Atributos identificadores y de estado
+ rlock_t lock_RDPG;
+ name: string
+ obj_id: int
+ select_comp: int
+ shot_result: int
+ posVP: size_t
+ posVT: size_t
+ vdim: size_t
+ mc: OBJ_CREATION_MARK
+ read_mode: ID_READ_MODE
+ read_smode: ID_READ_SUBMODE
+ s_plazas: string
+ s_transiciones: string
+ s_size: string
+ s_realSize: string
+ s_allocMode: string
+ s_header_transiciones: string
+ s_header_plazas: string
+ s_header_disparos: string

Figura 4.3 Atributos de identificación y estado de un objeto RDPG_o.

Los detalles de la figura 4.4 son los siguientes:

- **mII:** matriz de incidencia I de la RDPG. La matriz contiene todas las relaciones de arcos normales entre cada plaza y transición de la RDPG.
- **mIH:** matriz de incidencia H de brazos inhibidores de la RDPG. Contiene todas las relaciones de arcos inhibidores entre cada plaza y transición de la RDPG.
- **mIR:** matriz de incidencia R de brazos lectores de la RDPG. Contiene todas las relaciones de arcos lectores entre cada plaza y transición de la RDPG.
- **mIRE:** matriz de incidencia Re de brazos reset de la RDPG. Contiene todas las relaciones de arcos reset entre cada plaza y transición de la RDPG.
- **mD:** matriz de identidad de disparos posibles en la RDPG. Contiene todos los vectores de disparo de transición posible para la RDPG, una fila o columna completa representa un vector disparo de una transición.
- **vMI:** vector de marcado inicial de la RDPG. Indica el estado inicial de la RDPG, se entiende por estado inicial el número de tokens con el que inicia cada una de las plazas de la RDPG.
- **vMA:** vector de marcado actual de la RDPG. Indica el estado actual de la RDPG, se entiende por estado actual el número de tokens en tiempo real que tiene cada una de las plazas de la RDPG.
- **vMN:** vector de marcado nuevo de la RDPG. Indica el estado obtenido al realizar el disparo de una transición de la RDPG, si el disparo falla existirán valores negativos indicando la cantidad de tokens necesarios para sensibilizar la transición a disparar, si el disparo no falla el resultado coincide con el vector de marcado actual.
- **vE:** vector E de transiciones sensibilizadas de la RDPG. Indica con un uno cuales transiciones de la RDPG están sensibilizadas, caso contrario se indica con cero.
- **vQ:** vector Q de función cero. Cada elemento indica con cero si la marca de plaza es distinta de cero y uno si la marca de plaza es cero.
- **vW:** vector W de función uno. Cada elemento indica con uno si la marca de plaza es distinta de cero y cero si la marca de plaza es cero.
- **vL:** vector L de transiciones inhibidas por arco lector. Indica con un cero las transiciones inhibidas por arco lector.

- **vB:** vector B de transiciones inhibidas por arco inhibidor. Indica con un cero las transiciones inhibidas por arco inhibidor.
- **vG:** vector G de transiciones inhibidas por guardas. Cada elemento asociado a una transición indica con cero las transiciones inhibidas por guarda.
- **vA:** vector A de transiciones reset. Cada elemento asociado a una transición reset indica el número de tokens de la plaza que se quiere resetear a cero, el resto de elementos son uno.
- **vUDT:** vector UDT de resultado de último disparo de transición. Cada elemento se asocia a una transición donde se indica con uno si el último disparo de esa transición fue exitoso o cero si no fue exitoso.
- **vEx:** vector extendido de transiciones sensibilizadas. Cada elemento se asocia a una transición donde se indica con un uno si la transición está sensibilizada o está des-sensibilizada por alguno de los diferentes arcos.
- **vHQCV:** vector de hilos en cola de variables de condición. Cada elemento se asocia a una transición en donde se indica con un número mayor a cero la cantidad de hilos en cola de una variable de condición de la transición y cero si no hay hilos en cola.
- **vHD:** vector de hilos a despertar. Cada elemento se asocia a una transición en donde se indica con uno si se debe despertar un hilo para la transición asociado y se indica con cero si no existen hilos a despertar para esa transición.

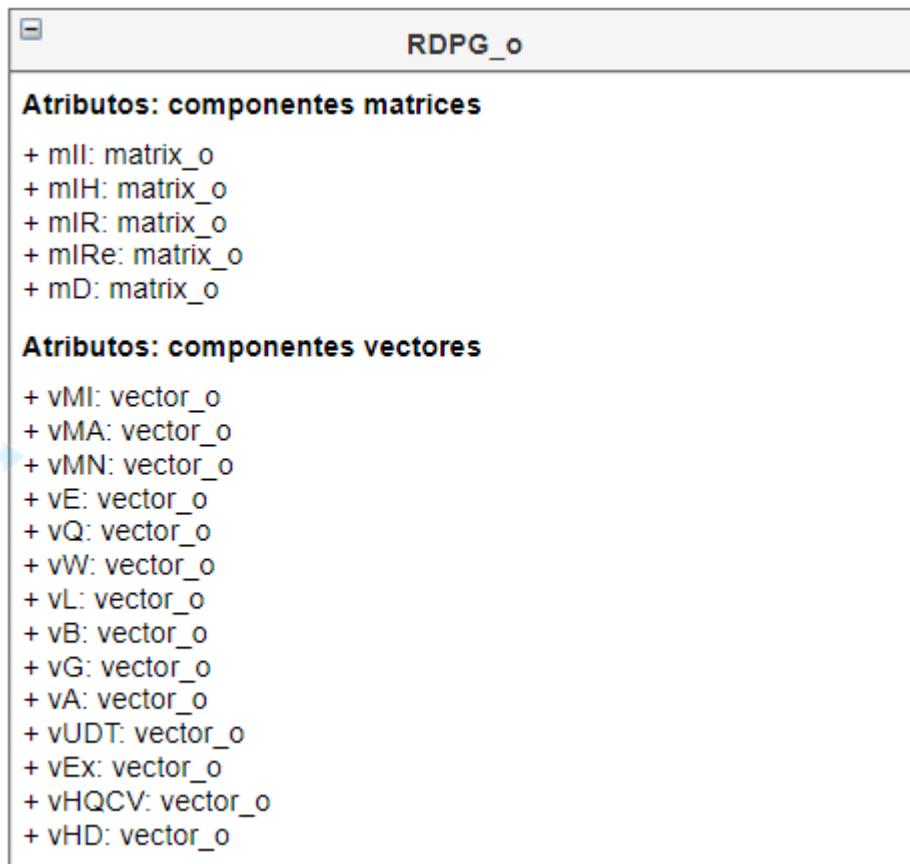


Figura 4.4 Atributos componentes de un objeto RDPG_o.

Por último los detalles de la figura 4.5 se realizan en la sección 4.5.2 donde se describe la responsabilidad de cada uno de los métodos que dispone un objeto RDPG_o definido en la librería RDPG_object.h diseñada para el kernel de Linux.

RDPG_o
Métodos Públicos
+ new_RDPG(name: string): *RDPG_o
+ create_rdp(RDPG_o *,char *): void
+ delete_rdp(RDPG_o *): void
+ add_value_in_mcomponent(matrix_o *,char *): int
+ add_value_in_vcomponent(vector_o *,char *): int
+ add_value_vG(RDPG_o *,char *): int
+ update_work_components(RDPG_o *): void
+ update_VG(RDPG_o *): void
+ load_vcomp_with_vcomp(RDPG_o *, int, int): void
+ shoot_rdp(RDPG_o *,int, SHOT_MODE): int
+ read_rdp_component(RDPG_o *,char *,size_t): int
+ read_rdp_info(RDPG_o *,char *,size_t): int
+ get_TokensPlace(RDPG_o *, char *): int
+ set_MemAllocMode(RDPG_o *, char *): void
+ set_select_comp(RDPG_o *, int): void
+ set_read_mode(RDPG_o *,ID_READ_MODE): void
+ set_read_smode(RDPG_o *, ID_READ_SUBMODE): void
+ set_posVP(RDPG_o *,char *): void
+ set_posVT(RDPG_o *,char *): void
+ set_vdim(RDPG_o *,char *): void
+ set_catComp)(RDPG_o *,int): void
+ inc_vHQCV(RDPG_o *,char *): void
+ dec_vHQCV)(RDPG_o *,char *): void
+ shoot_rdp_s(RDPG_o *,char *, SHOT_MODE): int
+ SMPs_shoot_rdp_s(RDPG_o *,char*, SHOT_MODE): int
+ SMPs_shoot_rdp(RDPG_o *,int, SHOT_MODE): int
+ SMPs_add_value_vG(RDPG_o *,char *): int
+ SMPs_update_work_components(RDPG_o *): void
+ SMPs_load_vcomp_with_vcomp(RDPG_o *, int, int): void
+ SMPs_get_TokensPlace(RDPG_o *, char *): int
+ SMPs_set_MemAllocMode(RDPG_o *, char *): void
+ SMPs_set_select_comp(RDPG_o *, int): void
+ SMPs_set_read_mode(RDPG_o *,ID_READ_MODE): void
+ SMPs_set_read_smode(RDPG_o *, ID_READ_SUBMODE): void
+ SMPs_set_posVP(RDPG_o *,char *): void
+ SMPs_set_posVT(RDPG_o *,char *): void
+ SMPs_set_vdim(RDPG_o *,char *): void
+ SMPs_set_catComp(RDPG_o *,int): void
+ SMPs_inc_vHQCV(RDPG_o *,char *): void
+ SMPs_dec_vHQCV(RDPG_o *,char *): void
+ SMPs_read_rdp_component(RDPG_o *,char *,size_t): int
+ SMPs_read_rdp_info(RDPG_o *,char *,size_t): int

Figura 4.5 Métodos de objeto RDPG_o.

4.2.3. CARACTERÍSTICA SMPs DE DRIVER MATRIXMODG

Como se vio en el diseño de la clase RDPG_o para los objetos RDPG que se gestionan en el kernel de Linux figura 4.3, uno de los atributos es un spinlock del tipo rwlock_t denominado lock_RDPG. Este atributo u objeto de la RDPG es un componente que se utiliza para proveer protección al recurso mediante la provisión de exclusión mutua en el kernel de Linux, esto hace al código del driver obtener la característica SMP-secure, lo que significa que el código es seguro al momento de ejecutarse sobre diferentes procesadores simultáneamente en sistemas operativos con soporte a la característica SMP del kernel de Linux. Como se describe en la sección 2.7.6 del capítulo 2 el objeto se representa como se muestra en la figura 4.6:

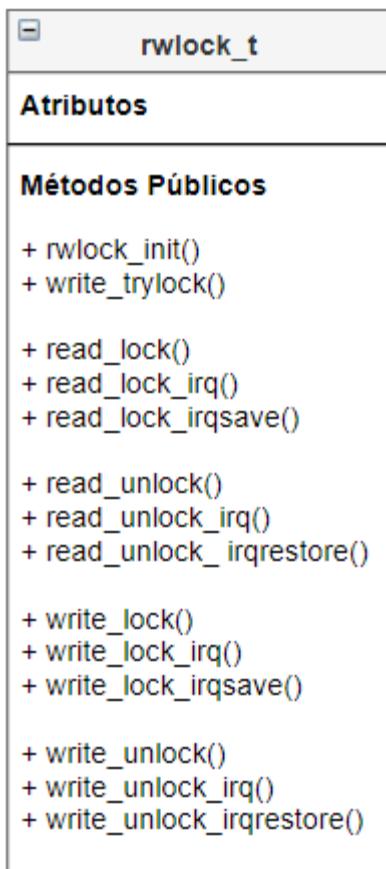


Figura 4.6 Objeto rwlock_t.

Este componente de la RDPG cargada en el kernel, se utiliza en todos los métodos de la RDPG cuyos nombres comienzan con “SMPs_método” (figura 4.5), lo que significa que todos estos métodos tienen protección del recurso dentro del kernel de Linux, en la sección 4.5.2 se puede ver el detalle de la responsabilidad que tiene cada uno de estos métodos en el dominio del problema relacionado al proyecto.

4.2.4. MACROS DE MENSAJES DE DEBUG

El driver MatrixmodG, se diseñó con un conjunto de macros que permiten habilitar o deshabilitar todos los mensajes de su funcionamiento por la salida de diagnóstico del kernel. Los mensajes son mensajes de información de las operaciones realizadas por el driver en el kernel y mensajes de errores en las operaciones. Estos últimos son los más importantes ya que son errores sobre el driver y pueden servir para permitir encontrar bugs sobre el código del mismo.

Las macros para controlar la habilitación o deshabilitación de mensajes sobre el driver MatrixmodG son tres y se muestran en la figura 4.7, allí se indica su denominación y en qué parte del código manipularlas para su habilitación o deshabilitación.

Este diseño es muy útil, ya que permite que durante el proceso de pruebas se tenga registro de cada uno de los movimientos que va realizando el driver para cada una de las operaciones realizadas, y así verificar si todo ejecuta correctamente libre de bugs o viceversa, permitiendo encontrar errores en las zonas de código donde se presenten.

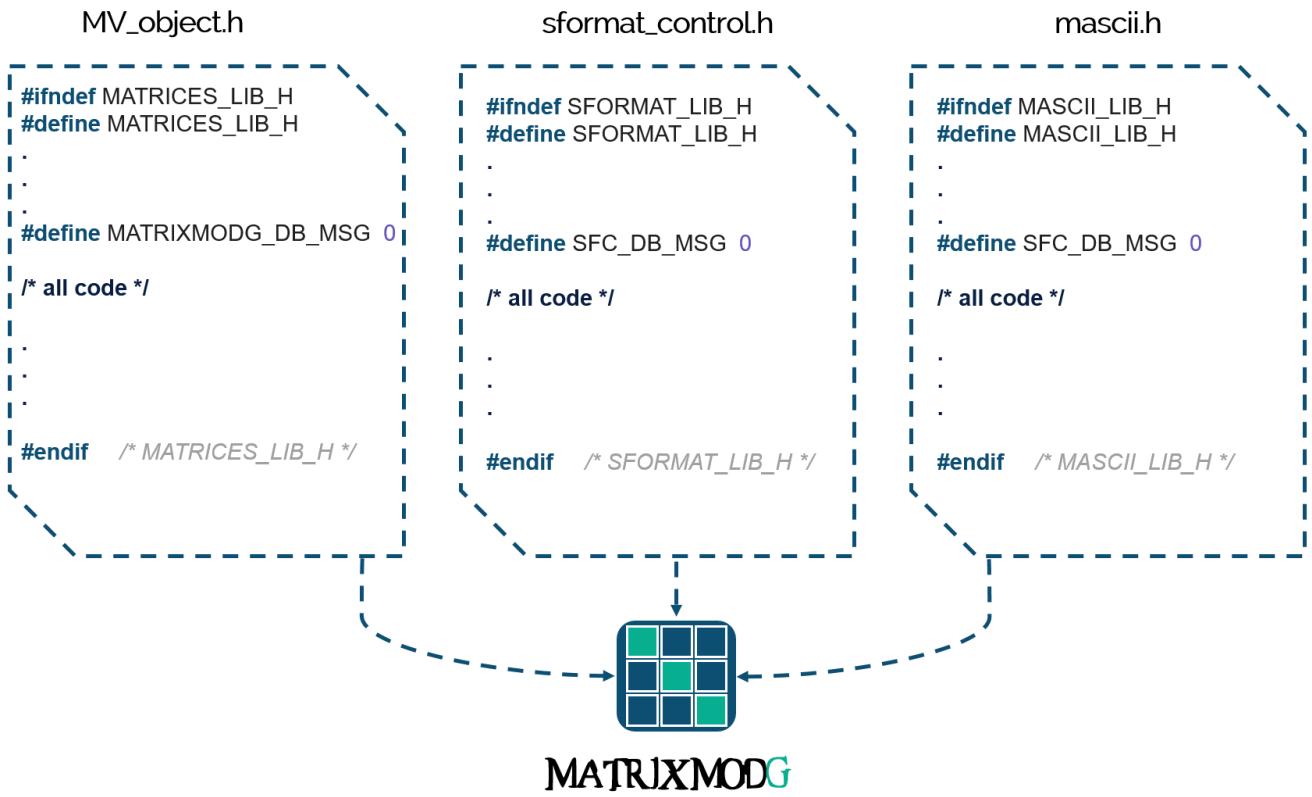


Figura 4.7 Macros de mensajes de debug.

Por otro lado es muy útil permitir deshabilitar los mensajes fuera del proceso de pruebas, ya que en este momento si todo funciona correctamente, ya no es necesario escribir los registros de movimientos del driver, sino más bien que este opere adecuadamente cuando se solicita la operación necesaria. Esto permite elevar significativamente el rendimiento del driver ya que, al no operar con la salida de diagnóstico, se ahorra tiempo de procesamiento dentro del kernel y solo se trabaja realizando la lógica de cada operación en particular para luego salir del kernel.

Este enfoque, de usar macros para la habilitación o deshabilitación de mensajes también fue utilizado en el espacio usuario para la creación de la librería del driver MatrixmodG como para la aplicación C++ que es utilizada para compararse con el driver.

4.2.5. DISPARO DE TRANSICIÓN

El método que dispara una transición de la RDPG, es un método clave en el diseño de la librería del driver matrixmodG ya que sobre este método se realizará el análisis del rendimiento existente para la gestión de una RDPG desde el espacio usuario usando el driver, como usando la aplicación C++. Además el método de disparo debe permitir el control de los procesos que interactúan con una RDPG, brindando eficiencia en la gestión de la RDPG, permitiendo dormir y despertar procesos solo cuando sea necesario.

Para esto se diseñó el diagrama de secuencia de la figura 4.8 que muestra la interacción que debe existir entre procesos (hilo 1 e hilo 2) y el recurso compartido, que es la RDPG protegida por un monitor.

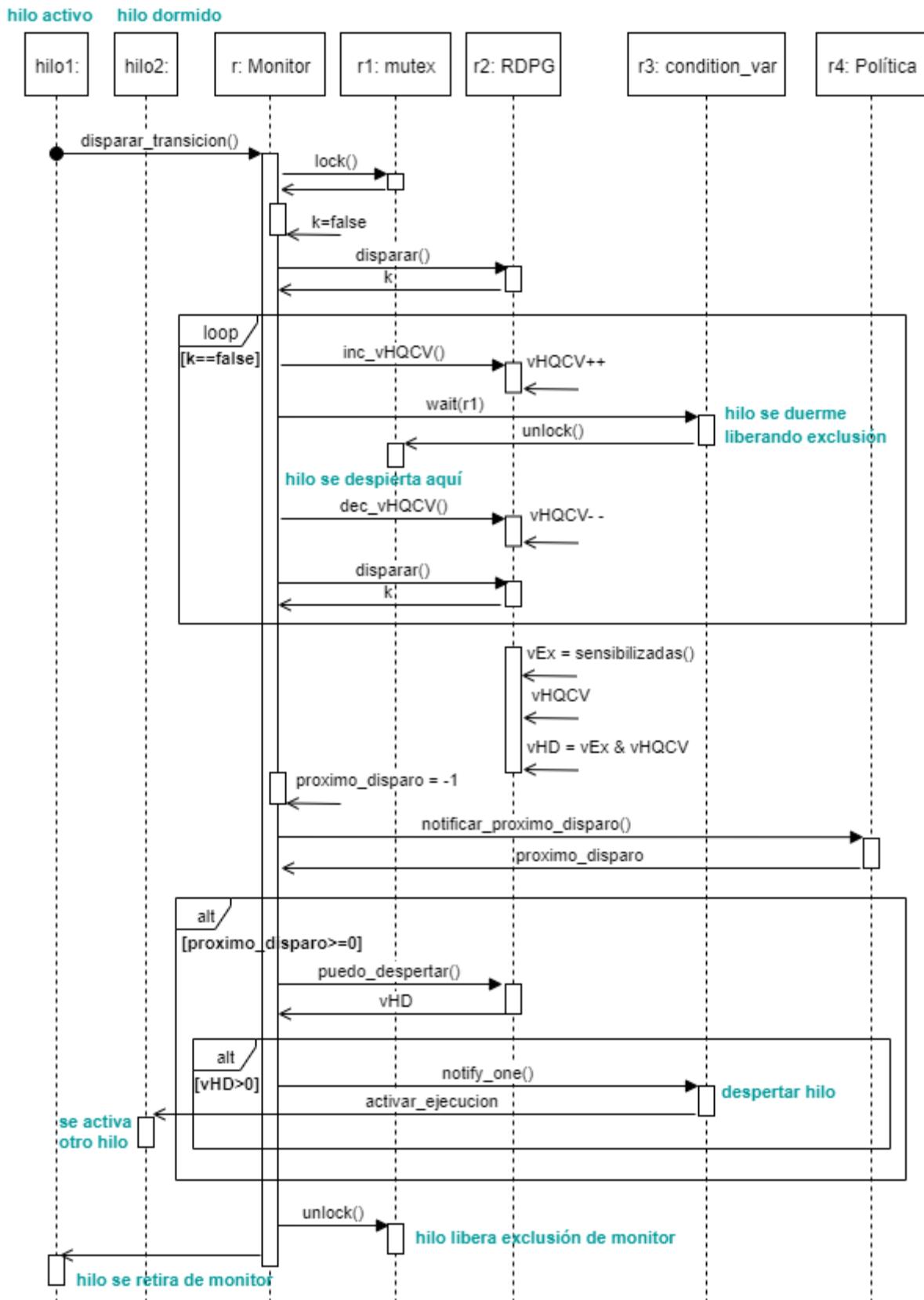


Figura 4.8 Diagrama de secuencia, método de disparo de transición.

Como se nota en el diagrama solo un proceso a la vez puede gestionar la RDPG, al solicitar el disparo de una transición se adquiere la exclusión mutua del monitor (mutex), se realiza el disparo sobre la RDPG y se guarda el resultado en una variable “k”, el disparo puede tener solo dos resultados, “exitoso” o “fallido”.

Si el disparo falló quiere decir que la transición no está sensibilizada por lo que el proceso debe esperar hasta que se sensibilice, para esto el proceso se debe dormir en la cola de la variable de condición de la transición a disparar ($CV[\text{transición}]$), pero antes de dormirse se debe liberar la exclusión mutua del monitor, así otros procesos podrán operar con la RDPG.

Por otro lado si el disparo de la transición es exitoso, cambia el estado de la RDPG, por lo que seguro otras transiciones están sensibilizadas, se verifica esta acción consultando el vector de transiciones sensibilizadas (vEx) y además se verifica si hay hilos dormidos en las colas a través del vector de colas de hilos en variables de condición ($vHQCV$). Realizando la operación “and” elemento a elemento entre cada uno de los vectores se obtiene un nuevo vector resultante (vHD) que muestra los hilos que pueden ser despertados ya que su transición está sensibilizada, y los hilos que no pueden ser despertados porque su transición no está sensibilizada o no hay hilos en la cola de la variable de condición asociada. Este último vector es útil para verificar con el próximo disparo a realizar que se conoce a través de la política, de manera tal que si es posible realizar el próximo disparo ($vHD[\text{próximo_disparo}] > 0$) antes de liberar la exclusión mutua se deberá despertar el hilo dormido en la variable de condición de la transición que ya está sensibilizada. Mientras que si no es posible realizar el próximo disparo ($vHD[\text{próximo_disparo}] == 0$) solo se libera la exclusión mutua del monitor. Todo este análisis muestra la eficiencia que se tiene en la gestión de procesos al interaccionar con la RDPG protegida por un monitor.

4.3. ARQUITECTURA DEL SISTEMA

4.3.1. FUNDAMENTOS

De acuerdo con la sección 8.3 sobre el diseño arquitectónico, se fundamenta cuáles son las decisiones tomadas para el diseño de la arquitectura del sistema desarrollado. En primer lugar se trata de una arquitectura en pequeño, que se basa en la arquitectura de los programas individuales que hacen al sistema y la forma en que cada uno de estos se separa en componentes.

La forma en que se pensó que será utilizada la arquitectura del sistema es “Como una forma de documentar una arquitectura del sistema que se diseña en el presente proyecto”. Se coincide con la idea de que la meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes del sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema. Esto es muy importante para poder llevar a cabo proyectos futuros que se pueden desprender del presente proyecto.

El patrón o estilo arquitectónico particular sobre el que se decide incluir la arquitectura del sistema es la arquitectura en capas a un bajo nivel de software ya que el sistema es pequeño. Esta arquitectura coincide con la arquitectura del kernel de Linux que es donde se centra la mayor parte del desarrollo del sistema, lo que motivó a ver el sistema partiendo desde esta arquitectura y analizando sus relaciones con los diferentes programas a los que se vincula.

La arquitectura depende de los requerimientos no funcionales de sistema, vinculados al rendimiento y la mantenibilidad tal como se analizó en la sección 3.1 con respecto a los requerimientos no funcionales y a los atributos que se buscan para el sistema.

Según la sección 8.3.2, se decidió realizar la vista lógica, la vista de desarrollo y la vista física del modelo 4+1 para la arquitectura del sistema. Estas vistas de la arquitectura del sistema serán muy útiles para los administradores, los desarrolladores y los ingenieros que desarrollan el sistema.

Para la representación de la vista física se utiliza un diagrama de arquitectura basado en la teoría de la arquitectura del kernel de Linux. Para la vista de desarrollo se utiliza un diagrama de bloques similar al diagrama de paquetes de UML. Por último, para la vista lógica se baja de nivel a la capa sobre la que está la mayor parte del trabajo, utilizando el diagrama de clases y el diagrama de objetos como parte de los modelos de estructura

brindados por UML, generando así la representación lógica de la arquitectura del sistema de acuerdo con los conceptos de diseño orientado a objetos descriptos en el capítulo de “Diseño orientado a objetos usando UML” propuesto por [21].

4.3.2. ARQUITECTURA DE ALTO NIVEL

Vista Física

Para representar la arquitectura del sistema desde su vista física se utilizó como referencia la arquitectura del kernel de Linux ya que el driver MatrixmodG forma parte de la ella siguiendo el mismo patrón de la arquitectura por capas.

De manera general la arquitectura del sistema se resume en dos capas importantes como muestra la figura 4.9, estas son:

- **Capa de presentación:** Es la capa de mayor nivel de software. En esta capa se programó la librería de espacio usuario (sección 4.4), la interfaz de usuario y la interfaz de pruebas de sistema. Esta capa permite automatizar muchas operaciones que se realizan sobre el driver para que sea todo más transparente y fácil para el usuario final del sistema. La interfaz se ejecuta en modo usuario.
- **Capa de lógica y datos:** Esta capa es la de menor nivel que tiene contacto directo con el hardware. En esta capa se programa el driver matrixmodG. Es la capa en donde se encuentran los datos y en donde se procesan y gestionan de acuerdo a todas las solicitudes enviadas por el usuario desde la interfaz de usuario utilizando llamadas al sistema. Aquí todas las operaciones se ejecutan en modo kernel.



Figura 4.9 Vista física general de la arquitectura del sistema.

Entrando en detalles la capa de presentación es todo programa que funcione como interfaz para comunicarse con el driver matrixmodG. Mientras que la capa de lógica y datos se divide en un conjunto de subcapas como se observa en la figura 4.10.

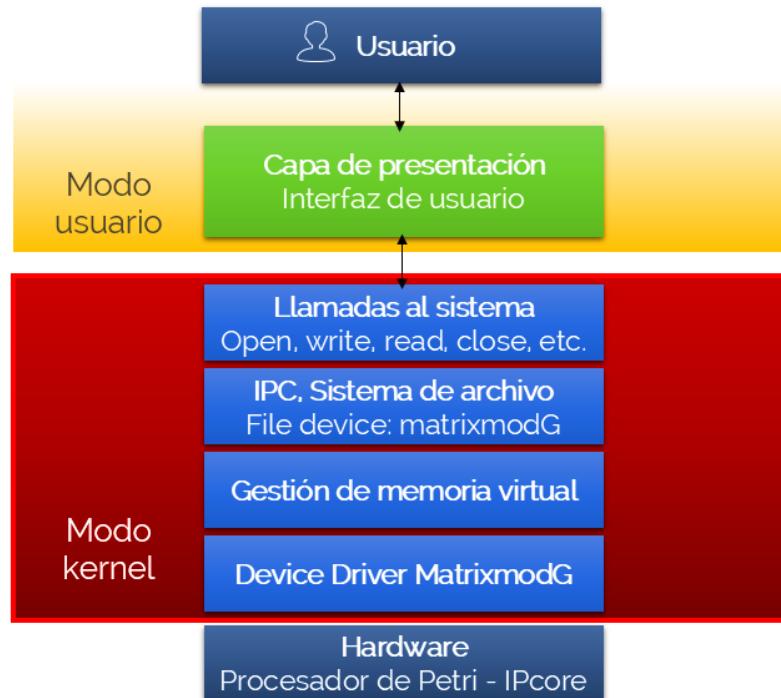


Figura 4.10 Vista física detallada de la arquitectura del sistema.

La capa de lógica y datos es la capa sobre la que se basó en gran porcentaje todo el trabajo de implementación del presente proyecto. Todo el esfuerzo y trabajo está sobre esta capa ya que es donde se buscan los objetivos y resultados de investigación y análisis del proyecto.

Específicamente, el trabajo de implementación en esta capa es sobre el driver MatrixmodG. Las llamadas al sistema funcionan como interfaz entre la interfaz de usuario y el driver. Aquí es donde se termina de utilizar el modo usuario y se pasa al modo de ejecución del kernel y viceversa.

Las subcapas de IPC (Comunicación entre procesos), Sistema de archivos y la capa de gestión de memoria virtual son todas transparentes para los desarrolladores, estas son manipuladas por el sistema operativo al igual que las llamadas al sistema. Por esta razón es que las podemos omitir en la arquitectura teniendo en cuenta que existen implícitamente como se hace en la figura 4.11.

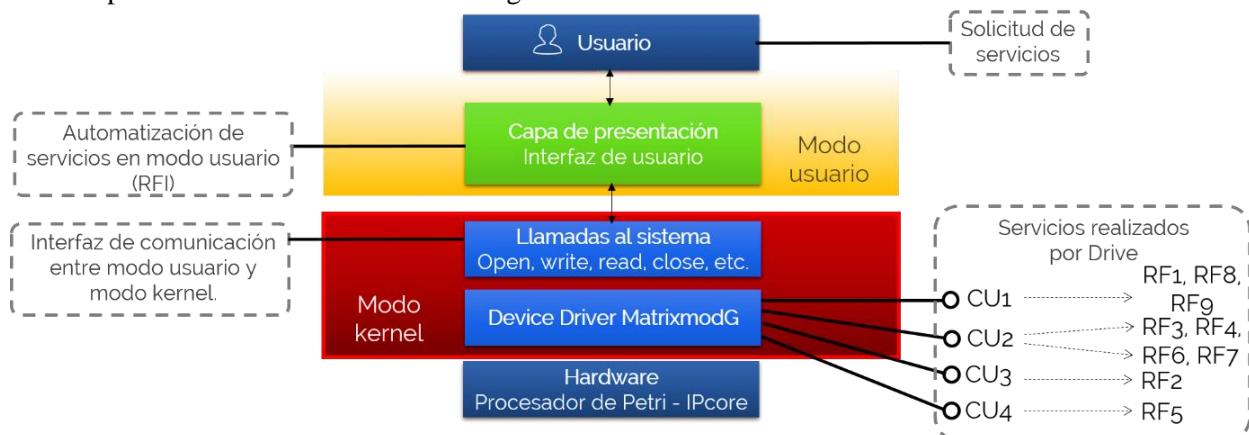


Figura 4.11 Vista física de la arquitectura del sistema y servicios brindados.

La figura 4.11 muestra la arquitectura de alto nivel del sistema vinculando los servicios prestados por el driver y su relación con los usuarios y el resto de las capas.

Esto es sencillo de explicar, el driver MatrixmodG es quien brinda todos los servicios buscados en el presente proyecto, estos se vinculan con el conector de punta circular. Los servicios son los casos de uso que se definieron

en la sección 3.1.3. A su vez se ven las relaciones de cada caso de uso con los requerimientos funcionales definidos en la sección 3.1.4, se utiliza el conector de inclusión haciendo referencia a que requerimientos incluye cada caso de uso.

Luego se observa que los usuarios son quienes realizan todas las solicitudes de los servicios requeridos. De por medio se tiene la interfaz de usuario encargada de tomar las solicitudes y enviarlas al driver mediante las llamadas al sistema. En la interfaz de usuario además de enviar las solicitudes requeridas por los usuarios se envían los datos previamente automatizados y preparados. También la información que proporciona el driver es procesada para mostrarla al usuario de manera adecuada.

Vista de Desarrollo

Teniendo en cuenta la vista física como base y siguiendo la misma lógica, para representar gráficamente el desarrollo de toda la implementación del sistema se realiza un diagrama de bloques que muestra la relación entre cada uno de los subsistemas del sistema general y el detalle de cada uno de estos subsistemas, representando de esta forma la arquitectura de alto nivel del sistema desde su vista de desarrollo.

La figura 4.12 es la vista de desarrollo de manera general donde podemos ver todo el desarrollo que se realizó en el sistema y su comunicación.

Tanto la implementación de la interfaz de usuario como la interfaz de pruebas, como cualquier otra aplicación que se ejecuta en el modo usuario, un ejemplo es una terminal bash de Linux. Como vimos anteriormente la interfaz encargada de pasar los datos y los procesos de modo usuario a modo kernel son las llamadas al sistema y son totalmente transparentes para los desarrolladores porque están son gestionadas por el sistema operativo, un desarrollador solo hace uso de las llamadas al sistema dentro de las implementaciones de las interfaces de usuario como también de los driver del kernel. Las llamadas al sistema son el medio de comunicación entre el espacio usuario y el espacio kernel, la información va y vuelve de un espacio a otro, esto se representa por las flechas de doble sentido.

La terminal bash se incorporó en el diagrama no porque se implementó, sino porque se puede hacer uso de esta aplicación para interactuar con el driver además de las interfaces implementadas. También se utilizó mucho la terminal bash en conjunto con el framework de pruebas KTF (Kernel Test Framework) para ejecutar pruebas unitarias e integrales sobre el driver MatrixmodG.

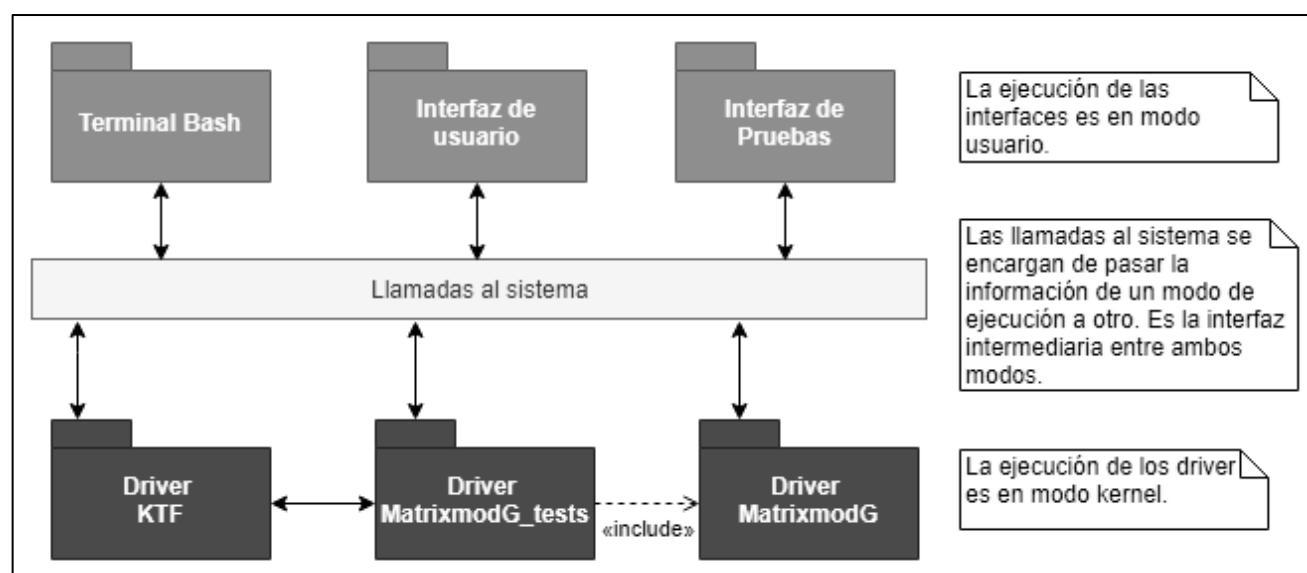


Figura 4.12 Arquitectura de alto nivel del sistema: Vista de desarrollo general.

Dentro del espacio kernel se ejecuta en modo kernel toda la implementación del driver MatrixmodG. Lo mismo sucede con el driver KTF y MatrixmodG_test, este último es un driver que incluye al driver MatrixmodG para crear y automatizar todas las pruebas unitarias e integrales sobre el mismo desde el kernel de Linux. Esto se logra con la ayuda del framework de pruebas KTF (Kernel Test Framework) con el que se comunica el driver MatrixmodG_test, dicha comunicación se representa por una flecha de doble sentido indicando la existencia de la comunicación entre el driver KTF y el driver de pruebas.

Por último es importante conocer todo el código implementado en el desarrollo del sistema a través de su arquitectura. Para esto se proporciona el detalle de cada uno de los subsistemas que conforman el sistema general, como se observa en la vista de desarrollo de la figura 4.12, separando sus elementos en tres listas diferentes, estas son:

- **Datos u objetos:** Muestra los datos y objetos con los que interacciona la implementación del subsistema. Cuando se refiere a datos son datos externos con los que interacciona un programa como archivos de texto, imágenes, etc. Cuando se refiere a objetos se habla de datos internos que maneja la implementación como variables, funciones y objetos importantes en su implementación.
- **Fuentes – Programas:** Es el conjunto de archivos implementados. Son los archivos fuentes o programas fuentes que conforman las diferentes capas de la implementación del subsistema. Por lo general en este caso son archivos de extensión .h/.hpp y .c/.cpp ya que la interfaz se armó en lenguaje C.
- **Binario – Ejecutable:** Es el archivo ejecutable que se obtiene como resultado de toda la implementación generada por las fuentes.

Las arquitecturas de los subsistemas son:

- **Arquitectura de Interfaz de usuario:** Se representa por la figura 4.13.

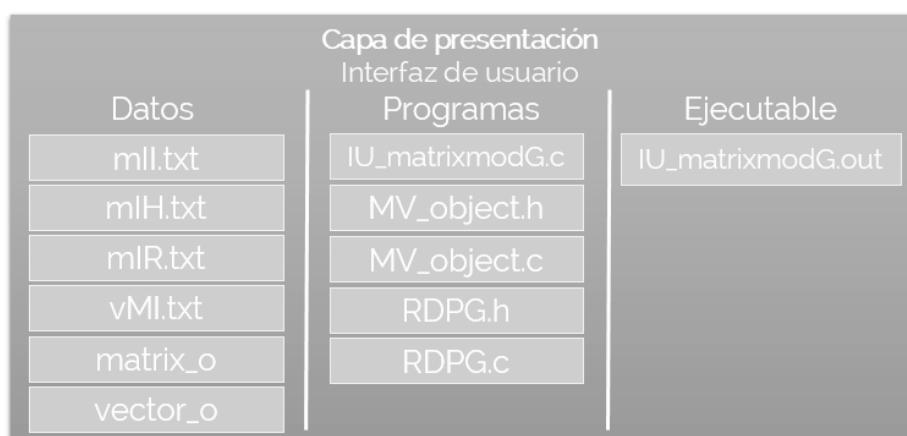


Figura 4.13 Arquitectura de Interfaz de usuario.

- **Datos:** Son los archivos de extensión .txt que contienen las matrices de incidencia I, H, R y Re y el vector de marcado inicial. Los datos internos manejados por la implementación son los objetos matrix_o y vector_o que son vectores y matrices de asignación dinámica donde se almacena la información de los archivos de texto para gestionarlos internamente.
- **Programas:** Los programas que conforman la implementación de la interfaz son cinco.
- **Ejecutable:** Se utiliza un solo ejecutable como resultado de la compilación de los sus programas.
- **Arquitectura de interfaz de Pruebas de sistema:** Se representa por la figura 4.14.
 - **Datos:** Los datos utilizados en la interfaz de pruebas son los mismos que los de la interfaz de usuario.
 - **Programas:** Los programas también son los mismos que los de la interfaz de usuario con la diferencia que se utilizan librerías de gestión de tiempo y un programa que gestiona semáforos para realizar pruebas de comparación.

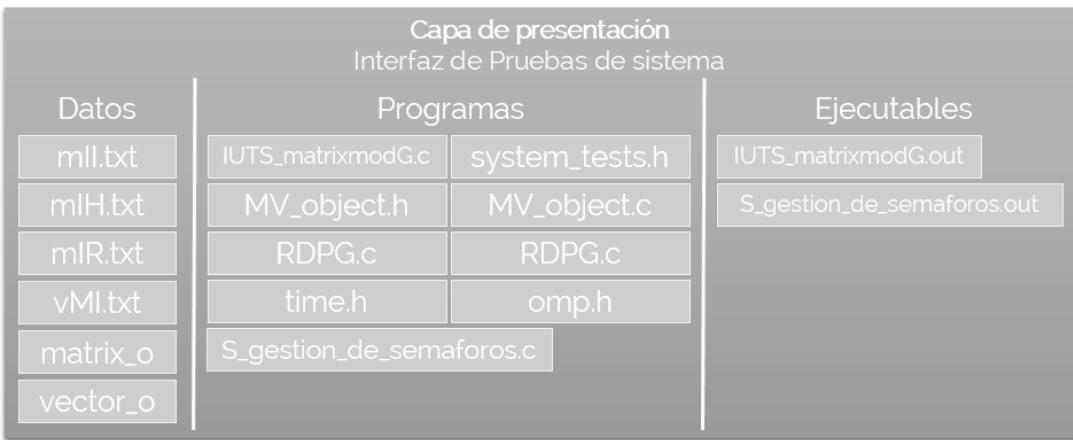


Figura 4.14 Arquitectura de Interfaz de pruebas de sistema.

- **Ejecutable:** Se utilizan dos ejecutables como resultado de la compilación de los sus programas, uno es el de la interfaz de pruebas y el segundo ejecutable es del programa que gestiona semáforos.
- **Arquitectura de Driver MatrixmodG:** Se representa por la figura 4.15.



Figura 4.15 Arquitectura de Driver MatrixmodG.

- **Datos:** Los datos son totalmente internos. El driver maneja objetos internamente creados a partir de las solicitudes indicadas por las interfaces de usuario. Los objetos son RDG_o que a su vez utiliza conjuntos de matrices y vectores (matrix_o y vector_o) para gestionar el estado de esta una RDG.
- **Programas:** Los programas que conforman la implementación de la interfaz son once. Para mayor detalle ver la tabla de estructura del driver en la sección 4.5.
- **Ejecutable:** Se utiliza un solo ejecutable como resultado de la compilación de los sus programas.

- **Arquitectura de Driver Matrixmod_tests:** Se representa por la figura 4.16.



Figura 4.16 Arquitectura de driver MatrixmodG_tests.

- **Datos:** Como este módulo incluye al módulo anterior, además de todos los datos de este último se adicionan los TEST y ktf_thread. Serían dos tipos de datos importantes para la gestión de cada una de las pruebas y para la creación de hilos en el kernel de Linux.
- **Programas:** Los programas que conforman la implementación de la interfaz son los del driver anterior más dos programas principales, el del propio driver de pruebas y el del framework KTF para obtener sus funciones de prueba y para establecer comunicación con este.
- **Ejecutable:** Se utiliza un solo ejecutable como resultado de la compilación de los dos programas.

Las arquitecturas de la terminal bash y del driver ktf no se muestran en detalle ya que no fueron implementadas en el presente proyecto sino que fueron implementaciones ya creadas y que se adicionaron al sistema para aumentar la calidad del software gracias a la incorporación de los casos de prueba unitarios e integrales que permite incorporar el framework KTF.

Vista Lógica

La mayor parte del trabajo se basó en la creación del driver matrixmodG, por esta razón es importante conocer su estructura interna para la cual se decidió hacerlo a través del diagrama de clases que muestra la estructura de cada una de las clases y el diagrama de objetos que muestra la estructura de las instancias de objetos en un momento de tiempo determinado, esto permite conocer cuáles son las instancias de objetos utilizadas y su comunicación según el diseño propuesto.

El diagrama de clase en su forma más simple es el de la figura 4.17 donde muestra la relación de las diferentes clases del diseño realizado, sus asociaciones y las diferentes enumeraciones asociadas a cada clase.

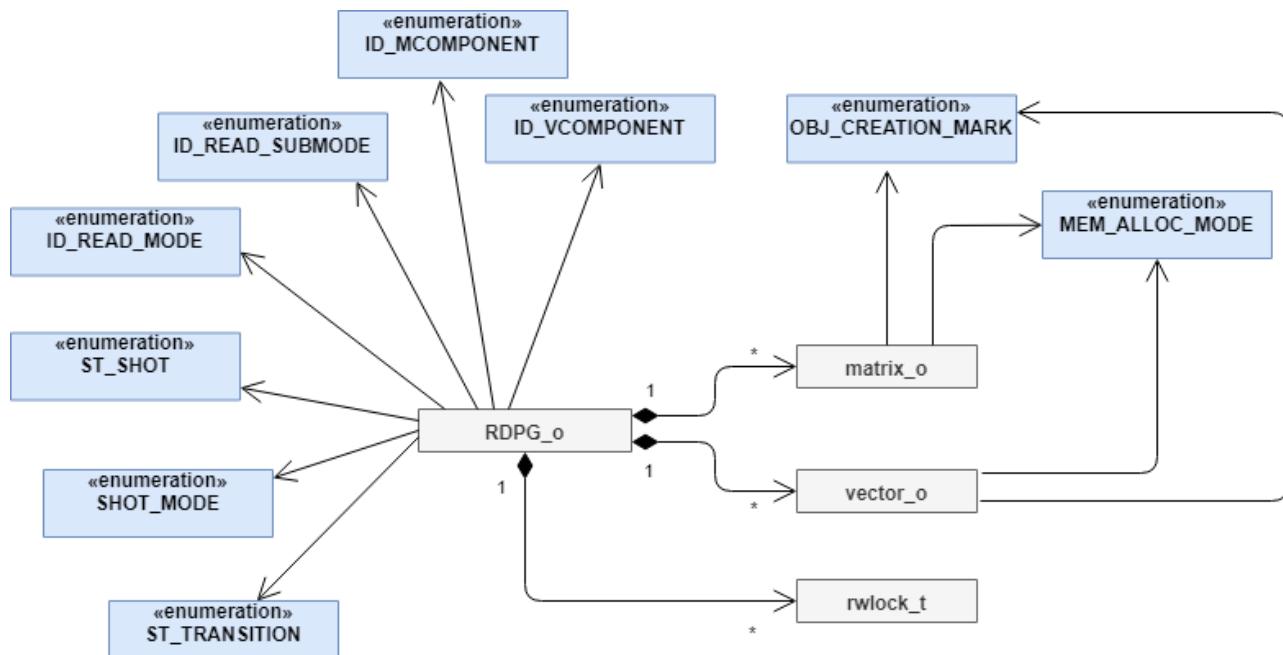


Figura 4.17 Diagrama de clase de manera general.

La clase principal es RDPG_o, esta clase es la que representa a un objeto RDPG al momento de instanciarse. En el diagrama de la figura 4.17 se puede observar las asociaciones de composición (arcos con extremo de rombo negro) que se desprenden de la clase RDPG_o hacia tres tipos de clases, que son la clase matrix_o, vector_o y rwlock. Estas últimas clases representan los objetos matrices, vectores y spinlock lector-escritor por los que se compone una RDPG_o al momento de instanciarse. Se observa como cada asociación tiene una correspondencia o multiplicidad entre las clases, para el caso de la clase RDPG_o indica que un objeto RDPG_o se compone de uno o muchos objetos matrix_o, lo mismo con objetos vector_o y con objetos rwlock. Por otro lado nuestro diseño utiliza un número limitado de objetos al momento de instanciarse una RDPG_o dependiente

de nuestras necesidades, es por esto que el diagrama de objetos muestra las instancias en tiempo de ejecución al momento de crear un objeto RDPG_o de acuerdo al diseño implementado (ver figura 4.19). Cada clase se relaciona con un conjunto de enumeraciones, esto se representa mediante asociaciones simples (sin ningún símbolo en los extremos) aunque se decidió agregar la flecha indicando la navegabilidad, tratando de representar que la clase RDPG_o es la que hace uso de las enumeraciones y no a la inversa, además de la existencia de una relación entre ambos elementos. El detalle de los valores de las enumeraciones se muestra en la figura 4.18.

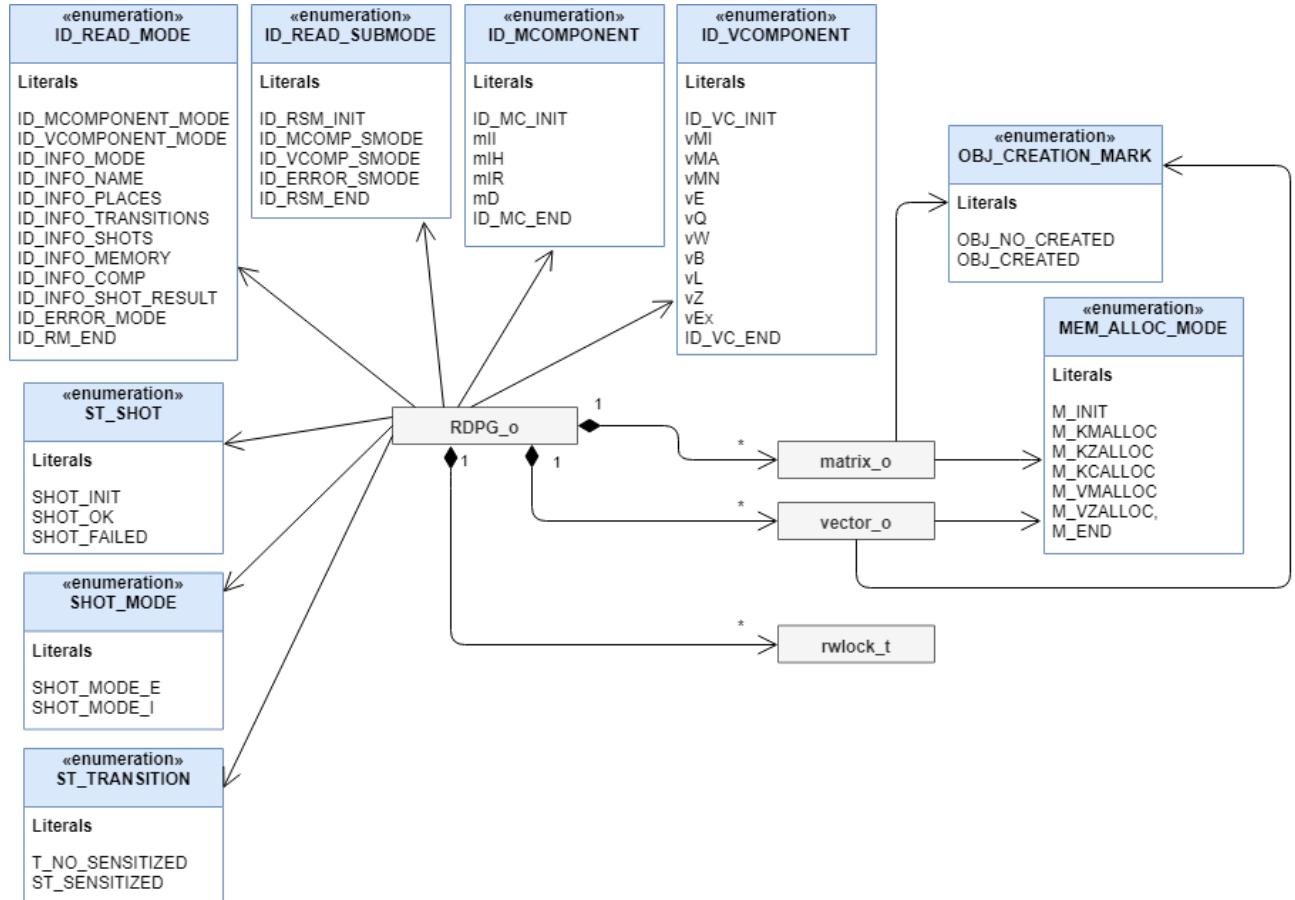


Figura 4.18 Detalle de enumeraciones en diagrama de clase.

Cada una de las enumeraciones utilizadas en la implementación de las diferentes clases aporta organización para la gestión de sus atributos y sus operaciones, brindando además mayor legibilidad al código para los desarrolladores. En el diseño las enumeraciones encapsulan un conjunto de datos que permiten saber:

- Si un objeto se encuentra instanciado en el kernel o no.
- El modo de reserva de memoria para la instancia del objeto en el kernel.
- Los identificadores con los que un objeto `RDPG_o` reconoce sus objetos `matrix_o` y `vector_o`.
- El modo y submodo de lectura seleccionado sobre el objeto `RDPG_o`.
- Estado de una transición.
- Estado de los disparo realizados.
- Modos de disparar una RDPG.

El detalle de los atributos y operaciones de cada una de las clases del diseño se presentan en la figuras 4.1, 4.2, 4.3, 4.4 y 4.5 de la sección 4.2.2.

Por último, el diagrama de objetos de la figura 4.19, muestra cómo se comporta el diseño en tiempo de ejecución al momento de instanciar una RDPG en el kernel de Linux.

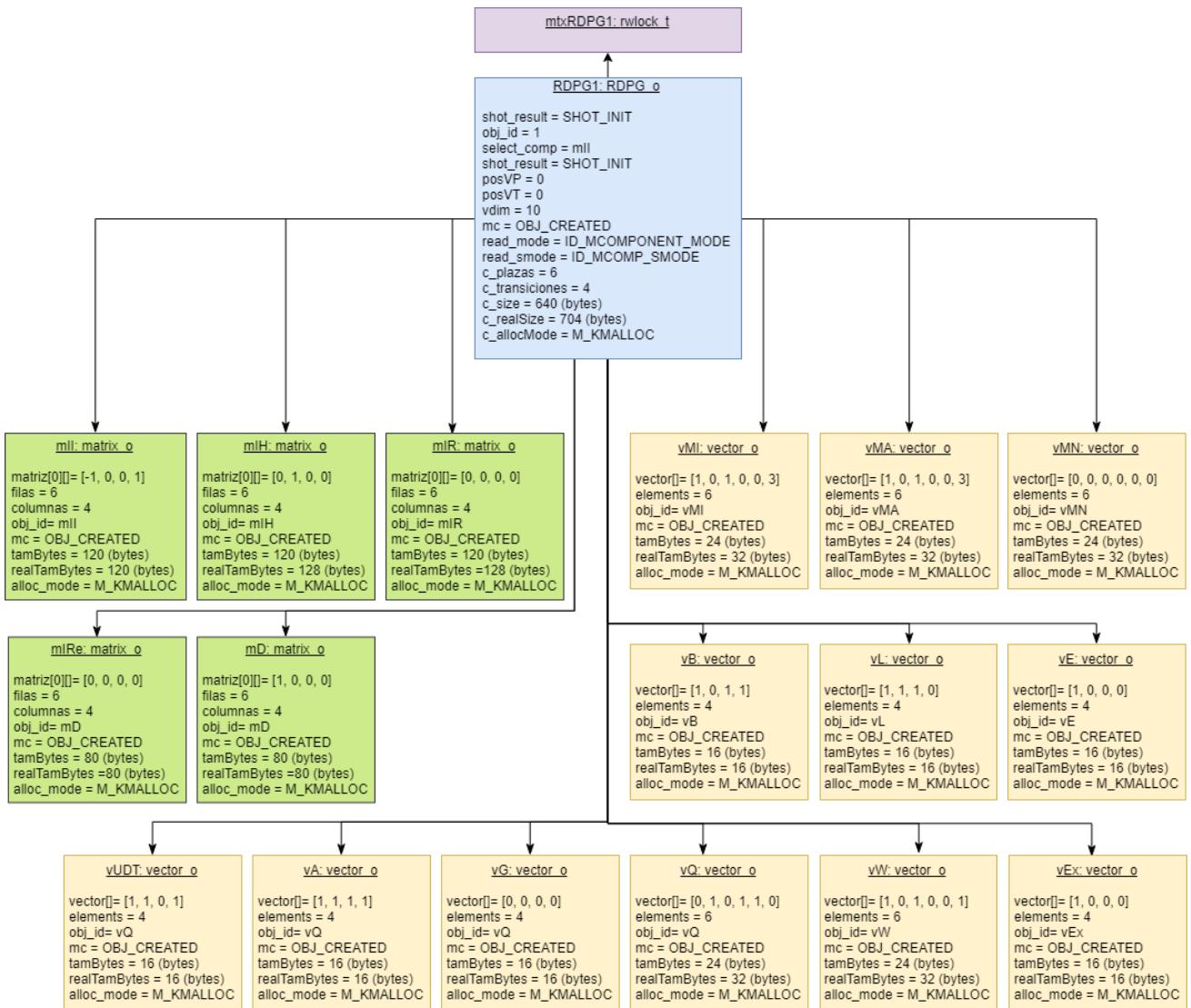


Figura 4.19 Diagrama de objetos de driver MatrixmodG.

Debe quedar claro que la diferencia entre el diagrama de clases y el diagrama de objetos es que el primero muestra la generalización del diseño desarrollado mientras que el diagrama de objetos muestra el diseño real al momento de instanciarse, es decir que muestra los límites actuales del diseño. Por ejemplo si se utilizan diez vectores para una RDPG el diagrama de objetos muestra las diez instancias mientras que si se agrega un vector extra se mostrarán once instancias al instanciar una RDPG. Por otro lado para la misma relación, RDPG y vectores, en el diagrama de clases nos indica que una RDPG puede componerse por todos los vectores que se deseen de 1 a muchos (indicado por el símbolo *).

Desde el espacio usuario, se hace uso de las RDPG, ya sea mediante la librería del driver MatrixmodG, o mediante la aplicación C++ de iguales características al driver para gestionar las RDPG. En ambos casos se buscó un modelo que permita gestionar las RDPG por múltiples procesos y con protección del recurso compartido (RDPG). Para esto se hizo uso de monitores que brindan las herramientas necesarias para garantizar exclusión mutua del recurso compartido. La figura 4.20 muestra una nueva abstracción de la arquitectura con el uso de monitores y sus componentes desde el espacio usuario.

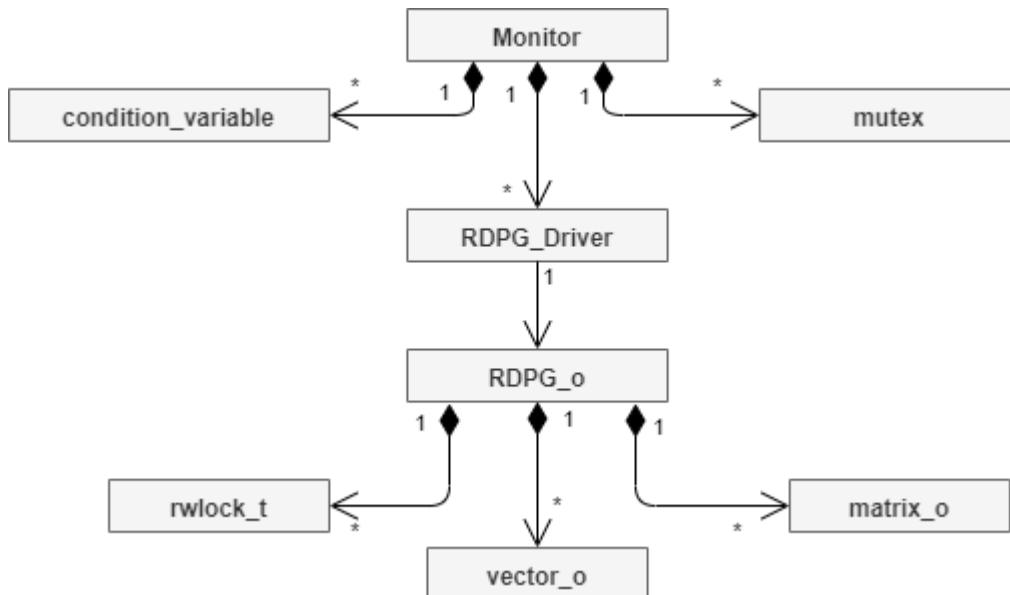


Figura 4.20 Diagrama de clases desde espacio usuario.

La idea del diagrama de clase de la figura 4.20 es mostrar cómo se tratan las RDPG desde el espacio usuario ya sea desde el driver MatrixmodG o desde la aplicación de C++. La clase principal en el espacio usuario es el Monitor y sus componentes, ya que el resto del diagrama fue explicado anteriormente. Los componentes del monitor son el mutex (encargado de proveer exclusión mutua a los procesos) y las variables de condición (quienes proveen las diferentes colas de procesos necesarias para una RDPG). Existe un único mutex dentro del monitor, ya que en el monitor, solamente un proceso puede operar con el recurso compartido. Las variables de condición coinciden con el número de transiciones de la RDPG que se esté utilizando, es decir que se trata de un vector de variables de condición, y representan las diferentes colas en donde puede dormirse un proceso. La clase RDPG_Driver, es la clase necesaria de la capa de aplicación que se encarga de tratar a las RDPG como objetos al momento de instanciarse y que funciona como interfaz para comunicarse con el kernel a través del uso de llamadas al sistema.

4.4. LIBRERÍA LIBMATRIXMODG DE ESPACIO USUARIO

El desarrollo de la librería LibMatrixmodG de espacio usuario, permite tres objetivos particulares:

- Hacer uso del driver matrixmodG desde el espacio usuario (por ejemplo programas de C/C++).
- Automatizar las llamadas al sistema con el driver de una manera sencilla y eficiente.
- Tener la posibilidad de medir el rendimiento del driver desde el espacio usuario.

El primer objetivo busca hacer uso del driver MatrixmodG en el código de programas C y C++. De esta manera los programas de espacio usuario que utilicen la librería obtienen todos los beneficios que provee el driver matrixmodG.

El segundo objetivo busca adicionar eficiencia y automatización que se logra con el uso de la librería haciendo todo más simple para un usuario final. La librería se diseñó para trabajar con RDPG de manera eficiente en el kernel de Linux desde el espacio usuario. La librería provee un objeto denominado Driver_RDPG_o (en librería de C) RDPG_Driver (en librería de C++), los cuales proveen todos los métodos necesarios para gestionar las RDPG en el kernel de Linux de una manera simple para el usuario final de la misma. Esto se logra mediante la automatización de llamadas al sistema entre el programa C/C++ con el file device del driver MatrixmodG, todo totalmente transparente para el usuario final (figura 4.21).

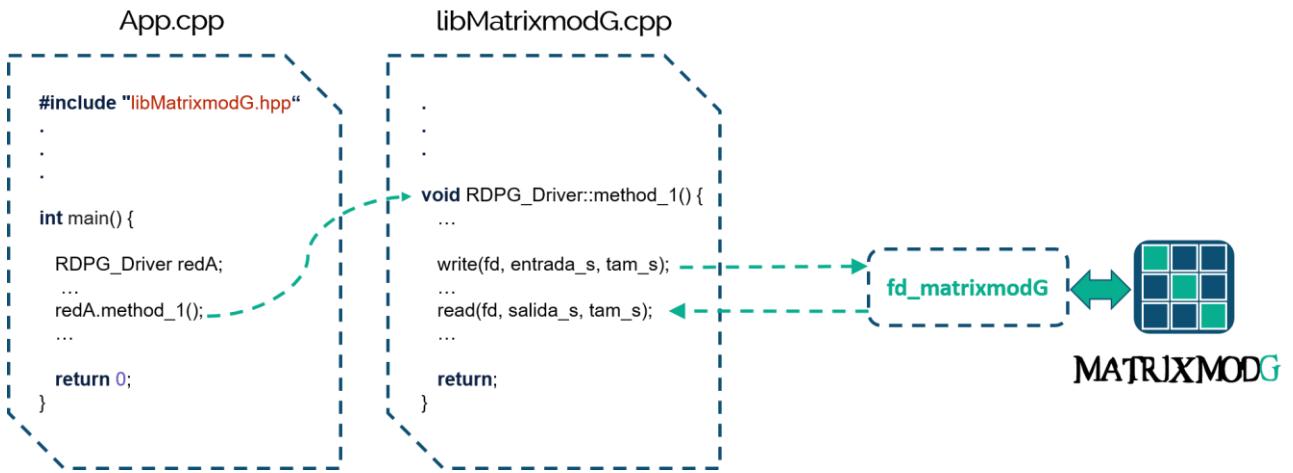


Figura 4.21 Métodos de objeto RDPG_Driver desde libMatrixmodG.

El tercer objetivo permite utilizar la librería para abstraer el objeto que gestiona las RDPG en el kernel de Linux logrando que sea simple realizar la medición de tiempos para obtener el rendimiento del driver MatrixmodG en cada una de sus operaciones.

4.5. DISPONIBILIDAD A MÚLTIPLES LENGUAJES

El driver MatrixmodG es una nueva funcionalidad disponible para todo el sistema operativo Linux, de manera que una aplicación C++ puede gestionar una RDPG en el kernel, teniendo el control completo sobre todos los subprocessos que interactúan con la RDPG del kernel. Este enfoque fue el implementado y comprobado en el capítulo 6 del presente proyecto tal como muestran las figuras 4.20 y 4.22.

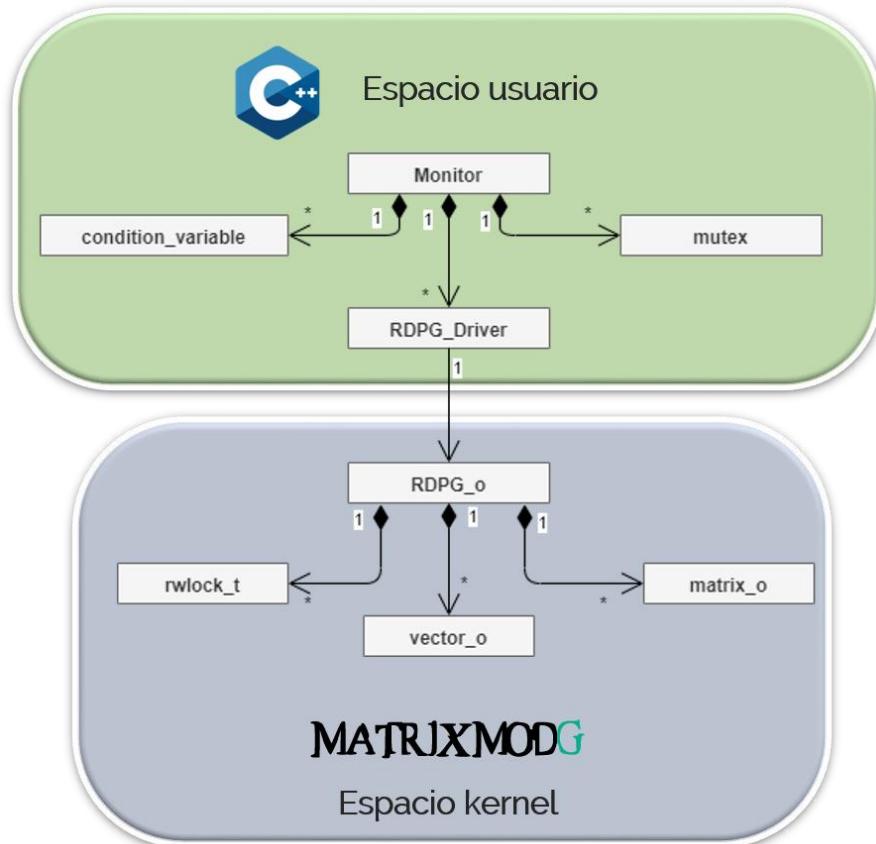


Figura 4.22 Diagrama de clases combinado (espacio usuario – espacio kernel).

Sin embargo, esta nueva funcionalidad sobre el sistema operativo Linux, permite que el driver sea accedido y controlado por múltiples aplicaciones de lenguajes diferentes al mismo tiempo desde el espacio usuario. Esto significa que dos o más aplicaciones de diferentes lenguajes pueden estar operando con una misma RDPG en el kernel mediante el uso del driver MatrixmodG, este nuevo paradigma de gestionar una RDPG del kernel por múltiples aplicaciones de lenguajes diferentes se muestra en la figura 4.23.

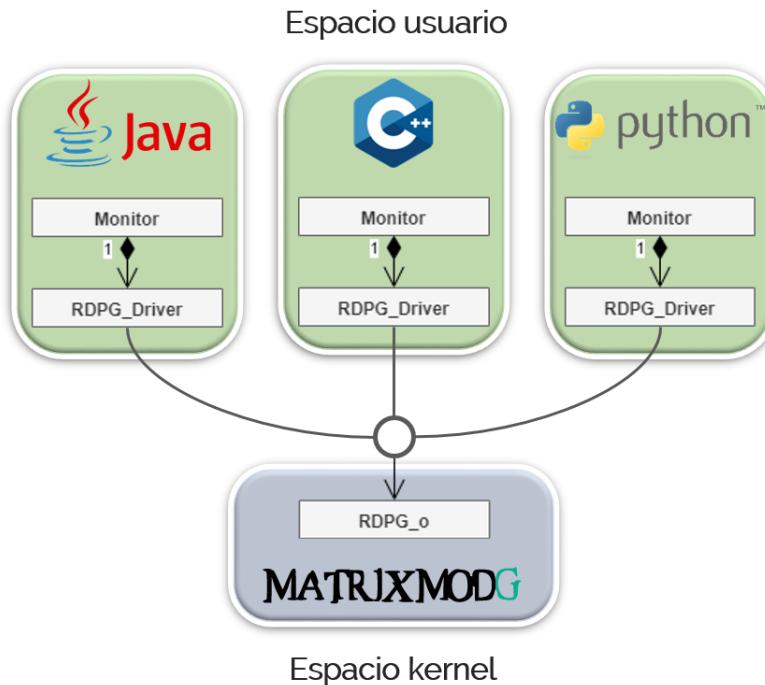


Figura 4.23 Extensión diagrama de clases a múltiples aplicaciones de lenguajes diferentes.

El diagrama de clases extendido de la figura 4.23 muestra el beneficio brindado por el driver MatrixmodG para ser utilizado por múltiples aplicaciones de diferentes lenguajes, pero presenta un problema que debe ser resuelto. El problema aparece cuando, en la gestión de una RDPG en el kernel por múltiples aplicaciones, la notificación del estado global del sistema simulado (vector de marcado actual) por la RDPG se informa solo al lenguaje que hace la última petición, pero se pierde para el resto de los lenguajes. Por esta razón se debe resolver el inconveniente descrito para obtener un funcionamiento óptimo y adecuado en el paradigma de gestionar una RDPG por múltiples aplicaciones de lenguajes diferentes.

5. PLAN DE PRUEBAS

Todo desarrollo de productos de software independientemente de la metodología que se está implementando, es necesario que se incluya la fase de pruebas, la cual nos permitirá determinar si el producto a entregar cumple con la calidad especificada y esperada. En la actualidad las personas dedicadas a las pruebas de software (Testing), necesitan de un Plan de Pruebas de Software, el cual tiene como propósito comunicar a todos los involucrados del proyecto los entregables, las características a ser o no ser probadas, las necesidades de entornos de trabajo para pruebas, etc [22].

El Plan de Pruebas de Software se puede aplicar a todo proyecto de software, se ajusta a las necesidades de cada organización de software considerando el tamaño del proyecto, el tiempo, el costo, el ciclo de vida del software, los involucrados, entre otras necesidades. Cada organización puede definir su propio Plan de Pruebas de Software basándose en las buenas prácticas y en la mejora continua.

5.1. INTRODUCCIÓN

5.1.1. OBJETIVO DEL PLAN DE PRUEBAS

El objetivo central que se busca con el plan de pruebas para el presente proyecto es describir los detalles más importantes del proceso de pruebas del sistema desarrollado, aclarando que es lo que se prueba, como se prueba y porque se prueba.

El objetivo es simple y claro, pero vale la pena descomponer los tres propósitos buscados.

El “porque se prueba” se relaciona con el interés de verificar el correcto funcionamiento de todos los requerimientos propuestos para el sistema desarrollado, verificar que los riesgos del proyecto están controlados y por último se busca garantizar mayor calidad en el producto final como consecuencia del proceso de pruebas.

El “cómo se prueba” son las aclaraciones referidas al alcance de las pruebas, que estrategia se utilizará para las pruebas, cuáles serán los entornos de las pruebas y los procedimientos de trabajo sobre cada nivel de pruebas.

Por último el “que se prueba” aclara las características de las pruebas y presenta las listas de los bancos de pruebas y cada uno de los casos de pruebas realizados en las diferentes etapas del desarrollo del sistema.

Todo lo anterior se complementa con los objetivos del proyecto integrador, donde se buscó desde el inicio conformar un sistema de software profesional.

5.1.2. ALCANCE DE PRUEBAS

Para nuestro proyecto se presentan y delimitan las pruebas a un alcance que cubre tres niveles de pruebas, estos son:

- Pruebas unitarias.
- Pruebas integrales.
- Pruebas de sistema.

El plan de pruebas para cada prueba finaliza cuando se cubren los tres niveles mencionados. Pero mientras se mantenga el sistema y continúe extendiéndose el plan de pruebas nunca finaliza ya que queda en un ciclo iterativo de mejora continua donde este puede seguir evolucionando.

5.1.3. ESTRATEGIA DEL PLAN DE PRUEBAS

Para cumplir con el objetivo principal del plan de pruebas se tomaran un conjunto de decisiones sobre las acciones y recursos a utilizar. Estas son:

- Las pruebas unitarias e integrales se automatizan para agilizar la gestión de las mismas.
- La automatización de las pruebas unitarias e integrales se realiza con el framework KTF disponible para código del kernel de Linux.
- Para las pruebas de sistema no es necesaria su automatización, basta con el cumplimiento del caso de prueba en el sistema desarrollado pero toda automatización posible será utilizada como complemento.
- En caso de usar automatización para las pruebas de sistema, se realiza con la ayuda de programas de complemento que satisfagan y se amolden a las necesidades requeridas.
- La especificación de las pruebas se realiza en todos los niveles, pruebas unitarias, integrales y de sistema.
- La documentación de la ejecución y resultados de las pruebas se realiza solo para las pruebas de sistema que requieran de un análisis detallado.
- El seguimiento y control de defectos se realiza para todas las pruebas de sistema que los detecten.

5.2. CARACTERÍSTICAS DE PRUEBAS

En esta sección se presentarán las características a ser probadas y las características a no ser probadas para el sistema desarrollado en el presente proyecto.

5.2.1. CARACTERÍSTICAS A SER PROBADAS

Las características a ser probadas se enlistan en la siguiente tabla.

Tabla 5.1: Características a ser probadas	
Característica - ID	Descripción
RF: Requerimientos funcionales	Una de las características principales por las que se creó el plan de pruebas y las respectivas pruebas es por la búsqueda de garantizar el correcto funcionamiento del sistema desarrollado de acuerdo con cada uno de sus requerimientos funcionales. Todos los requerimientos funcionales tendrán como mínimo un caso de prueba asociado. Y se buscara que todos los casos de pruebas asociados a esta característica pasen exitosamente.
FMEM: Fugas de memoria	Dentro de los requerimientos no funcionales y de acuerdo con el análisis de riesgo las fugas de memoria se decidieron priorizar para reducir su aparición (riesgo RC3), ya que de lo contrario el sistema estará operando inefficientemente. Por esta razón esta característica se tiene en cuenta y se genera un conjunto de casos de pruebas asociados a la misma para garantizar el control del riesgo.
MT: Multithreading (Protección de recursos compartidos)	Al igual que la característica anterior la protección de recursos compartidos por varios hilos simultáneamente reduce el riesgo RC5. Por esta razón esta característica también se tiene en cuenta para la generación de un conjunto de casos de pruebas asociados a la misma, permitiendo controlar el riesgo RC5 y garantizando una adecuada gestión de los recursos compartidos en el kernel de Linux.

REND: Rendimiento	Saber el rendimiento que brinda el sistema desarrollado es de gran interés y es uno de los atributos del sistema que se buscan para el mismo. Debido a esto es necesario un conjunto de casos de pruebas que nos permitan comparar los tiempos obtenidos con otros sistemas similares. El rendimiento también forma parte de los requerimientos no funcionales como se vio en la sección 3.1.5.
IEF: Interfaces externas (Funcionalidad)	Esta característica es muy amplia, porque pueden existir múltiples interfaces externas para el driver MatrixmodG una vez que se encuentra finalizada una versión del mismo. Lo importante es que toda interfaz externa realizada será el nexo entre el driver y un usuario que haga uso del mismo. Por esta razón se considera esta característica a generar casos de prueba solo si es necesario para garantizar que la interfaz opera adecuadamente entre el driver MatrixmodG y un usuario, ya que el driver independientemente garantiza su correcta funcionalidad. Esta característica es particular pero va de la mano con los requerimientos funcionales ya que se analiza la funcionalidad desde la interfaz externa.

5.2.2. CARACTERÍSTICAS A NO SER PROBADAS

Las características a no ser probadas se enlistan a continuación en la siguiente tabla.

Tabla 5.2: Características a no ser probadas	
Característica	Descripción
PIP: Presencia de inacción de procesos.	Como vimos en la sección de análisis de riesgos, el riesgo RC8 detecta la posibilidad de la presencia de inanición de los procesos escritores al utilizar spinlock lector-escritor para la protección de un objeto RDPG. Este riesgo fue añadido a la lista de riesgos confirmados prioritarios a tener en cuenta y controlarlo pero por los resultados obtenidos en las pruebas de la características a ser probadas no es necesario actualmente un conjunto de casos de pruebas para esta característica ya que como vimos en el análisis la probabilidad de ocurrencia es baja, por esta razón nos arriesgamos a las consecuencias que podrían presentarse en caso de presencia del riesgo y se decide seguir controlando el riesgo manteniéndolo en mira pero solo subirlo a la lista de características a ser probadas cuando la probabilidad de ocurrencia suba al nivel “Moderado”.
IEE: Interfaces externas (Errores por mal uso)	Como se menciona en la característica IEF de la tabla 5.1, debido a que esta característica es amplia, la parte de mal uso de las interfaces por medio de un usuario, uso inadecuado, envío de datos incorrectos desde espacio usuario, etc. No será controlada, por lo que no se generaran casos de prueba extras respecto a estas características. Se corre el riesgo de que por estos usos inadecuados falle el sistema de usuario, pero se garantiza que el sistema del kernel no fallará ya que en este si se tienen en cuenta todos los errores de datos y usos inadecuados que podrían presentarse.

5.3. ENTORNOS DE PRUEBAS

En esta sección se definen y documentan las características de los entornos de hardware y software necesarios para realizar la ejecución de las pruebas de software [23].

Se definen las técnicas de depuración y las herramientas necesarias para llevar adelante los diferentes niveles de pruebas considerados.

5.3.1. TÉCNICAS DE DEPURACIÓN EN EL KERNEL

La programación del kernel trae sus propios y únicos desafíos de depuración. El código del kernel no se puede ejecutar fácilmente en un depurador, ni se puede rastrear fácilmente, porque es un conjunto de funcionalidades no relacionadas con un proceso específico. Los errores en el código del kernel también pueden ser extremadamente difíciles de reproducir y pueden derribar todo el sistema, destruyendo así gran parte de la evidencia que podría usarse para rastrearlos [9].

Esta subsección presenta las técnicas utilizadas para monitorear el código del kernel y rastrear errores que se complementa con las herramientas de pruebas unitarias, integrales y de sistema como se verá en las próximas subsecciones.

Depuración por impresión

La técnica de depuración más común es el monitoreo por impresión, que en la programación de aplicaciones se realiza llamando a puntos printf. Cuando está depurando el código del kernel, puede lograr el mismo objetivo con printk.

Función printk

A diferencia de la función printf del espacio usuario, en el kernel printk permite clasificar los mensajes según su gravedad al asociar diferentes niveles de registro o Prioridades, con los mensajes. Por lo general, se indica el nivel de registro con una macro. Por ejemplo, KERN_INFO que vimos ante algunas de las declaraciones de impresión anteriores, es uno de los niveles de registro posibles del mensaje. La macro loglevel se expande a una cadena, que se concatena al texto del mensaje en tiempo de compilación; es por eso que no hay una coma entre la prioridad y la cadena de formato en los siguientes ejemplos. Aquí hay dos ejemplos de comandos printk, un mensaje de depuración (KERN_DEBUG) y un mensaje crítico (KERN_CRIT):

```
printk (KERN_DEBUG "Aquí estoy:% s:% i \ n", _FILE_ _, _LINE_ );
printk (KERN_CRIT "Estoy destrozado; desistiendo de% p \ n", ptr);
```

Hay ocho posibles niveles de cadenas de registro, definidas en el encabezado `<linux/kernel.h>`; Los enumeramos en orden de severidad decreciente son:

- **KERN_EMERG:** Se usa para mensajes de emergencia, generalmente aquellos que preceden a un choque.
- **KERN_ALERT:** Una situación que requiere acción inmediata.
- **KERN_CRIT:** Condiciones críticas, a menudo relacionadas con fallas graves de hardware o software.
- **KERN_ERR:** Se utiliza para informar condiciones de error; Los controladores de dispositivos suelen utilizar KERN_ERR para informar dificultades de hardware.
- **KERN_WARNING:** Advertencias sobre situaciones problemáticas que, en sí mismas, no crean problemas serios con el sistema.
- **KERN_NOTICE:** Situaciones que son normales, pero aun así dignas de mención. Se informan varias condiciones relacionadas con la seguridad en este nivel.
- **KERN_INFO:** Mensajes informativos. Muchos controladores imprimen información sobre el hardware que encuentran en el momento de inicio en este nivel.
- **KERN_DEBUG:** Se utiliza para la depuración de mensajes.

Cada cadena (en la expansión de macros) representa un número entero entre paréntesis angulares. Los enteros varían de 0 a 7, con valores más pequeños que representan prioridades más altas.

Una declaración printk sin prioridad especificada hace uso de la prioridad predeterminada DEFAULT_MESSAGE_LOGLEVEL, definida como un entero en kernel/printk.c. En el kernel 2.6.10, DEFAULT_MESSAGE_LOGLEVEL es KERN_WARNING, pero se sabe que eso cambió en el pasado.

Según el nivel de registro, el núcleo puede imprimir el mensaje a la consola actual, ya sea un terminal de modo de texto, un puerto serie o una impresora paralela. Si la prioridad es menor que la variable entera console_loglevel, el mensaje se entrega a la consola una línea a la vez (no se envía nada a menos que se proporcione una nueva línea). Si tanto klogd como syslogd se ejecutan en el sistema, los mensajes del kernel se agregan en /var/log/messages (o se tratan de otro modo según su configuración de syslogd), independientemente de console_loglevel. Si klogd no se está ejecutando, el mensaje no llegará al espacio del usuario a menos que lea /proc/kmsg (que a menudo se hace más fácilmente con el comando dmesg). Cuando use klogd, debe recordar que no guarda líneas idénticas consecutivas; solo guarda la primera línea de este tipo y, posteriormente, la cantidad de repeticiones que recibió.

La variable console_loglevel se inicializa DEFAULT_CONSOLE_LOGLEVEL y se puede modificar a través de la llamada al sistema sys_syslog. Una forma de cambiarlo es especificando el modificador -c al invocar klogd, como se especifica en la página de manual de klogd. Tenga en cuenta que para cambiar el valor actual, primero debe eliminar klogd y luego reiniciarlo con la opción -c. Alternativamente, puede escribir un programa para cambiar el nivel de registro de la consola. Encontrará una versión de dicho programa en misc-progs/setlevel.c en los archivos fuentes provistos en el sitio FTP de O'Reilly. El nuevo nivel se especifica como un valor entero entre 1 y 8, ambos inclusive. Si está configurado en 1, solo los mensajes de nivel 0 (KERN_EMERG) llegarán a la consola; si está configurado en 8, se muestran todos los mensajes, incluidos los de depuración.

También es posible leer y modificar el nivel de registro de la consola utilizando el archivo de texto /proc/sys/kernel/printk. El archivo aloja cuatro valores enteros: el nivel de registro actual, el nivel predeterminado para los mensajes que carecen de un nivel de registro explícito, el nivel de registro mínimo permitido y el nivel de registro predeterminado de tiempo de arranque.

```
# cat /proc/sys/kernel/printk
4 4 1 7
```

Escribir un solo valor en este archivo cambia el nivel de registro actual a ese valor; así, por ejemplo, puede hacer que todos los mensajes del kernel aparezcan en la consola simplemente ingresando:

```
# echo 8 > /proc/sys/kernel/printk
```

5.3.2. HERRAMIENTAS DE PRUEBAS UNITARIAS E INTEGRALES

Para la ejecución de las pruebas unitarias e integrales se utilizaron las diferentes técnicas descriptas en la subsección anterior y la herramienta Kernel test Framework (KTF) que permite obtener automatización de pruebas unitarias e integrales con varias funcionalidades y características muy útiles y simples de utilizar.

Kernel Test Framework

El Kernel Test Framework implementa un entorno de trabajo de pruebas unitarias para el kernel de Linux. Existe una amplia selección de frameworks de pruebas unitarias disponibles para las pruebas de código del espacio usuario, pero hasta ahora no se ha visto ningún marco similar que se pueda usar con el código del kernel, para

probar los detalles de las API del kernel exportadas y no exportadas. La esperanza es que al proporcionar una manera fácil de usar y conveniente de escribir pruebas unitarias simples para los componentes internos del kernel, esto pueda promover un enfoque más orientado a las pruebas para el desarrollo del kernel, cuando sea apropiado [24].

Un objetivo de diseño importante es hacer que KTF se adapte bien a un ciclo normal de desarrollo del kernel, y que se integre bien con las pruebas unitarias del espacio usuario, para permitir que las pruebas del kernel y el espacio usuario se comporten, se vean y se sientan tan similares como sea posible. Se espera que esto sea más intuitivo de usar y más gratificante. También se cree que incluso una prueba del kernel que pase debe tener un resultado amigable, fácil de leer y agradable, y que un marco de prueba debe tener buena observación, que es un buen mecanismo para depurar lo que salió mal, tanto en el caso de errores en las pruebas y el propio framework de prueba.

KTF está diseñado para probar el kernel de la misma manera que se ejecuta. Esto significa que queremos evitar cambiar las opciones de configuración, o realizar cambios que dificulten la lógica de decir desde un punto de vista de alto nivel si el kernel con KTF es realmente “lo mismo” lógicamente que el kernel al que están expuestos nuestros usuarios. Por supuesto, todos sabemos que es muy difícil probar algo sin afectarlo, con la mecánica cuántica como extremo, pero al menos queremos hacer un esfuerzo para que la huella sea lo más pequeña posible.

KTF prueba el código del kernel ejecutando pruebas en el contexto del kernel, o en el caso de pruebas híbridas, tanto en el contexto del usuario como del kernel. Al hacer esto, nos aseguramos de que probemos las rutas de código del kernel de manera real, sin emular un entorno de ejecución del kernel. Esto nos da mucho más control sobre lo que pueden hacer las pruebas en comparación con las pruebas impulsadas por el espacio usuario, y aumenta la confianza de que las pruebas coinciden con lo que hace el kernel ya que el entorno de ejecución de pruebas es idéntico.

KTF es un producto de una refactorización de código utilizado como parte del desarrollo de prueba de un controlador de Linux para un HCA Infiniband. Está en uso activo para las pruebas de componentes del kernel dentro de Oracle.

¿Porque KTF?

Se decidió utilizar KTF como entorno de pruebas unitarias e integrales y para la automatización de estas pruebas porque es una de las pocas herramientas existentes a la fecha para el kernel de Linux, junto con la reciente aparición de KUnit. KTF garantiza ser una herramienta muy buena en esta actividad ya que se basó de la evolución de herramientas del espacio usuario como lo es GTEST (GoogleTest).

Investigando las herramientas y formas de probar el código del kernel de Linux, como lo son sus drivers, hasta hace muy poco tiempo las pruebas se realizaban en el espacio usuario ya que en este espacio existe una variedad muy amplia de herramientas para pruebas, el método consiste en llevar la lógica del kernel al espacio usuario simulando su funcionamiento y verificándolo desde este nivel. Se puede decir que esta forma de trabajar es una de las posibilidades útiles, fáciles, y que más se aproxima a lo que se desea para la automatización de pruebas y la búsqueda de calidad en un sistema desarrollado en el kernel, sin embargo tiene la desventaja de duplicación de código además de que la prueba no se está probando en el entorno real sobre el que se desempeña la ejecución de los drivers de Linux [25].

KTF busco elevarse a esta idea, donde las pruebas sean realmente ejecutadas en el kernel además de proveer todas las ventajas que tiene un entorno de trabajo de prueba como los del espacio usuario siendo fácil y amigable para su uso, para la presentación de los resultados y gestión de pruebas. Además de todo esto KTF es candidata por ser una herramienta que provee documentación de sus características, pasos de instalación, modos de uso, ejemplos de uso, formas de debug, entre mucha más información de importancia disponible para quienes desean

utilizar la misma. Es una herramienta open source, evita el código duplicado y simulado en el espacio usuario, permite la gestión de las pruebas desde el espacio usuario a través del debugfs (debug file system), provee de un analizador de cobertura y permite la ejecución de hilos en el espacio kernel.

Sin duda para las pruebas unitarias e integrales, todo lo propuesto por KTF nos llevó a probar nuestro sistema con dicho entorno de trabajo.

5.3.3. HERRAMIENTA DE PRUEBAS DE SISTEMA

Al finalizar las pruebas unitarias e integrales, se procede con la ejecución de las pruebas de sistema. Para la ejecución de las pruebas de sistema se creó una interfaz de usuario de pruebas de sistema (IUTS) que ayuda a la automatización de las mismas desde el espacio usuario. Por otro lado para las pruebas relacionadas a la verificación de que el sistema desarrollado está libre de fugas de memoria se utilizó el framework “KErnel-mode Drivers in Runtime” (KEDR) como herramienta de complemento para este propósito.

Programa IUTS

La interfaz de usuario IUTS se creó para probar el módulo MatrixmodG a nivel de sistema y para automatizar operaciones hacia este, agilizando ciertas funcionalidades. IUTS también implementa todos los casos de pruebas de sistema que pueden ser cubiertos por este.

De manera general IUTS prueba la biblioteca de espacio usuario que hace uso del driver MatrixmodG para la gestión de RDPG dando soporte a las pruebas de requerimientos funcionales, a la característica multithreading (protección de recursos compartidos por el drive), a las pruebas de rendimiento y a las pruebas de funcionalidades de las interfaces externas.

El soporte a las pruebas de requerimientos funcionales se realiza ejecutando todas las operaciones o funcionalidades que proporciona la biblioteca de espacio usuario que permite el manejo del driver MatrixmodG.

Para dar soporte a las características multithreading del driver se utiliza la API OpenMP que brinda un conjunto de funcionalidades útiles para la gestión de subprocessos y de esta forma probar la respuesta del driver y su protección de sus recursos compartidos mediante su exposición a múltiples hilos paralelos enviados desde el espacio usuario.

Para probar el rendimiento del driver se provee soporte mediante la medición de los tiempos de cada una de las operaciones y su comparación con otros programas con las mismas características. La medición de los tiempos desde el espacio usuario se realiza con dos herramientas diferentes una es la ya nombrada API OpenMP que también brinda funciones para la medición de tiempos de ejecución de una operación y además se utiliza como complemento la interfaz time.h. Ambas herramientas brindan buenos resultados para la medición de los tiempos aunque de acuerdo al análisis de resultados los de la interfaz time.h son más precisos.

Por último la interfaz IUTS funciona como una interfaz externa al driver MatrixmodG por lo que brinda el soporte a cómo debe comportarse una interfaz externa de espacio usuario para gestionar la biblioteca del driver MatrixmodG y de esta formar gestionar las RDPG.

Kernel-mode Drivers in Runtime

El framework KEDR proporciona herramientas para facilitar el análisis en tiempo de ejecución de los módulos del kernel de Linux (incluidos los controladores de dispositivos, los módulos del sistema de archivos, etc.). “KEDR” es un acrónimo de “KE rnel-mode D rivers in R untime”. El entorno está destinado a los

desarrolladores de módulos de kernel y, en particular, puede ser útil para construir sistemas de verificación automatizados para software en modo kernel [26].

Las herramientas proporcionadas por KEDR operan en un módulo del kernel elegido por el usuario (un módulo destino). Recopilan información sobre las llamadas a las funciones del kernel que realiza el módulo y lo envían a la traza, realizan una simulación de fallas, detectan pérdidas de memoria, etc. Esto puede complementar las herramientas existentes para la inyección de fallas, la verificación de memoria, etc., que generalmente operan en el kernel de Linux en su conjunto.

El uso típico de KEDR es el siguiente. El usuario trabaja con el módulo destino del kernel normalmente o tal vez ejecuta algunas pruebas específicas en él. Al mismo tiempo, las herramientas KEDR están monitoreando el funcionamiento del módulo, verificando si funciona correctamente, realizando una simulación de fallas si se solicita, descargando los datos sobre las acciones realizadas por el módulo al archivo de rastreo para su análisis futuro, etc.

El framework KEDR está diseñado para ser fácil de ampliar y desarrollar. Por ejemplo, el conjunto de funciones del kernel a monitorear es completamente personalizable; Los escenarios de simulación de fallas también pueden ser proporcionados y controlados por el usuario. Además de eso, KEDR presenta una arquitectura basada en complementos y proporciona una interfaz para implementar tipos personalizados de recopilación y análisis de datos además de los mencionados anteriormente.

Actualmente, KEDR trabaja en arquitecturas x86 y x86-64.

KEDR es software libre y se distribuye bajo los términos de la Licencia Pública General de GNU Versión 2.

¿Porque KEDR?

Se decidió utilizar KEDR ya que sus características se adaptan a las necesidades buscadas, su herramienta LeakCheck de adapta para las pruebas de pérdidas de memoria, es por lo cual se decidió utilizar este entorno de pruebas para dicha característica.

Investigando las diferentes herramientas para detectar fugas de memoria en el espacio kernel no se encontraron muchos caminos, de los cuales los más prometedores fueron dos: la herramienta Kmemleak (Kernel Memory Leak Detector) y la herramienta LeakCheck de KEDR.

La funcionalidad LeakCheck del framework KEDR y Kmemleak, tienen diferentes habilidades hasta cierto punto. Ninguna de estas herramientas es estrictamente superior a la otra.

Algunas de las diferencias entre estos dos sistemas se describen a continuación, sin ningún orden en particular.

Kmemleak (versión de kernel 3.0.3 y mayor)	KEDR (componente LeakCheck)
Funciona de manera similar a un recolector de basura, escaneando la memoria en busca de objetos huérfanos en intervalos de tiempo regulares. Se realiza un seguimiento de las funciones de asignación de memoria, desasignación y se actualiza la colección de los objetos asignados pero no liberados.	Intercepta las llamadas a las funciones de asignación de memoria y desasignación, realiza un seguimiento del conjunto de los bloques de memoria asignados que aún no se han liberado. No escanea la memoria en busca de punteros a estos bloques.
Puede detectar pérdidas de memoria incluso si el componente que se está analizando todavía está cargado y funcionando en el kernel.	Reporta las pérdidas de memoria solo después de que se descarga el componente que se está analizando (módulo del kernel, en este caso).
Puede ser utilizado durante el inicio del sistema.	No se puede utilizar durante el inicio del sistema.

Funciona en el núcleo en su conjunto. Es difícil restringir el análisis a un módulo o grupo de módulos en particular.	Opera solo en el módulo del kernel dado, el resto del kernel no se ve afectado. Actualmente, KEDR no se puede aplicar a varios módulos del kernel al mismo tiempo.
Las asignaciones de página y el mapa del mapa no se rastrean.	Se rastrean las asignaciones de páginas, actualmente no se admite el mapa del mapa de seguimiento (pero esto se puede implementar fácilmente utilizando la infraestructura existente que proporciona KEDR).
Si Kmemleak no está habilitado de forma predeterminada en el archivo .config para el kernel, el kernel debe reconstruirse antes de que se pueda usar Kmemleak.	Generalmente, la reconstrucción del kernel no es necesaria para poder configurar, iniciar y detener KEDR.

Tabla 5.3 Comparación Kmemleak vs. LeakChek (KEDR).

La comparación de las diferentes características de cada una de las herramientas permitió detectar dos características que fueron decisivas para inclinar la elección al framework KEDR. Las características son las remarcadas en negrita:

- La primera característica a favor para KEDR que se adaptó a las necesidades buscadas es la de tener la posibilidad de monitorear las pérdidas de memoria solo para un componente destino del kernel como lo es para el caso del driver MatrixmodG.
- La segunda característica también se adaptó a la necesidad de no perder demasiado tiempo en reconstruir el kernel como riesgo de utilizar Kmemleak, es por lo cual se decidió apostar por KEDR ya que se basa simplemente en instalar un módulo nuevo al kernel para su uso.

5.4. PROCEDIMIENTO DE TRABAJO SOBRE LAS PRUEBAS

Esta sección se describe el procedimiento de trabajo sobre los diferentes niveles de los casos de pruebas realizados para el sistema desarrollado en el presente proyecto.

El proceso de cada una de las pruebas se compuso de las siguientes etapas:

- Especificación de caso de prueba.
- Ejecución del caso de prueba.
- Análisis y registro de resultados.
- Gestión y reporte de defectos.

5.4.1. ESPECIFICACIÓN DE CASO DE PRUEBA

De acuerdo con la estrategia del plan de pruebas el proceso de un caso de pruebas comienza con su especificación.

Para los casos de pruebas unitarios e integrales se definieron bancos de pruebas que encapsulan un conjunto de casos de prueba asociados. Se crearon tablas que enlistan el nombre del banco de prueba, descripción y propósito del banco de prueba, y la vinculación con elementos del proyecto de software (caso de uso, riesgo, requerimiento funcional, etc.).

Definido el banco de prueba se definieron los casos de pruebas unitarios e integrales enlistándolos en tablas que detallan su ID y nombre asociado, el propósito del caso de prueba y el resultado obtenido en su ejecución (PASO o FALLO).

Por ultimo para los casos de pruebas de sistema se documentó mucho más que las pruebas unitarias e integrales. La especificación de los casos de prueba de sistema se realizaron sobre tablas con mayor nivel de detalle dejando registro de gran cantidad de información relevante en la etapa de especificación. En las mismas tablas se dejó lugar para el registro de los resultados y la vinculación con defecto que pudiera encontrar el caso de prueba.

Los datos que se registran en la etapa de especificación de una prueba de sistema son:

- Tipo de prueba.
- ID de prueba.
- Elementos de proyecto asociados.
- Parte del sistema a probar.
- Nombre de prueba.
- Propósito.
- Precondiciones.
- Pasos/acciones.
- Salida esperada.
- Salida obtenida.
- Poscondiciones.

Estos datos son los que fueron considerados de acuerdo al modelo de planilla utilizado y adaptado para los casos de prueba de sistema, tal como se detallan en la siguiente subsección.

Modelo de planilla utilizada para pruebas de sistema

Para las pruebas de sistema se utilizó la siguiente planilla de información asociada a cada caso de prueba.

Caso de prueba	Tipo de prueba	ID de prueba
Prueba sobre: Parte de sistema a probar		Elementos de proyecto asociados
Nombre de prueba		
Propósito		
Pre-condiciones		
Pasos/acciones		
Salida esperada		
Salida obtenida		
Pos-condiciones		
Nº de iteración/fase		
IDs de defectos detectados		
Resultado de CP (Paso/Fallo):		

Tabla 5.4 Modelo de plantilla para pruebas de sistema.

Esta planilla registra toda la información de un caso de prueba de sistema que consideramos importante de acuerdo al sistema desarrollado. La información registrada se centra en las etapas de especificación del caso de prueba, ejecución del caso de prueba y por ultimo gestión de defectos si se desprende uno o varios defectos para la prueba en cuestión.

A continuación se detalla cada uno de los campos considerados para un caso de prueba de sistema.

- **Tipos de pruebas:** Pruebas Unitarias, Pruebas Integrales/componentes, Pruebas de sistema, etc.
- **ID de prueba:** Identificador que dependerá del tipo de prueba y la cantidad de pruebas.
Ejemplo: CPU1, CPI1, CPS1
- **Elementos de proyecto asociados:** lista de todos los elementos que cubre la prueba por ejemplo requerimientos, casos de uso, riesgo, o cualquier otro elemento del proyecto que se asocie a la prueba.
- **Parte del sistema a probar:** Si se trata de una prueba unitaria se deberá indicar el método o función a probar y el sistema asociado. Si se trata de una prueba integral se deberá indicar la interfaz o comunicación de funciones o métodos a probar y el sistema asociado en el que se prueba. Y por último si se trata de una prueba de sistema se indicará la parte del sistema de software que se está comprobando o si se trata del sistema completo.
- **Nombre de prueba:** título representativo de la prueba.
- **Propósito:** Objetivo que se pretender alcanzar la prueba.
- **Precondiciones:** condiciones necesarias antes de realizar la prueba en caso de ser necesario.
- **Pasos/acciones:** Listados de acciones necesarias para llevar a cabo la prueba.
- **Salida esperada:** salida que se espera que devuelva de manera ideal el sistema, o parte de este y/o las herramientas de prueba, al ejecutar el caso de prueba.
- **Salida obtenida:** salida real obtenida al ejecutar el caso de prueba según las acciones establecidas para la prueba en cuestión. Esta salida se compara con la salida esperada para determinar si todo responde como se desea
- **Poscondiciones:** condiciones necesarias luego de haber realizado la prueba para dejar el sistema o parte de este tal como estaba.
- **Nº de iteración/fase:** número de veces que se realiza la prueba hasta obtener el resultado PASO.
- **ID de defectos detectados:** Identificador de un nuevo defecto detectado en el sistema para localizarlo en la tabla de defectos.
- **Resultado de prueba:** Se indica si la prueba paso correctamente o fallo, en caso de detectarse un defecto en el sistema.

5.4.2. EJECUCIÓN DEL CASO DE PRUEBA

En esta etapa los casos de prueba unitarios e integrales se ejecutan de manera automatizada y se determinan si existen o no defectos asociados, en caso de existir se analiza el resultado y se ejecuta hasta eliminar el defecto, para estos casos de prueba no se registra información de ninguna otra etapa, al momento de obtener el resultado correcto finaliza su ciclo de vida.

Para los casos de prueba de sistema estos se ejecutan siguiendo cada uno de los pasos propuestos por las tablas de la especificación del caso. Al finalizar la ejecución se analizan los resultados y se registran como se describe en la siguiente etapa.

5.4.3. ANÁLISIS Y REGISTRO DE RESULTADOS

Como se mencionó en la etapa anterior, los únicos casos de prueba en los que se documenta los resultados de la ejecución de la prueba son solo en los casos de prueba de sistema sobre las mismas tablas donde se encuentra

la especificación del caso. Se realizó de esta manera para que quede la mayor información posible del proceso de prueba asociado al caso en una única tabla. Además en caso de detectar defectos sobre cualquiera de los casos ejecutados se genera un nuevo defecto y se lo vincula al caso de prueba que lo encontró. En este caso el proceso de la prueba continúa en la última etapa de gestión y reporte de defecto y finaliza el ciclo de la prueba cuando no se detecten más defectos.

Los datos que se registran de la etapa de ejecución son:

- Salida esperada.
- Salida obtenida.
- N° de iteración/fase.
- ID de defectos detectados.
- Resultado de prueba.

5.4.4. GESTIÓN Y REPORTE DE DEFECTOS

En esta última etapa se realizó un control, revisión y solución a los defectos encontrados por todos los casos de prueba. Recordamos que solo se documentaron los defectos de los casos de prueba de sistema ya que los casos de pruebas unitarias e integrales fueron automatizados y se decidió no realizar una gestión de sus defectos de manera detallada.

Para los casos de prueba de sistema se utilizó una planilla de defectos que registra la información que se consideró importante para su gestión. En la siguiente subsección se muestra la planilla utilizada para la gestión de los defectos.

Modelo de planilla utilizada para gestión de defectos

La planilla utilizada para la gestión de defectos es la siguiente:

Defecto detectado		
ID de defecto	ID de prueba asociada	Nº de iteración/fase
Descripción:		
Descripciones y comentarios asociados al defeco.		
Principal causa del problema.		
Resumen de solución.		

Tabla 5.5 Modelo plantilla para gestión de defectos.

Se puede observar que la tabla es sencilla, brindando información del caso de prueba asociado, y el número de iteración sobre el que se encontró el defecto. Por último se describe las características del origen del defecto y cuál es la posible solución al mismo para resolverlo.

5.5. COBERTURA DE PRUEBAS

En esta sección se analiza la cobertura de pruebas obtenida como resultado de todos los conjuntos de pruebas analizados anteriormente sobre cada una de las características propuestas para el sistema. Para esto, se decidió

analizar la trazabilidad de los bancos y casos de prueba con los diferentes elementos del proyecto considerados importantes para el análisis de esta etapa.

5.5.1. TRAZABILIDAD DE PRUEBAS

Al igual que la trazabilidad de requerimientos del sistema desarrollado, analizada en la sección 3.3 del presente proyecto, se desea hacer lo mismo pero en este caso con los bancos y casos de prueba de todos los niveles. Obviamente que las trazabilidades cubiertas en la sección 3.3 no se volverán a ver aquí, como por ejemplo la trazabilidad de los requerimientos con los casos de prueba, pero si se verán un conjunto de trazabilidades que resultan interesantes analizar con respecto a diferentes elementos del presente plan de pruebas.

En este caso para el análisis de la trazabilidad de bancos y casos de prueba son de interés las siguientes correlaciones:

- **BPU/BPI – Características de prueba:** Esta correlación permitirá visualizar cada “característica de prueba” por qué bancos de pruebas unitarias (BPU) e integrales (BPI) fue cubierto.
- **CPS – Características de prueba:** Esta correlación permitirá visualizar cada “característica de prueba” por qué casos de pruebas de sistema fue cubierto.

Correlación BPU/BPI vs Características de prueba

A continuación se proporciona la matriz de trazabilidad de la correlación BPU/BPI versus características de prueba.

		Característica de Prueba					
		RF	RFN	FMEM	MT	REND	IEF
BPU/BPI	BPU1	✓					
	BPU2	✓					
	BPU3	✓					
	BPU4	✓					
	BPU5	✓					
	BPU6	✓					
	BPI1	✓	✓		✓		
	BPI2	✓	✓		✓		
	BPI3	✓					

Correlación CPS vs Características de prueba

A continuación se proporciona la matriz de trazabilidad de la correlación casos de pruebas de sistema versus características de prueba.

	Característica de Prueba					
	RF	RFN	FMEM	MT	REND	IEF
CPS1		✓	✓			
CPS2		✓	✓			
CPS3		✓	✓			
CPS4		✓	✓			
CPS5		✓	✓			
CPS6		✓	✓			
CPS50	✓	✓				✓
CPS51	✓	✓				✓
CPS52	✓	✓				✓
CPS53	✓	✓				✓
CPS54	✓	✓				✓
CPS55	✓	✓				✓
CPS56	✓	✓				✓
CPS57	✓	✓				✓
CPS58	✓	✓				✓
CPS59	✓	✓				✓
CPS101		✓		✓		
CPS201		✓			✓	
CPS202		✓			✓	
CPS203		✓			✓	
CPS204		✓			✓	
CPS205		✓			✓	
CPS206		✓			✓	

6. RESULTADOS

Como parte de los resultados del proyecto, para el kernel de Linux se obtuvo una librería que permite el control y la gestión de las RDPG de manera modular, <RDPG_object.h>. Desde el principio se trabajó con esta idea haciendo que toda la lógica utilizada por el driver MatrixmodG pueda ser utilizada en cualquier otro driver que requiera RDPG. Por otro lado, para el espacio usuario, se trabajó con la idea de lograr utilizar el driver desde códigos de este espacio, obteniendo como resultado la librería <libMatrixmodG.h>.

De manera resumida todo el trabajo realizado funciona como se describe en las figuras 6.1 y 6.2.

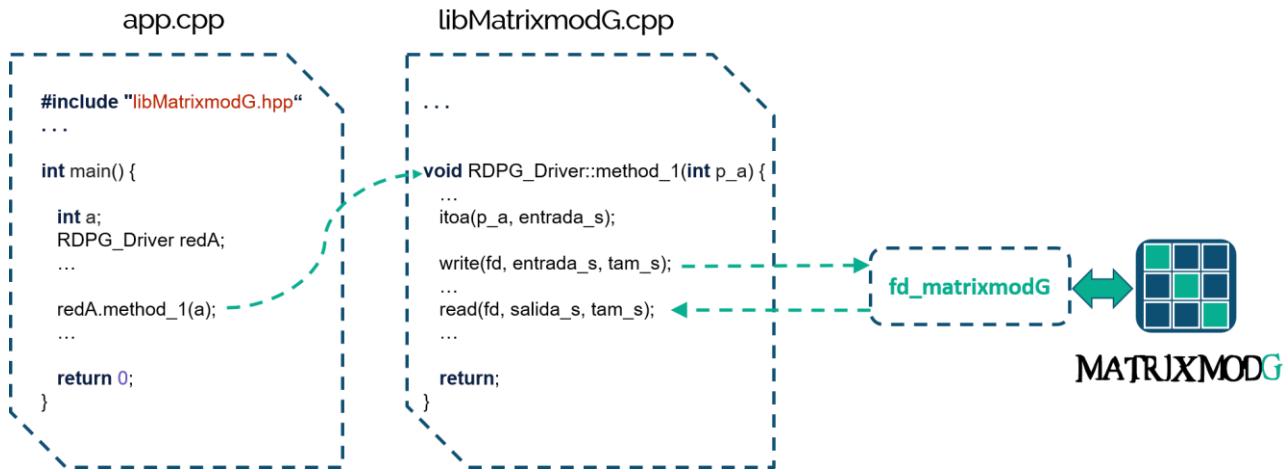


Figura 6.1 Funciones desde el espacio usuario.

La figura 6.1 muestra como un código de espacio usuario puede hacer uso del driver mediante la inclusión de la librería <libMatrixmodG.h>, la misma procesa los datos para convertirlos a cadenas de carácter y armar los comandos interpretados por el driver, que se envían a través de llamadas al sistema logrando la comunicación con el driver MatrixmodG desde su file device (o archivo de dispositivo) asociado, denominado fd_matrixmodG.

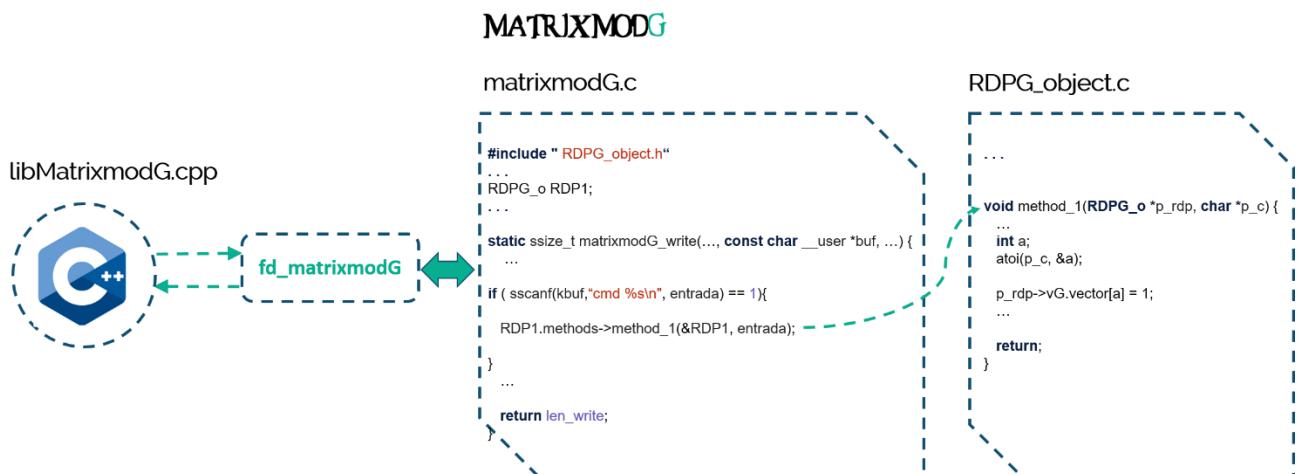


Figura 6.2 Funciones desde el espacio kernel.

La figura 6.2 muestra como desde el lado del kernel se reciben todas las peticiones del espacio usuario en cadenas de carácter, que son los comandos interpretados por el driver. Ante operaciones de escritura, el driver MatrixmodG siempre se encuentra interpretando los comandos y procesando las cadenas para extraer los datos adecuados de cada comando. De esta forma se puede realizar la operación asociada a la solicitud de un usuario, también para el caso de las lecturas al driver se convierten todos los datos del kernel a cadenas de carácter para enviarlos adecuadamente al espacio usuario.

A continuación se explican los detalles de cada una de las diferentes librerías mencionadas que participan del proceso de la gestión y control de RDPG en el kernel de Linux desde el espacio usuario.

6.1. MODO DE USO DE OBJETOS EN DRIVER MATRIXMOD

6.1.1. CONSTRUCCIÓN Y CREACIÓN DE OBJETOS

Construcción de objetos

De acuerdo con el paradigma de programación orientado a objetos en el kernel, previo a la creación de cada uno de los objetos en el kernel estos deben construirse. Esta etapa se realiza con el método constructor de cada objeto. En el diseño del driver se normalizo el uso del nombre new_obj para cada constructor de los objetos que componen el driver. Por ejemplo para la construcción de una nueva RDPG, una matriz y un vector en el driver se usa de la siguiente forma:

```
/* Declaración de objetos RDPG_o, matrix_o y vector_o. */
RDPG_o RDPG1;
matrix_o matrix1;
vector_o vector1;

/* Construcción de nuevo objeto RDPG_o con nombre RDPG1. */
new_RDPG(&RDPG1, "RDPG1");

/* Construcción de nuevo objeto matrix_o con nombre Matriz de prueba. */
new_matrix(&matrix1, "Matriz de prueba");

/* Construcción de nuevo objeto vector_o con nombre Vector de prueba. */
new_vector(&vector1, "Vector de prueba");
```

El constructor de los objetos se encarga de inicializar las variables y configurar los valores por defecto sobre los atributos de un objeto. De esta forma el objeto del kernel ya puede llamar a cualquier de sus métodos propios del objeto ya que la primer inicialización se encarga de hacer conocer al objeto cuales son todos sus métodos, es decir cada objeto conoce su propia vtable de métodos disponibles.

Creación de objetos

Luego de la etapa de construcción de un objeto en el kernel, sigue la etapa de creación del objeto. Se aprovecha esta etapa para inicializar los valores por parámetros, configurar nombres y asignar memoria dinámicamente según los requerimientos de cada objeto en particular. Esto da gran flexibilidad y eficiencia en la gestión de memoria ya que no es lo mismo una RDPG de 1000 plazas y 1000 transiciones que una RDPG de 10 plazas y 10 transiciones. Por ejemplo para la creación de una RDPG se usa:

```
/* Creación del objeto RDPG_o RDPG1, con 10 plazas y 5 transiciones. */
RDPG1.methods->create_rdp(&RDPG1, "10_5");
```

Todos los parámetros se envían como parámetros de tipo carácter ya que el driver recibe todo por medio de caracteres. De esta forma los métodos de cada objeto RDPG deben filtrar los caracteres para la gestión de errores y para la extracción de los datos como enteros. Si las entradas de carácter pasan adecuadamente los filtros de control y detección de errores que tiene el objeto RDPG, entonces los datos se pudieron extraer correctamente para la creación de cada uno de los componentes de una RDPG. Cada componente que conforma una RDPG es una matriz o un vector, a los cuales se le debe asignar memoria dinámicamente (en tiempo de ejecución) para almacenar sus datos. Como se vio en el marco teórico para la reservar memoria en el kernel existen diferentes interfaces, de acuerdo con esto, en el driver se diseñó un conjunto de alternativas para la asignación de memoria

en cada componente de una RDPG, esto permitirá estudiar cuál de los tipos de asignadores utilizados tiene un mejor rendimiento. El driver permite asignar memoria de acuerdo con la tabla 6.1:

Tabla 6.1	
Modo de asignación	Descripción
M_KMALLOC (Id = 1)	Asigna memoria en regiones físicamente contiguas. Requiere de inicialización de elementos. Puede asignar más memoria de la solicitada para satisfacer la región contigua.
M_KZALLOC (Id = 2)	Asigna memoria en regiones físicamente contiguas. Inicializa todos los elementos a cero. Puede asignar más memoria de la solicitada para satisfacer la región contigua.
M_KCALLOC (Id = 3)	Asigna memoria en regiones físicamente contiguas. Inicializa todos los elementos a cero. Optimización para la gestión de vectores o arreglos. Puede asignar más memoria de la solicitada para satisfacer la región contigua.
M_VMALLOC (Id = 4)	Asigna memoria en regiones virtualmente contigua, puede ser no contigua físicamente. Requiere de inicialización de elementos. Asigna la memoria solicitada. Soporta grandes asignaciones de memoria.
M_VZALLOC (Id = 5)	Asigna memoria en regiones virtualmente contigua, puede ser no contigua físicamente. Inicializa todos los elementos a cero. Asigna la memoria solicitada. Soporta grandes asignaciones de memoria.

De esta forma para seleccionar un modo de asignación de memoria se usa:

```
/* Configuración de un modo de asignación de memoria dinámica. */
RDPG1.methods->set_allocMemMode(&RDPG1, "3");
```

El parámetro “3” indica el modo M_KCALLOC para la asignación de memoria en cada componente de la RDPG.

6.1.2. CREACIÓN DE OBJETOS RDPG

Los pasos que se requieren para crear de manera adecuada una RDPG en el kernel de Linux son los siguientes.

Inclusión de librería y declaración de objetos RDPG

```
/* Inclusión de librería de objeto RDPG_o. */
#include "RDPG_object.h"

/* Declaración de objeto RDPG_o. */
RDPG_o RDPG1;
```

Lo primero será la inclusión de la librería necesaria para la creación de objetos del tipo RDPG_o. Como se vio anteriormente luego de la inclusión se puede declarar objetos RDPG para que este sea inicializado posteriormente mediante el uso de su constructor.

Construcción del objeto RDPG

```
/* Construcción de nuevo objeto RDPG_o con nombre RDPG1. */
new_RDPG(&RDPG1, "RDPG1");
```

La construcción de un objeto RDPG se encarga de establecer los valores por defecto de cada uno de los atributos de una RDPG y permite conocer los métodos que tiene disponible el objeto. Por lo general el constructor se utiliza en la función init de un driver (module_init) para el caso del driver MatrixmodG se utilizó esta idea. Se puede establecer cualquier nombre a la RDPG que se adapte al tipo de problema a resolver, el límite es que no supere los 128 caracteres.

Creación de un objeto RDPG

```
/* Construcción de nuevo objeto RDPG_o con nombre RDPG1. */
RDPG1.methods->create_rdpg(&RDPG1, "10_6");
```

Una vez inicializado un objeto RDPG en el kernel, este ya puede ser creado con el uso del método create_rdpg(), como se muestra en el ejemplo. Se debe enviar el número de plazas y transiciones de la RDPG en una cadena de caracteres, esto se realizó así ya que los device driver reciben toda la información del usuario como cadena de caracteres, la librería cuenta con los módulos de detección de errores y extracción de datos para trabajar adecuadamente en el kernel y en caso de error se notificara el mismo por la salida de diagnóstico del kernel.

En este momento el kernel ya tiene creado todos los componentes necesarios de la RDPG pero no se conoce su estructura. La estructura se define cuando se establecen los componentes bases de la RDPG, para establecer los componentes base se debe conocer cada uno de los elementos de las matrices de incidencia y el marcado inicial. En resumen los componentes bases son:

- Matriz de incidencia I.
- Matriz de incidencia H de brazos inhibidores.
- Matriz de incidencia R de brazos lectores.
- Matriz de incidencia Re de brazos reset.
- Vector de marcado inicial.

Para todos los componentes base, un objeto RDPG cuenta con una función que le permite ingresar cada uno de sus elementos de la siguiente forma.

```
/* Ingreso de un elemento del componente mII de La RDPG. */
RDPG1.methods->add_value_in_mcomponent(&(RDPG1.mII), "0_0_-1");
```

El ejemplo anterior muestra cómo se ingresa un elemento a la matriz de incidencia I (componente mII dentro de un objeto RDPG_o). La cadena de caracteres al igual que la función create_rdpg() recibe los datos que se desean establecer con un formato de espacios dados por el símbolo “_” y con soporte al control de errores y extracción de datos, si todo es correcto el primer valor se interpreta como la posición 0 (cero) dentro de las filas de la matriz, el segundo valor se interpreta como la posición 0 (cero) de las columnas de la matriz y el tercer valor se interpreta como el valor -1 a asignar sobre las respectivas posiciones de fila y columna de la matriz. En relación con las RDPG en otras palabras se está indicando que la plaza 0 (cero) tiene un arco hacia la transición 0 (cero) de la red ya que el valor es -1, si el valor fuera 1 indicaría que la transición 0 (cero) tiene un arco hacia la plaza 0 (cero).

Para el caso del resto de las matrices de incidencia la lógica es similar, solo que se reciben 0 (ceros) o 1 (unos) ya que todos los arcos siempre salen de una plaza hacia una transición, en donde el cero indica que no existe relación entre una plaza y una transición al igual que en el componente mII, pero un uno indica que existe un arco de la plaza i hacia la transición j. Por ejemplo:

```
/* Ingreso de un elemento del componente mIH de La RDPG. */
RDPG1.methods->add_value_in_mcomponent(&(RDPG1.mIH), "3_7_1");
```

```
/* Ingreso de un elemento del componente mIR de La RDPG. */
RDPG1.methods->add_value_in_mcomponent(&(RDPG1.mIR), "1_4_1");
```

```
/* Ingreso de un elemento del componente mIRe de La RDPG. */
RDPG1.methods->add_value_in_mcomponent(&(RDPG1.mIRe), "14_13_1");
```

Por último se deben establecer los elementos del vector de marcado inicial de la RDPG, esto se hace mediante el uso de la siguiente función.

```
/* Ingreso de un elemento del componente vMI de La RDPG. */
RDPG1.methods->add_value_in_vcomponent(&(RDPG1.vMI), "0_3");
```

Entonces para cada valor de marcado inicial que se tenga se deberá llamar este método del objeto RDPG que se encarga de dicha operación. En este componente al tratarse de un vector solo se debe indicar en la cadena de carácter la posición del vector en el cual establecer el valor, en el ejemplo la posición es cero y el valor 3, en relación a las RDPG lo que esto indica es que la plaza cero cuenta con una cantidad de 3 tokens inicialmente.

En el driver MatrixmodG todos estos métodos se llaman sobre un objeto RDPG cuando se recibe la información desde el espacio usuario a través de los comandos que el driver provee.

Con la definición de la estructura de la RDPG mediante la configuración de los componentes base, la RDPG ya tiene estructura, para ser gestionada solo falta un paso, el cual consiste en que se establezcan los valores del resto de componentes internos que maneja la RDPG, para esto el objeto RDPG cuenta con el siguiente método:

```
/* Actualización de componentes de trabajo de RDPG. */
RDPG1.methods->update_work_components(RDPG1);
```

Esta función se encarga de establecer los valores de los componentes internos del objeto RDPG, estos componentes son:

- Vector E.
- Vector Q.
- Vector W.
- Vector B.
- Vector L.
- Vector A.
- Vector G.
- Vector UDT.
- Vector Ex.
- Vector HQCV.
- Vector HD.

Luego de todos estos pasos, el objeto RDPG ya está listo para ser gestionado y controlado desde el kernel de Linux.

6.1.3. OBTENER INFORMACIÓN DE OBJETOS RDPG

Para obtener información de un objeto RDPG del kernel, los objetos cuentan con dos métodos para esto, los métodos se diferencian de acuerdo a qué tipo de información es la que se desea obtener.

Obtener estado de cualquiera de los componentes de la RDPG

Un objeto RDPG permite obtener el estado de cualquiera de los componentes de la RDPG creada. Para esto el método se presenta en el siguiente ejemplo:

```
/* Obtención de estado de un componente de La RDPG. */
RDPG1.methods->read_rdpq_component(RDPG1, cadena_usr, leng);
```

```
/* Cambio de componente de La RDPG. */
RDPG1.methods->set_select_comp(RDPG1, mIH);
RDPG1.methods->read_rdpg_component(RDPG1, cadena_usr, leng);
```

Por defecto una RDPG muestra el estado del componente mII, pero este se puede modificar como muestra el ejemplo con el uso de la función set_select_comp() que mediante el uso de macros de una enumeración permite cambiar de componentes de acuerdo a las necesidades, en este caso se cambia a mIH que será la matriz que se mostrara en la próxima ejecución de la función read_rdpg_component(), en el segundo parámetro se pasa una cadena de caracteres que es en donde se almacena la información que se envía al espacio usuario, el tercer parámetro es el tamaño de la cadena de espacio usuario.

Obtener información de la RDPG

Si se desea obtener información acerca de las características de un objeto RDPG, se provee el método read_rdpg_info(), como muestra el siguiente ejemplo:

```
/* Obtención de información de La RDPG. */
RDPG1.methods->read_rdpg_info(&RDPG1, cadena_usr, leng);
```

Este método trabaja en conjunto con un atributo denominado read_mode, que de acuerdo a su valor se muestra información diferente relacionada a la RDPG. Los valores que se pueden asignar a read_mode se definieron en una enumeración para fijar el tipo de información que se desea obtener. La enumeración es la siguiente:

```
/* Enumeracion de Los tipos de información que se puede obtener en La RDPG. */
enum{
    ID_MCOMPONENT_MODE = 0,      /**< Modo con el cual el drive escribe al espacio usuario un componente matriz
(matrix_o) de La RDPG_o.*/
    ID_VCOMPONENT_MODE,          /**< Modo con el cual el drive escribe al espacio usuario un componente vector
(vector_o) de La RDPG_o.*/
    ID_INFO_MODE,                /**< Este es el modo de lectura de información. A partir de este id termina el
modo de lectura de componentes.*/
    ID_INFO_NAME,                /**< Modo con el cual el drive escribe al espacio usuario el nombre configurado
de La RDPG_o.*/
    ID_INFO_PLACES,              /**< Modo con el cual el drive escribe al espacio usuario el número de plazas
configurado en La RDPG_o.*/
    ID_INFO_TRANSITIONS,         /**< Modo con el cual el drive escribe al espacio usuario el número de
transiciones en La RDPG_o.*/
    ID_INFO_SHOTS,               /**< Modo con el cual el drive escribe al espacio usuario el número de disparos
en La RDPG_o.*/
    ID_INFO_MEMORY,               /**< Modo con el cual el drive escribe al espacio usuario la memoria reservada
en La RDPG_o.*/
    ID_INFO_COMP,                 /**< Modo con el cual el drive escribe al espacio usuario datos de submode de
componente en La RDPG_o (ver ID_READ_SUBMODE).*/
    ID_INFO_SHOT_RESULT,          /**< Modo con el cual el drive escribe al espacio usuario el resultado del
ultimo disparo realizado en La RDPG_o.*/
    ID_ERROR_MODE,                /**< Modo con el cual el drive escribe al espacio usuario una cadena de error de
lectura en La RDPG_o.*/
    ID_RM_END                    /**< Número que indica el fin de modos para La enumeracion ID_READ_MODE.*/
}ID_READ_MODE;
```

El método necesario para cambiar el atributo read_mode es set_read_mode(), como muestra el siguiente ejemplo:

```
/* Cambio de modo de Lectura de La RDPG para mostrar su nombre. */
RDPG1.methods->set_read_mode(&RDPG1, ID_INFO_NAME);

/* Cambio de modo de Lectura de La RDPG para mostrar la memoria utilizada. */
RDPG1.methods->set_read_mode(&RDPG1, ID_INFO_MEMORY);

/* Cambio de modo de Lectura de La RDPG para mostrar información del componente seleccionado. */
RDPG1.methods->set_read_mode(&RDPG1, ID_INFO_COMP);
```

6.1.4. DISPARO DE TRANSICIÓN DE OBJETOS RDPG

Teniendo la estructura de una RDPG y su marcado inicial ya se puede comenzar a disparar cualquiera de las transiciones de la RDPG. Los objetos RDPG cuentan con un método para realizar el disparo de una transición, este método es muy útil ya que es el que permite la ejecución de la RDPG de acuerdo a los disparos que se pueden realizar. El siguiente ejemplo muestra el método de disparo:

```
/* Disparo de una transición de La RDPG. */
RDPG1.methods->disparar_rdp桔s(&RDPG1, "2", SHOT_MODE_E);
```

El ejemplo muestra el método para realizar un disparo el cual recibe como parámetros la RDPG sobre la cual realizar el disparo, el segundo parámetro es el número de transición a disparar y el último parámetro es el modo en el que se desea realizar el disparo. Este último parámetro es muy útil ya que permite cambiar el estado actual de la RDPG si el disparo es exitoso (usando macro SHOT_MODE_E) o simplemente conocer el próximo estado de la RDPG pero no actualizar el estado actual de la RDPG.

6.1.5. SETEO DE PARÁMETROS EN OBJETOS RDPG

La forma en que se trabaja con los objetos RDPG en el kernel de Linux puede configurarse de acuerdo a las necesidades que se adapten para cada caso particular. La librería RDPG_object creada para la gestión de las RDPG en el kernel de Linux proporciona un conjunto de métodos que le permiten al objeto RDPG cambiar su forma de trabajar en el kernel.

Estos métodos permiten el cambio de un conjunto de parámetros que utilizan los objetos RDPG para dicha finalidad. Estos parámetros son cuatro atributos del objeto que se detallan a continuación:

- **s_allocMode:** Cadena de carácter con el nombre del modo de asignación de memoria configurado para la creación de todos los componentes de la RDPG. Su cambio solo admite los modos de memoria validos enumerados en la sub-sección 6.1.1.
- **posVP:** Variable que contiene el número de una plaza de la RDPG. Al consultar el estado de cualquiera de los componentes de la red se obtendrá el mismo iniciando desde la plaza con el número configurado en posVP. Su cambio tiene el límite mínimo de cero y máximo de acuerdo a la cantidad de plazas que tiene la RDPG creada en el kernel.
- **posVT:** Variable que contiene el número de una transición de la RDPG. Al consultar el estado de cualquiera de los componentes de la red se obtendrá iniciando desde la transición con el número configurado en posVT. Su cambio tiene el límite mínimo de cero y máximo de acuerdo a la cantidad de transiciones que tiene la RDPG creada en el kernel.
- **vdim:** Variable que contiene el número de plazas y transiciones que se muestran como límite al consultar el estado de cualquiera de los componentes de la RDPG. Su cambio tiene el límite mínimo de 10 o el número mínimo de plazas o transiciones existentes y el límite máximo fijo de 30 elementos (plazas o transiciones).

Los métodos para el cambio de los parámetros de un objeto RDPG son los que muestra el siguiente ejemplo.

```
/* Cambio del modo de asignación de memoria de La RDPG. */
RDPG1.methods->set_MemAllocMoode(&RDPG1, "2");

/* Cambio del número de plaza inicial (o posición inicial de plaza) de La RDPG. */
RDPG1.methods->set_posVP(&RDPG1, "10");

/* Cambio del número de transición inicial (o posición inicial de transición) de La RDPG. */
RDPG1.methods->set_posVT(&RDPG1, "5");

/* Cambio de número de elementos a mostrar en La consulta del estado de componentes de La RDPG*/
RDPG1.methods->set_vdim(&RDPG1, "30");
```

6.1.6. PROTECCIÓN SMPs DE OBJETOS RDPG

Para cada operación que se realiza sobre un objeto RDPG_o se debe tener especial cuidado a las solicitudes concurrentes ya sean de lectura o escritura sobre una misma RDPG. Para esto se diseñó un conjunto de métodos que protege a los objetos RDPG frente a los accesos concurrente brindando el soporte necesario de protección de recursos para su adecuado funcionamiento.

Los métodos necesarios son exactamente los mismos que se mostraron en las subsecciones anteriores, solo que en este caso se utiliza una variable de protección de tipo spinlock encargada de proteger el recurso. Su funcionamiento es igual al de una puerta con llave, si viene una persona y la puerta está abierta, la persona ingresa y cierra la puerta con llave para que no ingrese nadie más hasta realizar su operación, al finalizar la persona abre la puerta y sale dejándola nuevamente abierta para el resto de las personas. En el kernel si suponemos que las personas son hilos y la puerta con llave es la variable de tipo spinlock, entonces esta es la forma de protección del recurso.

En un objeto RDPG_o el mayor inconveniente se da sobre las operaciones de escritura, es decir cuando se actualiza el estado actual de la RDPG, porque no se puede leer la RDPG mientras se está actualizando su estado actual, es entonces donde el spinlock funciona perfectamente para esta situación. Pero se penaliza a las operaciones de lectura concurrentes ya que si un hilo lee la RDPG para conocer su estado y otro también lo quiere hacer al mismo tiempo, el segundo deberá esperar la salida del primero, para evitar esto y proporcionar mayor eficiencia en este tipo de operaciones se utilizó un tipo de spinlock especial denominado rwlock (spinlock de lectura y escritura) que es la variable especial que protege al recurso y provee eficiencia para las operaciones de lectura.

De esta forma es como un objeto RDPG del kernel puede trabajar tranquilamente con soporte al multiprocesamiento simétrico seguro con el uso de las funciones SMPs. Ahora un objeto RDPG para realizar cualquier operación debe llamar a su función correspondiente con protección al multiprocesamiento utilizando el siguiente conjunto de métodos:

```
/* Ingreso de un elemento del componente mII de La RDPG. */
RDPG1.methods->SMPs_add_value_in_mcomponent(&(RDPG1.mII), "0_0_-1");

/* Ingreso de un elemento del componente mIH de La RDPG. */
RDPG1.methods->SMPs_add_value_in_mcomponent(&(RDPG1.mIH), "3_7_1");

/* Ingreso de un elemento del componente mIR de La RDPG. */
RDPG1.methods->SMPs_add_value_in_mcomponent(&(RDPG1.mIR), "1_4_1");

/* Ingreso de un elemento del componente mIRe de La RDPG. */
RDPG1.methods->add_value_in_mcomponent(&(RDPG1.mIRe), "14_13_1");

/* Ingreso de un elemento del componente vMI de La RDPG. */
RDPG1.methods->SMPs_add_value_in_vcomponent(&(RDPG1.vMI), "0_3");

/* Actualización de componentes de trabajo de RDPG. */
RDPG1.methods->SMPs_update_work_components(RDPG1);

/* Disparo de una transición de La RDPG. */
RDPG1.methods->SMPs_disparar_rdpg_s(&RDPG1, "2", SHOT_MODE_E);

/* Cambio de componente de La RDPG. */
RDPG1.methods->SMPs_set_select_comp(RDPG1, vEx);

/* Obtención de estado de un componente de La RDPG. */
RDPG1.methods->SMPs_read_rdpg_component(RDPG1, cadena_usr, leng);

/* Cambio de modo de Lectura de La RDPG para mostrar La memoria utilizada. */
RDPG1.methods->SMPs_set_read_mode(&RDPG1, ID_INFO_memory);
```

```

/* Obtención de información de La RDPG. */
RDPG1.methods->SMPs_read_rdp_info(&RDPG1, cadena_usr, leng);

/* Cambio del modo de asignación de memoria de La RDPG. */
RDPG1.methods->SMPs_set_MemAllocMode(&RDPG1, "2");

/* Cambio del número de plaza inicial (o posición inicial de plaza) de La RDPG. */
RDPG1.methods->SMPs_set_posVP(&RDPG1, "10");

/* Cambio del número de transición inicial (o posición inicial de transición) de La RDPG. */
RDPG1.methods->SMPs_set_posVT(&RDPG1, "5");

/* Cambio de número de elementos a mostrar en la consulta del estado de componentes de La RDPG*/
RDPG1.methods->SMPs_set_vdim(&RDPG1, "30");

```

Se puede notar que todas las funciones son iguales pero ahora inician su nombre con “SMPs_” lo cual indica que la función se ejecutará con protección del recurso RDPG para gestionar múltiples hilos simultáneos adecuadamente.

La implementación de este tipo de métodos se realizó de una manera muy sencilla al trabajar con el paradigma POO, ya que antes de operar en el objeto se lo protege mediante un spinlock. A continuación un ejemplo para la función de disparo:

```

int SMPs_disparar_rdp(RDPG_o *p_rdp, int p_idT, SHOT_MODE p_mode)
{
    int rt_fshoot; /* Retorno de función shoot_rdp() */

    write_lock(&p_rdp->lock_RDPG); /* Protejo recurso. */

    rt_fshoot= p_rdp->methods->shoot_rdp(p_rdp, p_idT, p_mode);

    write_unlock(&p_rdp->lock_RDPG); /* Libero recurso. */

    return rt_fshoot;
}

```

Para todas las funciones se utilizó análogamente esta misma implementación, la cual provee la protección necesaria frente al multiprocesamiento simétrico, convirtiéndolo en multiprocesamiento simétrico seguro.

6.2. ESCRITURA Y LECTURA DE DRIVER MATRIXMODG

El driver MatrixmodG responde a las escrituras y lecturas que se realizan desde el espacio usuario sobre el archivo de dispositivo (o file device) asociado.

Una forma de realizar la escritura al file device del driver es mediante la redirección de cadenas de caracteres al file device desde una terminal de Linux, por ejemplo:

```

$ echo RDPG config alloc_mode 3 > /proc/matrixmodG_fd
$ echo create RDPG 6_4 > /proc/matrixmodG_fd

```

Los comandos de consola del ejemplo anterior, son una de las formas de escribir sobre el file device del driver cuyo nombre es “matrixmodG_fd”, ubicado sobre el sistema de archivos /proc. El primer comando del ejemplo realiza el cambio del modo en que el driver MatrixmodG reservara memoria para los objetos gestionados en el kernel. Por otro lado el segundo comando del ejemplo solicita al driver MatrixmodG que realice la creación de una RDPG en el kernel de 6 plazas con 4 transiciones.

La forma de realizar una lectura al file device del driver mediante el uso de una consola de Linux se realiza con el comando cat propio del sistema, por ejemplo:

```
$ cat /proc/matrixmodG_fd
```

```
T0 T1 T2 T3  
P0 0 0 0 0  
P1 0 0 0 0  
P2 0 0 0 0  
P3 0 0 0 0  
P4 0 0 0 0  
P5 0 0 0 0
```

La salida devuelta por el driver MatrixmodG al realizar una lectura va depender de la configuración realizada con los comandos de escritura previamente realizados, en el ejemplo anterior se muestra la matriz de incidencia de una RDPG donde no se cargaron sus valores asociados a la matriz de incidencia.

Si se desea obtener la información de la RDPG cargada en el kernel de Linux se debe solicitar una lectura de información mediante un comando de escritura, como se muestra en el siguiente ejemplo.

```
$ echo RDPGinfo memory > /proc/matrixmodG_fd  
$ cat /proc/matrixmodG_fd  
- Modo de reserva de memoria configurado en la RDPG: M_KMALLOC  
- Memoria reservada en el sistema: 792 bytes  
- Memoria real reservada en el sistema: 864 bytes
```

En este ejemplo se nota que se alteró la salida obtenida en la lectura del file device debido a la configuración previa realizada con el comando de escritura.

Esta es la forma de trabajar en conjunto con el módulo MatrixmodG y el espacio usuario en donde los comandos realizados de escritura y lectura a través de su file device son llamadas al sistema. El lenguaje de programación C contiene su propio conjunto de llamadas al sistema para trabajar con cualquier archivo. La librería de espacio usuario creada para el driver MatrixmodG en el lenguaje C hace uso de su conjunto de llamadas al sistema para realizar la manipulación del file device del driver y gestionar todas las escrituras y lecturas sobre el mismo, en la sección 4.4 se explica la composición de la librería de espacio usuario.

6.3. MODO DE USO DE LIBRERÍA DE ESPACIO USUARIO

Otro de los resultados obtenidos en el proyecto es la librería de espacio usuario que permite el uso del driver MatrixmodG desde este espacio de una manera simple y eficiente. La librería se programó en lenguaje C y C++ pero puede ser migrada a cualquier otro lenguaje de espacio usuario como JAVA, Python, Perl, etc. ya que se programó con el paradigma POO lo que la hace fácil de comprender.

6.3.1. INSTALACIÓN DE DRIVER MATRIXMODG

Para que la librería de espacio usuario del módulo MatrixmodG funcione adecuadamente en conjunto con el driver, primero será necesario que el driver MatrixmodG esté instalado en el kernel de Linux. Para esto se deberán seguir los siguientes pasos:

Ingresar como usuario sudo o administrador. Ingrese a donde se encuentra el código fuente del driver MatrixmodG, y ejecute el comando make, vea el siguiente ejemplo:

```
user:~$ sudo su  
password: ****  
root:~# cd matrixmodG  
root:~# make
```

Al finalizar la compilación del programa se genera un archivo con el nombre “matrixmodG.ko” este es el driver que se puede cargar en el kernel de Linux. Siguiendo en modo sudo, ejecute el siguiente comando:

```
root:~# insmod matrixmodG.ko
```

De esta forma el driver MatrixmodG ya está cargado en el kernel de Linux y es posible utilizar sin inconvenientes la librería de espacio usuario “libMatrixmodG”. Si desea desinstalar el driver MatrixmodG del kernel solo debe ejecutar el siguiente comando:

```
root:~# rmmod matrixmodG
```

6.3.2. GESTIÓN DE OBJETOS RDPG CON LIBRERÍA

Para gestionar objetos RDPG del kernel desde el espacio usuario en C/C++ primero se deberá incluir la librería de espacio usuario creada para la gestión del driver MatrixmodG. El siguiente ejemplo muestra cómo incluir la librería y el uso de su tipo de objeto (DriverRDPG_o en C y RDPG_Driver en C++) encargado de brindar todos los métodos necesarios para la gestión del driver MatrixmodG desde el espacio usuario:

```
/* LIBRERIA DE C. */
/* Inclusión de Librería de driver MatrixmodG. */
#include "libMatrixmodG.h"

/* Declaración de objeto DriverRDPG_o. */
DriverRDPG_o rdp;

/* Construcción de objeto DriverRDPG_o. */
new_MatrixmodGDriver(&rdp, "mII.txt", "mIH.txt", "mIR.txt", "mIRe.txt", "vMI.txt");

/* LIBRERIA DE C++. */
/* Inclusión de Librería de driver MatrixmodG. */
#include "libMatrixmodG.hpp"

/* Declaración de objeto RDPG_Driver. */
RDPG_Driver *rdp;

/* Construcción de objeto DriverRDPG_o. */
rdp = new RDPG_Driver("RDPG 1", "mII.txt", "mIH.txt", "mIR.txt", "mIRe.txt", "vMI.txt");
```

Como se observa en el ejemplo anterior el constructor del objeto encargado de peticionar al driver MatrixmodG recibe por parámetro cinco nombres de archivos de texto en donde se encuentran todas las matrices de incidencia de una RDPG y el vector de marcado inicial.

Para la creación de una nueva RDPG en el kernel desde el espacio usuario el constructor del objeto driverRDPG_o (o RDPG_Driver) realizan varios pasos en uno solo, estos son:

- Lectura de las matrices de incidencia y vector de marcado inicial desde archivos de extensión “.txt”.
- Conexión con driver MatrixmodG mediante su file device “/proc/matrixmodG_fd”.
- Creación de RDPG con número de plazas y transiciones de acuerdo a archivos de texto leídos.
- Envió de cada uno de los elementos de las matrices de incidencia y el vector de marcado inicial.
- Envió de comando de confirmación de la creación de la RDPG.

Si todo funciona exitosamente ya se puede realizar la gestión de la RDPG desde el espacio usuario. Es importante destacar que si una RDPG ya está creada en el kernel de Linux y solo se desea realizar la conexión desde el espacio usuario esto se logra con una variación en el uso del constructor, se deberá enviar NULL a todas las cadenas de carácter de los nombres de los archivos de texto que recibe como parámetros, si es en

lenguaje C, mientras que para C++ enviando string vacíos se obtiene el mismo resultado, tal como muestra el siguiente ejemplo:

```
/* LIBRERIA DE C. */
/* Uso de constructor para solo establecer conexión con driver MatrixmodG. */
new_MatrixmodGDriver(&rdp, NULL, NULL, NULL, NULL, NULL);

/* LIBRERIA DE C++. */
/* Uso de constructor para solo establecer conexión con driver MatrixmodG. */
string vacio;
rdp = new RDPG_Driver("RDPG 1", vacio, vacio, vacio, vacio, vacio);
```

Obtener estado de cualquiera de los componentes de la RDPG del kernel

Para obtener el estado de cualquiera de los componentes de la RDPG creada en el kernel. El objeto de espacio usuario proporciona un método que se presenta en el siguiente ejemplo:

```
/* LIBRERIA DE C. */
/* Obtención de estado de un componente de La RDPG. */
char cadena[USR_BUF_SIZE];
rdp.methods->view_compRDPG(&rdp, view_mII, cadena);

/* LIBRERIA DE C++. */
/* Obtención de estado de un componente de La RDPG. */
char cadena[USR_BUF_SIZE];
rdp->matrixmodG_view_compRDPG(view_mII, cadena);
```

Como se nota los parámetros enviados al método aparte del objeto driver son, el código identificador de componente que se desea obtener, en este caso se envía mediante el uso de una macro denominada “view_mII”, y en el último parámetro “cadena” se almacenan los datos retornados por el driver MatrixmodG asociados a la petición del estado del componente solicitado. Si se desea conocer el estado de todos los componentes se deberá realizar como muestra el siguiente ejemplo:

```
/* LIBRERIA DE C. */
/* Obtención del estado de todos Los componentes de La RDPG del kernel. */
rdp.methods->view_allCompRDPG(&rdp);

/* LIBRERIA DE C++. */
/* Obtención del estado de todos Los componentes de La RDPG del kernel. */
rdp->matrixmodG_view_allCompRDPG();
```

Obtener información de la RDPG del kernel

Para obtener la información acerca de las características de un objeto RDPG del kernel, el objeto de espacio usuario de la librería provee un método para dicha operación, el método se muestra en el siguiente ejemplo:

```
/* LIBRERIA DE C. */
/* Obtención de información de La RDPG del kernel. */
rdp.methods->view_RDPGinfo(&rdp);

/* LIBRERIA DE C++. */
/* Obtención de información de La RDPG del kernel. */
rdp->matrixmodG_view_RDPGinfo(&rdp);
```

Si se desea obtener solo la información de algún componente seleccionado en particular sobre la RDPG cargada en el kernel se utiliza el siguiente método:

```

/* LIBRERIA DE C. */
/* Obtención de información de un componente La RDPG del kernel. */
char cadena[USR_BUF_SIZE];
rdp.methods->view_compRDPG(&rdp, view_infoComponent, cadena);

/* LIBRERIA DE C++. */
/* Obtención de información de un componente La RDPG del kernel. */
char cadena[USR_BUF_SIZE];
rdp->matrixmodG_view_compRDPG(view_infoComponent, cadena);

```

Toda la información queda almacenada en la cadena de caracteres enviada en el último parámetro del método, en el ejemplo anterior es en “cadena”.

Ejecutar disparo de cualquier transición de la RDPG del kernel

Para realizar un disparo de cualquiera de las transiciones de la RDPG cargada en el kernel el objeto de la librería proporciona un método para dicha operación, tal como muestra el siguiente ejemplo:

```

/* LIBRERIA DE C. */
/* Ejecutar disparo de cualquier transición de La RDPG del kernel. */
rdp.methods->shoot_RDPG(&rdp, 3);

/* LIBRERIA DE C++. */
/* Ejecutar disparo de cualquier transición de La RDPG del kernel. */
rdp->matrixmodG_shoot_RDPG(3);

```

En el ejemplo se puede observar que se pasa por parámetro un entero que será convertido en una cadena de carácter con el número de transición a disparar sobre la RDPG cargada en el kernel. De esta forma si el disparo es exitoso la función del driver retornará al espacio usuario 1 (uno) si el disparo fue exitoso o 0 (cero) si el disparo no fue exitoso.

Gestionar política de la RDPG del kernel con edición de vector G

Para gestionar la política de la prioridad de las transiciones de la RDPG del kernel, el objeto de la librería dispone de un método para la edición del vector de guardas (vector G), de acuerdo a los valores del vector G, se prioriza una u otra transición de la RDPG, mecanismo que se conoce como guardas de transición de las RDPG tal como se explica en la sección 2.2.2. El siguiente ejemplo muestra cómo se edita el vector G de la RDPG cargada en el kernel:

```

/* LIBRERIA DE C. */
/* Modificación de valor de vector G de La RDPG del kernel. */
rdp.methods->set_vG(&rdp, "7_0");

/* LIBRERIA DE C++. */
/* Modificación de valor de vector G de La RDPG del kernel. */
rdp->matrixmodG_set_vG(7, 0);

```

Como muestra el ejemplo anterior en el segundo parámetro envía dos valores en una cadena de caracteres, el primero es la posición de la transición que se desea modificar y el segundo valor es el valor que se desea establecer en el vector G para esa transición.

Gestionar la política de la prioridad de transiciones de una RDPG requiere la toma de decisiones de acuerdo al estado actual de la RDPG, es por esto que es muy útil conocer el número de tokens de todas las plazas de la RDPG, para esto el objeto de la librería dispone de un método que se encarga de dicha tarea, este se muestra en el siguiente ejemplo:

```

/* LIBRERIA DE C. */
/* Declaro variable en la cual almaceno el número de tokens retornado desde el kernel. */
int n_tokens;

/* Obtener el número de tokens de una plaza de La RDPG. */
n_tokens = rdp.methods->get_TokensPlace(&rdp, "7");

/* LIBRERIA DE C++. */
/* Declaro variable en la cual almaceno el número de tokens retornado desde el kernel. */
int n_tokens;

/* Obtener el número de tokens de una plaza de La RDPG. */
n_tokens = rdp->get_TokensPlace(7);

```

6.4. CASO DE APLICACIÓN 1

Para poner a prueba el funcionamiento del driver MatrixmodG en la gestión de RDPG sobre el kernel de Linux, se analiza y estudia un caso de aplicación que hace uso de una RDPG que modela dos CPUs de un procesador dualcore. El procesador gestiona tareas para ser procesadas explotando sus diferentes cores, se representa ambos recursos en sus diferentes estados posibles, teniendo en cuenta un diseño basado en la RDP de un procesador monocore que busca un comportamiento eficiente en la gestión de tareas permitiendo un ahorro energético a la CPU del procesador cuando se terminan de procesar todas las tareas, esta RDP se extrae del informe publicado y propuesto por los autores de [27].

La idea principal de este caso de aplicación es utilizar la RDP propuesta por los autores del procesador monocore, convertir la RDP a una RDPG, estudiar y analizar la RDPG del procesador con la capacidad de expresión de las RDPG, extender la RDPG a un procesador dual Core y por ultimo manejar la política del uso de cada core del procesador utilizando guardas sobre las transiciones que lo requieran.

6.4.1. ESTUDIO DE RDP DE PROCESADOR MONOCORE

La Red de Petri de la figura 6.3 modela una CPU monocore que inicia desde un estado de espera “Stand_by” y se mueve a un estado encendido (CPU_ON) cuando llegan tareas para procesar. La CPU permanece en el estado “CPU_ON” mientras haya tareas en el búfer de la CPU. Si no hay tareas en el búfer de la CPU durante un intervalo de tiempo dado por el umbral de apagado, la CPU pasa al estado de “Stand_by” (en espera) para conservar energía, esta es la clave de la red ya que este estado permite que un procesador ahorre energía. Este modelo utiliza un generador de carga de tareas abierto ya que, cuando la transición de tasa de llegada de tareas (Arrival_rate) se desplaza para depositar una tarea en el búfer de la CPU, un token se mueve hacia atrás para estar disponible nuevamente a la transición Arrival_rate y permite así que se genere otra tarea [27].

Como se nota en el modelo de la red de la figura 6.3 las plazas representan dos recursos, estos son:

- El procesador monocore (CPU).
- Tareas para procesar (Tasks).

Además, sobre ambos recursos las plazas representan sus diferentes estados. Se pueden diferenciar tres tipos de estados:

- Estados de la CPU.
- Estados de las tareas gestionadas por la CPU.
- Estados de la CPU combinado con las tareas.

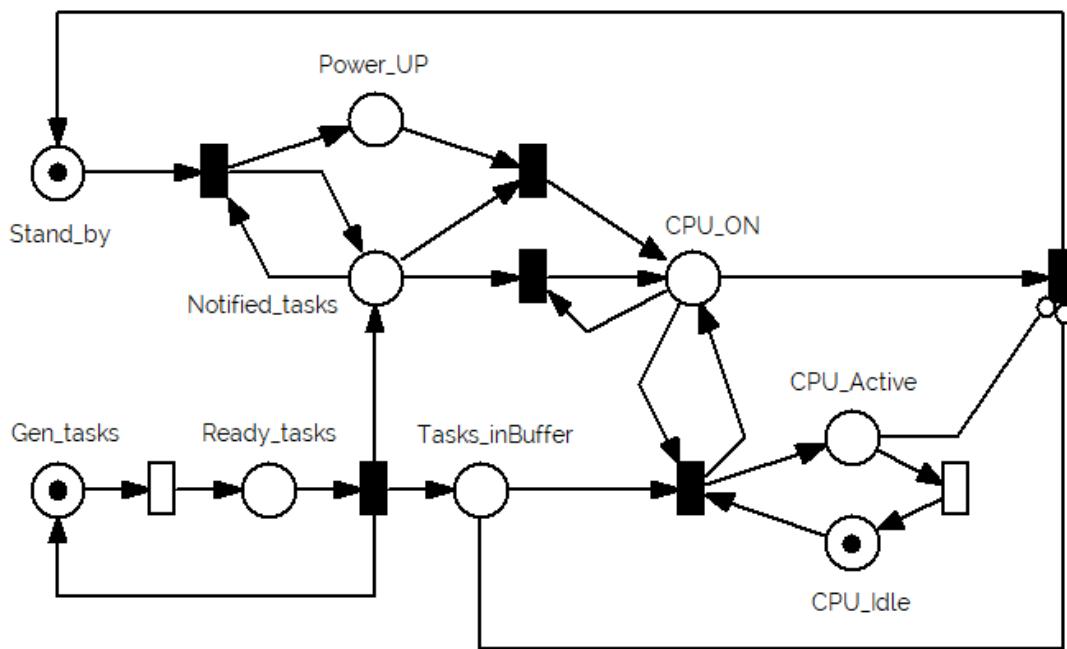


Figura 6.3 Modelo de RDP de procesador monocore: Plazas.

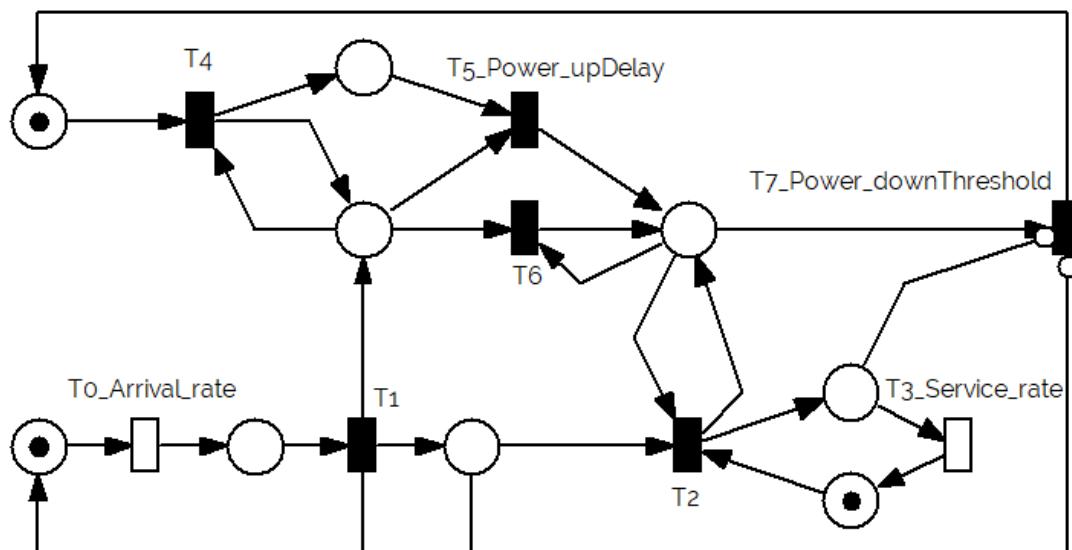


Figura 6.4 Modelo de RDP de procesador monocore: Transiciones.

Por otro lado las transiciones representan las acciones necesarias para cambiar de un estado a otro ya sea para la CPU o para las tareas a procesar por la CPU, en la figura 6.4 se detallan cada una de las transiciones que tiene la red. Se nota que hay dos tipos de transiciones:

- Transiciones inmediatas, rectángulos negros.
- Transiciones temporales, rectángulos blancos.

Las características de cada plaza de la RDP se muestran en la tabla 6.2 y las características de cada transición se muestran en la tabla 6.3 de la red.

Tabla 6.2

Plaza	Modo de uso	Descripción
Stand_by	Estado de CPU	Esta plaza representa la CPU en estado de espera (Stand_by). Es el estado que permite ahorrar energía a la CPU.
Power_up	Estado de CPU	Esta plaza representa la CPU en estado previo a su encendido. Es cuando la CPU tiene solicitud o aviso de encendido.
Notified_tasks	Estado de CPU	Esta plaza representa las notificaciones de tareas entrantes listas para procesarse por la CPU.
CPU_ON	Estado de CPU	Esta plaza representa la CPU en estado encendida. En este estado la CPU ya puede empezar a procesar tareas.
Gen_tasks	Estado de tarea	Esta plaza representa las tareas generadas.
Ready_tasks	Estado de tarea	Esta plaza representa las tareas en su estado listas para enviar a la CPU.
Task_inBuffer	Estado de tarea	Esta plaza representa las tareas que se encuentran en el buffer de la CPU. Son las tareas listas para ser procesadas por la CPU.
CPU_active	Estado de CPU y tarea	Esta plaza representa la CPU en estado activa mientras la CPU se encuentra encendida. Es el estado que representa el procesamiento de una tarea.
CPU_idle	Estado de CPU y tarea	Esta plaza representa la CPU en estado desocupado mientras la CPU se encuentra encendida.

Tabla 6.3

Transición	Tipo de transición	Descripción
T0	Temporal	Esta transición representa la acción de pasar las tareas a su estado de tarea lista para enviar a la CPU.
T1	Inmediata	Esta transición representa la acción de enviar las tareas al buffer de la CPU.
T2	Inmediata	Esta transición representa la acción de pasar las tareas a su procesamiento en la CPU y la CPU al estado activo.
T3	Temporal	Esta transición representa la acción de pasar la CPU al estado desocupado.
T4	Inmediata	Esta transición representa la acción de pasar la CPU al estado de petición de encendido.
T5	Inmediata	Esta transición representa la acción de pasar la CPU al estado encendido y limpiar la notificación de una nueva tarea en Buffer de CPU.
T6	Inmediata	Esta transición representa la acción de limpiar las notificaciones de tareas nuevas en la CPU.
T7	Inmediata	Esta transición representa la acción de pasar la CPU al estado de espera (Stand_by).

6.4.2. CONVERSIÓN DE RDP A RDPG

Como vimos en la sección anterior, se analizó y estudio la RDP de un procesador monocore. La red analizada es una RDP ordinaria, la idea de esta sección es mostrar la misma red pero modelada mediante el uso de una RDPG. La figura 6.5 muestra esta misma red expresada mediante el uso de RDPG.

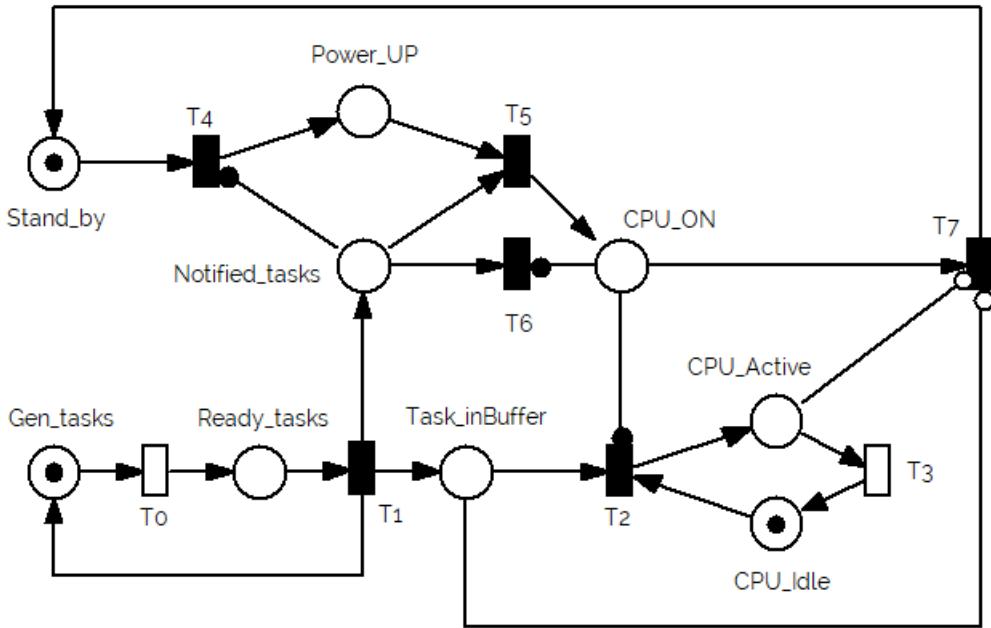


Figura 6.5 Modelo de RDPG de procesador monocore.

Como se nota en la figura 6.5 el modelo de la RDPG del procesador monocore es más simple de analizar y entender con la capacidad de expresión de las RDPG sobre sus arcos y transiciones. Con esta red se puede entender varias condiciones sobre las transiciones que se desprenden desde los arcos lectores e inhibidores, estas condiciones se detallan a en la tabla 6.4 para cada transición de la red.

Tabla 6.4

Transición	Arcos asociados	Condiciones dadas por arcos
T0	Arco común	La transición se sensibiliza cuando se cumple el tiempo la tasa de llegada de tareas (Arrival_rate) configurado en la transición y existe al menos una nueva tarea generada (Tasks_Gen).
T1	Arco común	La transición se sensibiliza cuando existe al menos una tarea lista o preparada para enviar a la CPU (Ready_tasks).
T2	Arco común, arco lector	La transición se sensibiliza cuando existe al menos una tarea en el buffer de la CPU (Tasks_inBuffer) y la CPU está encendida (arco lector a CPU_ON).
T3	Arco común	La transición se sensibiliza cuando se cumple el tiempo de procesamiento (Service_rate) configurado para las tareas en proceso (CPU_Active).
T4	Arco común, arco lector	La transición se sensibiliza cuando la CPU está en estado de espera (Stand_by) y existe al menos una notificación de nuevas tareas (arco lector a Notified_tasks).
T5	Arco común	La transición se sensibiliza cuando la CPU solicita encender el procesador (Power_up) y existe al menos una notificación de nuevas tareas (Notified_tasks).
T6	Arco común, arco lector	La transición se sensibiliza cuando la CPU desea limpiar el registro de notificaciones de nuevas tareas (arco lector a Notified_tasks).
T7	Arco común, arco inhibidor	La transición se sensibiliza cuando la CPU se encuentra en estado encendida pero siempre que no haya tareas procesándose (arco inhibidor a CPU_Active) ni tareas en el buffer de la CPU (arco inhibidor a Tasks_inBuffer).

6.4.3. MODIFICACIONES DE RDPG

Las RDPG expresan mayor capacidad de expresión al modelado con redes de Petri, por esta razón los modelos son más simples y sencillos de analizar. Como se nota en el modelo de la RDPG del procesador monocore la complejidad observada desde su RDP de origen, se vio simplificado en gran medida, pero además el nuevo modelo permite detectar algunos inconvenientes que pueden ser resueltos con un rediseño de la RDPG. Esta sección muestra el inconveniente detectado sobre la RDPG monocore y dos posibles mejoras.

El inconveniente detectado se da sobre la plaza “Notified_tasks”, ya que brinda información redundante. Se trata de una notificación a la CPU de una nueva tarea, el problema es que esta plaza no se limpia por cada vez que se ejecuta una tarea en la CPU, es decir que la plaza se limpia solo cuando se enciende la CPU, pero mientras se mantenga encendida no se vuelve a limpiar, a excepción de que se limpie externamente ejecutando T6 mientras la CPU está encendida. La información de esta plaza es redundante ya que es la misma información que representa la plaza de tareas en el buffer de la CPU (Tasks_inBuffer). Cuando la CPU se mantiene encendida, suponiendo que ai una cola de tareas en el buffer de la CPU, se presenta un problema grave, debido a que existirá el mismo número de notificaciones pero mientras la CPU resuelva todas las tareas, las notificaciones quedarán registradas, es por lo cual que en caso de no liberar todas las notificaciones con T6 mientras la CPU está encendida, la CPU se encenderá y apagará hasta agotar el número de notificaciones.

Para resolver el problema descripto anteriormente con las posibilidades que brindan las RDPG existen varias soluciones, aquí analizaremos solo dos:

- La primera solución (figura 6.6) se basa manteniendo la plaza “Notified_tasks” y utilizando un arco reset para limpiar todas las notificaciones cuando la CPU se enciende.
- La segunda solución (figura 6.7) se basa en eliminar la plaza “Notified_tasks”, debido a que la plaza “Tasks_inBuffer” representa la misma información, y se elimina la transición T6 ya que la transición T2 puede cumplir el mismo rol automáticamente.

En la figura 6.6 se puede observar que se reduce el riesgo de que la CPU se encienda y apague involuntariamente hasta agotar las notificaciones, ya que cada encendido de la CPU (T5) produce que se limpien todas las notificaciones gracias al arco reset, no obstante puede presentarse el caso de que mientras este encendida la CPU ingresen más tareas y sea necesario limpiar las notificaciones, para este caso la CPU se encenderá como máximo una sola vez para limpiar las notificaciones a excepción de que se ejecute la transición T6 antes de apagar la CPU.

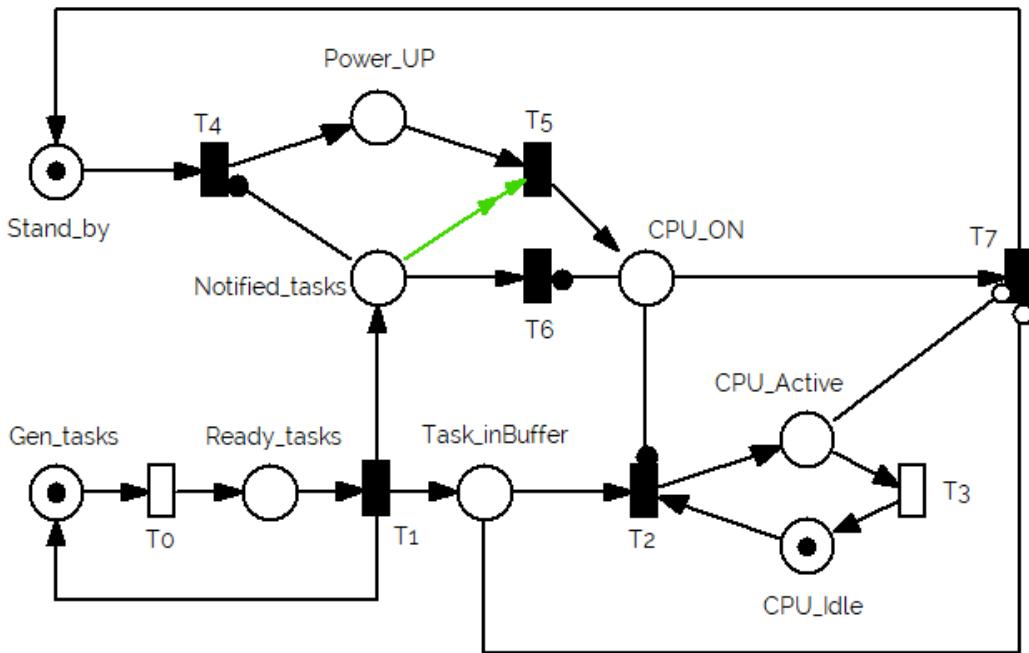


Figura 6.6 Modelo de procesador monocore alternativa 1.

La figura 6.7 muestra un modelo más simple eliminando directamente la plaza “Notified_tasks” y su transición para la limpieza de notificaciones (T6), haciendo que todo esto sea realizado por la plaza “Tasks_inBuffer” en conjunto con la transición T2. Es decir que la plaza “Tasks_inBuffer” además de representar las tareas en el buffer de la CPU son el medio de notificación a tener en cuenta para encender la CPU mientras que la transición T2 se encarga de pasar las tareas del buffer a su procesamiento y además de limpiar el buffer de notificaciones ya que es representado por la misma plaza.

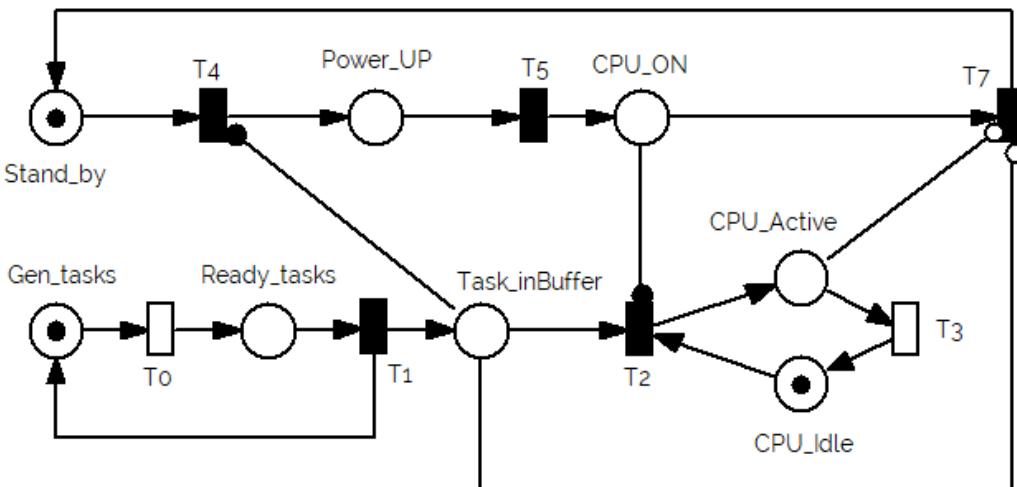


Figura 6.7 Modelo de procesador monocore alternativa 2.

Se decidió utilizar la segunda alternativa para el modelado de la RDPG del procesador monocore (figura 6.7), ya que esta red cumple el mismo funcionamiento que la RDP original y proporciona un diseño más simple y eficiente.

6.4.4. EXTENSIÓN DE RDPG A PROCESADOR DUALCORE

El caso de aplicación de interés se basa en la simulación de una RDPG que represente el modelo de los estados de un procesador dual Core respecto al procesamiento de tareas y que impacta en su ahorro energético. Para esto se extiende la RDPG del procesador monocore a un procesador dual Core, donde se administra y controla

cada CPU del procesador en busca de un comportamiento eficiente en la gestión de tareas permitiendo su ahorro energético cuando se acaben todas las tareas, tal como lo hace el procesador monocore analizado en la sección anterior.

La extensión de la RDPG del procesador monocore al procesador dual Core es bastante sencilla, basta con replicar análogamente cada uno de los estados de otra CPU para representar en este caso dos CPU diferentes. La figura 6.8 muestra el modelo de la RDPG del procesador dual Core. En el modelo se nota que cada tarea generada puede ser procesada por la CPU1 o la CPU2 del procesador, para controlar esto se debe definir una política para las guardas de las transiciones T1 y T7 (G(T1) y G(T7)), la próxima sección explica la política utilizada.

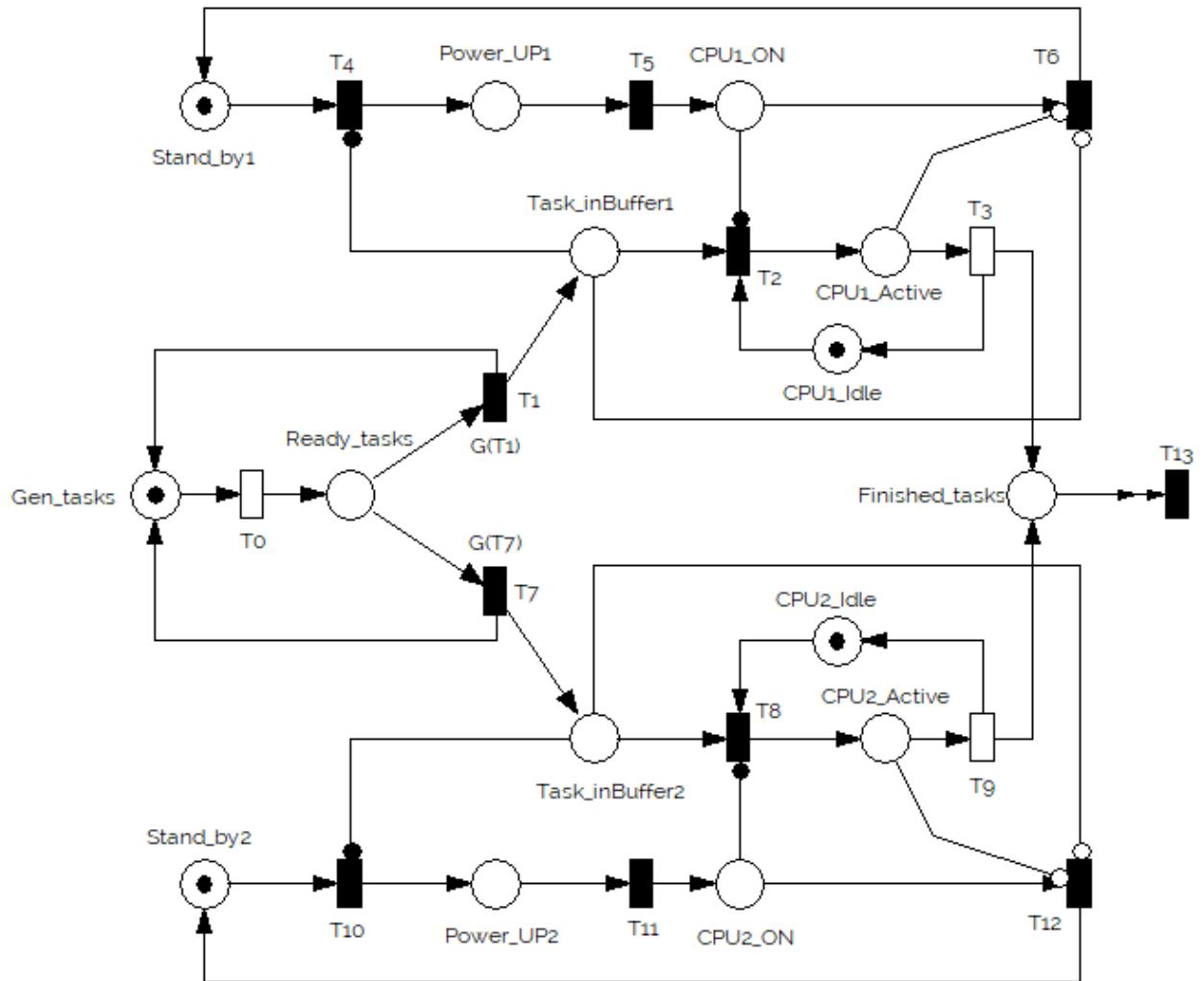


Figura 6.8 Modelo de RDPG para procesador dual Core.

Los detalles de las plazas y transiciones de la CPU1 se describen en las tablas 6.5 y 6.6. No se describen los detalles de las plazas y transiciones de la CPU2 ya que por analogía son iguales a los detalles de la CPU1 respectivamente. Por ultimo para explotar todas las capacidades de expresión de las RDPG que gestiona el módulo MatrixmodG, se hace uso de un arco reset para vaciar la plaza “Finished_tasks” que representa todas las tareas finalizadas por el procesador.

Tabla 6.5

Plaza	Modo de uso	Descripción
Gen_tasks	Estado de tarea	Esta plaza representa las tareas generadas.
Ready_tasks	Estado de tarea	Esta plaza representa las tareas en su estado listas para enviar a la CPU.
Stand_by1	Estado de CPU	Esta plaza representa la CPU1 en estado de espera (Stand_by1). Es el estado que permite ahorrar energía a la CPU1.
Power_up1	Estado de CPU	Esta plaza representa la CPU1 en estado previo a su encendido. Es cuando la CPU1 tiene solicitud o aviso de encendido.
CPU1_ON	Estado de CPU	Esta plaza representa la CPU1 en estado encendida. En este estado la CPU1 ya puede empezar a procesar tareas.
Task_inBuffer1	Estado de tarea	Esta plaza representa las tareas que se encuentran en el buffer de la CPU1. Son las tareas listas para ser procesadas por la CPU1.
CPU1_active	Estado de CPU y tarea	Esta plaza representa la CPU1 en estado activa mientras la CPU1 se encuentra encendida. Es el estado que representa el procesamiento de una tarea.
CPU1_idle	Estado de CPU y tarea	Esta plaza representa la CPU1 en estado desocupado mientras la CPU1 se encuentra encendida.

Tabla 6.6

Transición	Tipo de transición	Descripción
T0	Temporal	Esta transición representa la acción de pasar las tareas a su estado de tareas listas para enviar a la CPU correspondiente.
T1	Con guarda	Esta transición representa la acción de enviar las tareas al buffer de la CPU1. Si la guarda de la transición es verdadera la acción de la guarda se llevará a cabo, si es falsa no se lleva a cabo.
T2	Inmediata	Esta transición representa la acción de pasar las tareas a su procesamiento en la CPU1 y la CPU1 al estado activo.
T3	Temporal	Esta transición representa la acción de pasar la CPU1 al estado desocupado (idle). La transición es temporal, representa el tiempo de procesamiento de cada tarea.
T4	Inmediata	Esta transición representa la acción de pasar la CPU1 al estado de petición de encendido.
T5	Inmediata	Esta transición representa la acción de pasar la CPU1 al estado encendido.
T6	Inmediata	Esta transición representa la acción de pasar la CPU1 al estado de espera (Stand_by1).
T13	Inmediata	Esta transición representa la acción de limpiar el número de las tareas procesadas por la CPU1 y CPU2 del procesador.

6.4.5. POLÍTICA DE RDPG DE PROCESADOR DUALCORE

Para explotar las capacidades de expresión de las RDPG, se decide utilizar transiciones con guardas. Con el uso de guardas sobre las transiciones, las RDPG permiten un control más sencillo sobre la política, para este caso de estudio la política que se presenta es la que responde a las preguntas ¿Cómo se distribuyen las tareas entre cada CPU? Y ¿Qué CPU utilizar mientras exista una cola de tareas? La respuesta es definiendo una política, y en este caso la política es utilizar la CPU que menor trabajo de tareas tenga, será la CPU que tomará la nueva tarea. Para saber qué trabajo tiene cada CPU se debe analizar la cantidad de tareas que tiene la CPU en su buffer, de esta forma se debe analizar la plazas “Tasks_inBuffer” de cada CPU conociendo el trabajo de tareas que tiene cada una.

La política que se utilizará consiste en procesar la primer tarea con la CPU1, cuando ingresa otra tarea nueva se deberá analizar el trabajo de cada CPU (plaza Tasks_inBuffer), si se da el caso de que la cantidad de tareas del buffer de la CPU1 es mayor que la cantidad de tareas en el buffer de la CPU2 entonces se envía la tarea al buffer de la CPU2. Las condiciones que determinan esta política son las siguientes:

- Si $M(\text{Tasks_inBuffer1}) \leq M(\text{Tasks_inBuffer2})$: Procesara la tarea la CPU1, se inhibe la transición T7. $M(\text{Tasks_inBuffer1})$ es la marca de la plaza Task_inBuffer1 y que representa la cantidad de tareas en el buffer de la CPU1. Análogamente $M(\text{Tasks_inBuffer2})$ es la marca de la plaza Task_inBuffer2.
- Si $M(\text{Tasks_inBuffer1}) > M(\text{Tasks_inBuffer2})$: Procesara la tarea la CPU2, se inhibe la transición T1.

Estas dos condiciones controlan las guardas que se utilizan sobre las transiciones que envían las tareas a la cada CPU ($G(T1)$ y $G(T7)$), de acuerdo con la figura 6.8 son la transición T1 y la transición T7. Esto quiere decir que solo se podrá enviar una tarea a la CPU1 solo si el trabajo de la CPU1 es menor o igual al trabajo de la CPU2, mientras que se podrá enviar una tarea a la CPU2 solo si el trabajo de la CPU1 es mayor que el trabajo de la CPU2. De esta forma la RDPG permite controlar la política para la gestión de tareas y los propios CPUs. En conclusión con las guardas se inhibe T1 o T7:

- T1 se sensibiliza si además de las condiciones dadas por los arcos (tabla 6.3) se cumple la condición de la guarda, para esta transición la condición de guarda es $M(\text{Tasks_inBuffer1}) \leq M(\text{Tasks_inBuffer2})$.
- T7 se sensibiliza si además de las condiciones dadas por los arcos (análoga a T1 en CPU2) se cumple la condición de la guarda, para esta transición la condición de guarda es $M(\text{Tasks_inBuffer1}) > M(\text{Tasks_inBuffer2})$.

6.4.6. COMPONENTES BASE DE LA RDPG

Se entiende por componentes base de la RDPG a todos los componentes necesarios para la creación de la estructura de la misma, estos son:

- Matriz de incidencia I.
- Matriz de incidencia H de brazo inhibidores.
- Matriz de incidencia R de brazos lectores.
- Matriz de incidencia Re de brazos reset.
- Vector de marcado inicial.

Con todos los componentes de la lista anterior se puede iniciar la creación de una nueva RDPG.

Para mantener una mejor organización de plazas y transiciones se resume el nombre de las plazas de la RDPG en estudio, siguiendo el modelo de la figura 6.9. De acuerdo con la RDPG de la figura 6.9, para este caso de aplicación en estudio, los componentes base de la red son los siguientes:

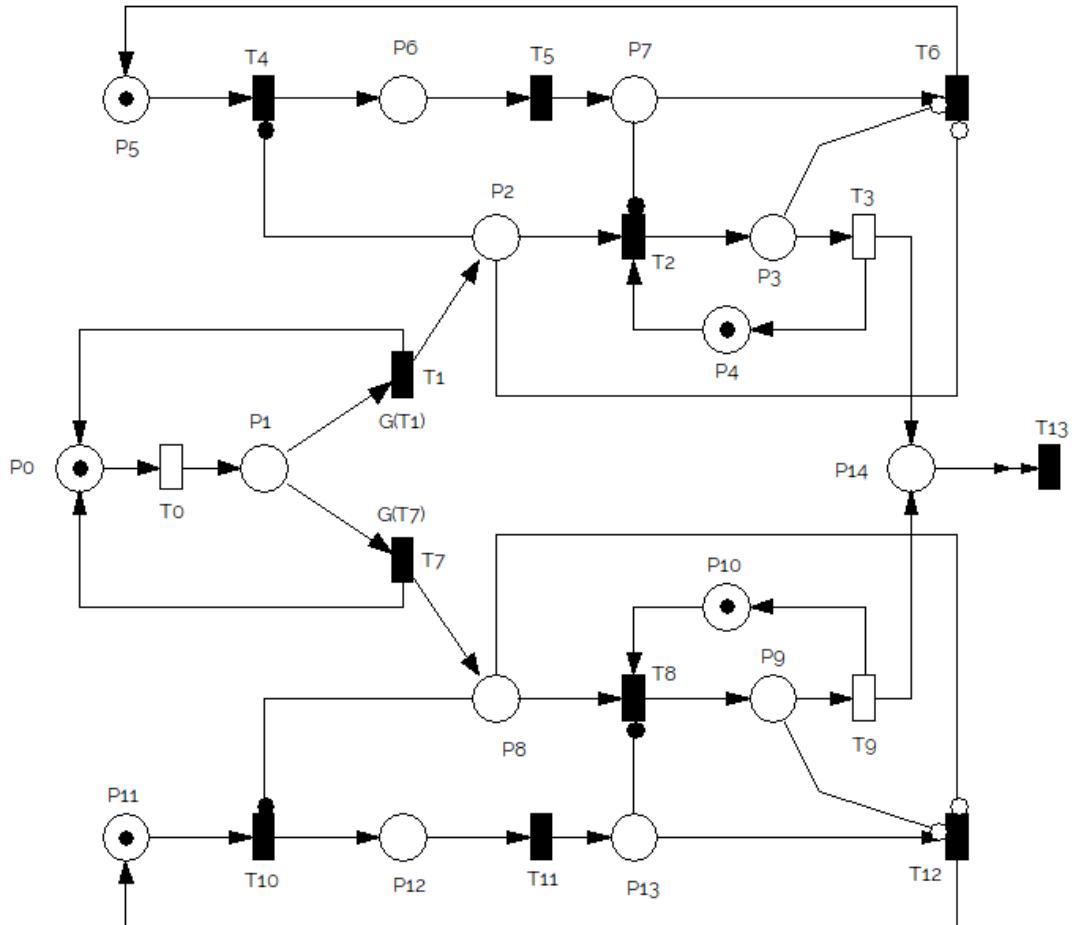


Figura 6.9 RDPCG de procesador dual core, resumen en nombre de plazas.

Matriz de incidencia I														
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
P0	-1	1	0	0	0	0	0	1	0	0	0	0	0	0
P1	1	-1	0	0	0	0	0	0	-1	0	0	0	0	0
P2	0	1	-1	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	1	-1	0	0	0	0	0	0	0	0	0	0
P4	0	0	-1	1	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	-1	0	1	0	0	0	0	0	0	0
P6	0	0	0	0	1	-1	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	1	-1	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	1	-1	0	0	0	0	0
P9	0	0	0	0	0	0	0	0	1	-1	0	0	0	0
P10	0	0	0	0	0	0	0	0	-1	1	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	-1	0	1	0
P12	0	0	0	0	0	0	0	0	0	0	1	-1	0	0
P13	0	0	0	0	0	0	0	0	0	0	0	1	-1	0
P14	0	0	0	1	0	0	0	0	0	1	0	0	0	0

Vector de marcado inicial														
P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
1	0	0	0	1	1	0	0	0	0	1	1	0	0	0

	Matriz de incidencia H^T													
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
P0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	0	0	0	1	0	0	0	0	0	0	0
P3	0	0	0	0	0	0	1	0	0	0	0	0	0	0
P4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	0	0	0	0	0	0	1
P9	0	0	0	0	0	0	0	0	0	0	0	0	1	0
P10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	Matriz de incidencia R^T													
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
P0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	0	1	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P7	0	0	1	0	0	0	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	0	0	0	1	0	0	0
P9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	1	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	Matriz de incidencia Re^T													
	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
P0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P3	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P8	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P9	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P10	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	0	0	0	0	1

6.4.7. MODO DE EJECUCIÓN DE LA RDPG

Desde la capa de aplicación, se trabajó con ayuda de monitores, mutex y variables de condición, para lograr gestionar las RDPG con múltiples procesos concurrentes brindando protección mediante la exclusión mutua del recurso compartido. En esta oportunidad para el presente caso de aplicación la RDPG planteada puede ser ejecutada de tres formas diferentes.

- Ejecución de RDPG con proceso principal, se plantea ejecutar la RDPG con un solo hilo.
- Ejecución de RDPG con un proceso por CPU y generación de tareas por hilo principal.
- Ejecución de RDPG con múltiples hilos por cada CPU, se plantea una ejecución de la RDPG con cinco hilos.

Para el primer caso la idea es ejecutar la RDPG mediante un solo hilo sabiendo de antemano cuales son los disparos de transiciones activas (o sensibilizadas), se tiene éxito en el disparo de cada una de las transiciones. Es una ejecución que puede demostrar un gran rendimiento, pero no existe la posibilidad de obtener eficiencia a través de la gestión de hilos. La figura 6.10 muestra como el hilo principal va disparando las transiciones de la RDPG y saltando al próximo disparo sabiendo de antemano cual es el siguiente disparo en cada caso.

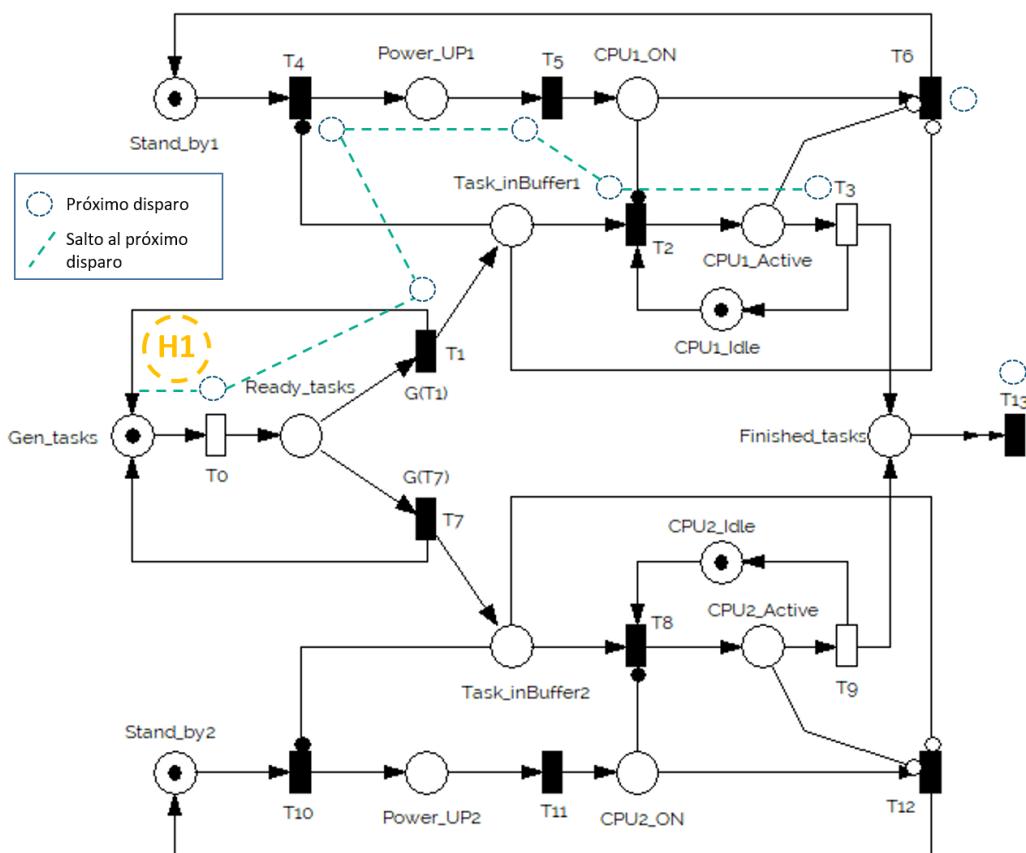


Figura 6.10 Ejecución de RDPG procesador dual core con 1 hilo.

El segundo planteo se basa en una mejora del primero, agregando la ayuda de más procesos para su ejecución y mejorando más el rendimiento ya que más hilos gestionan la RDPG de manera concurrente con protección de recurso, pero en este caso no se adiciona eficiencia en la gestión de los procesos, ya que también se plantea que los procesos ejecuten cada uno de los disparos de transiciones correctos. Este enfoque se analiza en la figura 6.11, donde se puede ver como cada proceso va disparando sus transiciones una seguida de la otra sabiendo cual es la siguiente sin posibilidad de fallo en ninguno de los disparos. Primero el hilo H1 genera todas las tareas así, ya los buffer de cada CPU están llenos con tareas distribuidas de acuerdo a la política analizada en la sección 6.4.5. Luego los hilos H2 y H3 se ejecutan paralelamente ejecutando un disparo de transición detrás de otro,

encendiendo la respectiva CPU cuando sea necesario y ejecutando las tareas hasta vaciar el buffer asociado a la CPU.

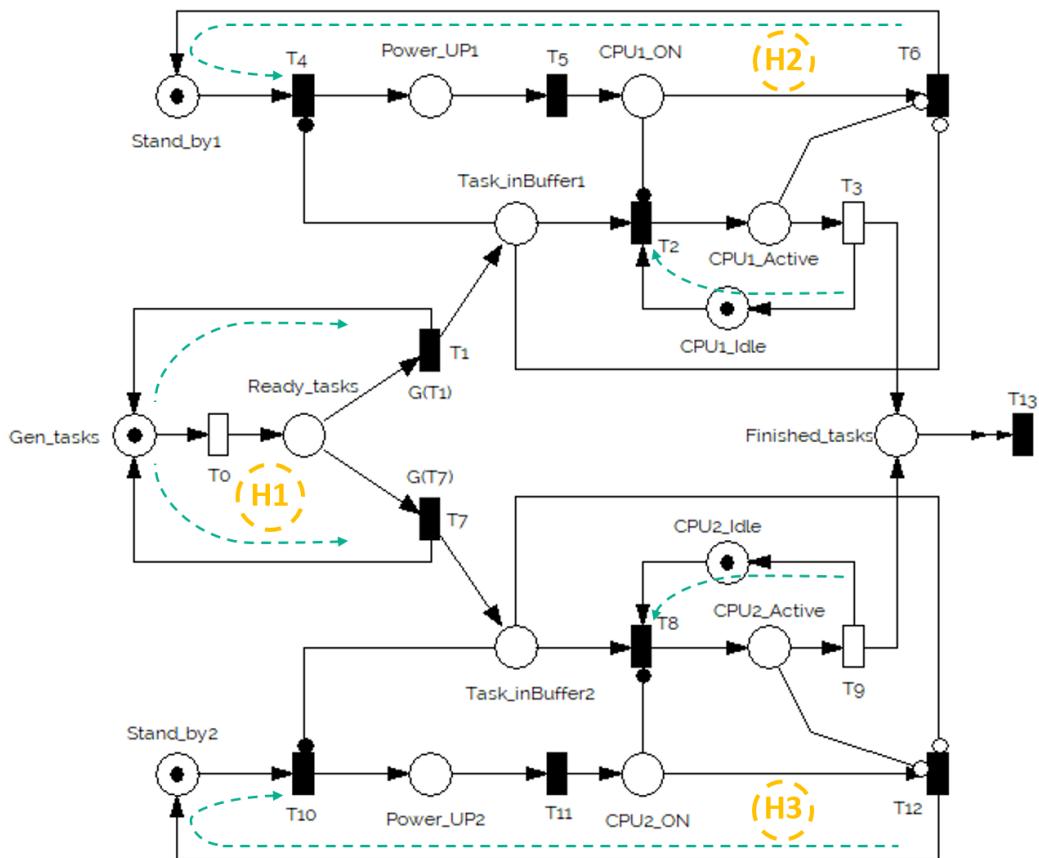


Figura 6.11 Ejecución de RDPG procesador dual core con 3 hilos.

Por último, el tercer caso plantea una ejecución por un conjunto de cinco hilos independientes donde cada hilo tiene una tarea particular. Estos son:

- **Hilo 1 (H1):** Genera tareas para las CPU1 y CPU2.
- **Hilo 2 (H2):** Enciende o apaga CPU1.
- **Hilo 3 (H3):** Activa o desactiva el procesamiento de una tarea en la CPU1.
- **Hilo 4 (H4):** Enciende o apaga la CPU2.
- **Hilo 5 (H5):** Activa o desactiva el procesamiento de una tarea en la CPU2.

La figura 6.12 muestra esta idea de cómo cada hilo utiliza la RDPG.

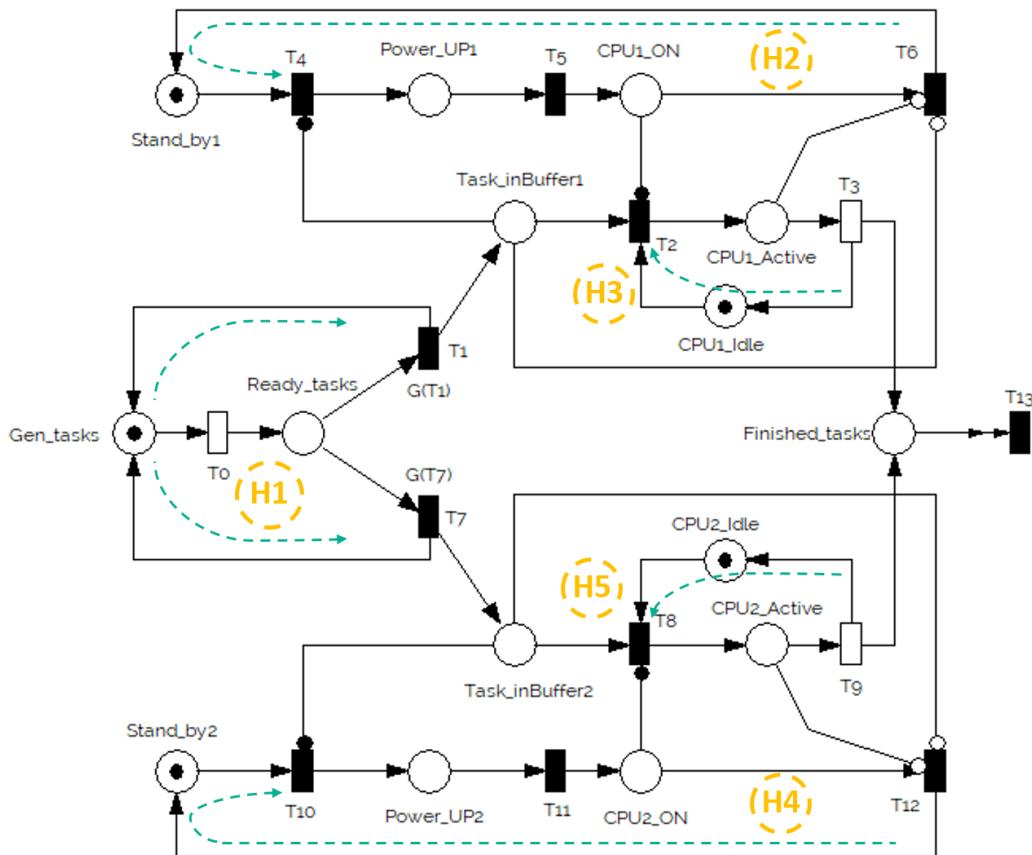


Figura 6.12 Ejecución de RDPG procesador dual core con 5 hilos.

La diferencia de este modo de ejecución planteado es que se provee eficiencia en la gestión de la RDPG dando la posibilidad de lanzar todos los hilos al mismo tiempo (H2, H3, H4 y H5) a excepción del hilo principal (H1). Los hilos pueden disparar y fallar el disparo de la transición, permitiendo en este caso que se duerma el hilo y sea despertado cuando se lo necesite. El hilo principal (H1) inicia la generación de una tarea y despierta los hilos que realizan la gestión de cada CPU según sea el caso. De esta forma al momento de generar cada una de las tareas las CPU las van procesando o esperan en el buffer hasta que la CPU pueda procesarlas. Este enfoque es más inteligente ya que permite que un proceso falle cualquier disparo haciendo que se duerma y sea despertado por otro proceso cuando sea necesario, puede que se presenten demoras por la gestión de hilos, por lo que es de gran para el caso de RDPG que tengan transiciones temporales con tiempos de operación altos, para el caso de RDPG de transiciones inmediatas se podrían notar bajas de rendimiento por la gestión de los procesos.

6.5. CASO DE APLICACIÓN 2

Para poner a prueba el funcionamiento del driver MatrixmodG en la gestión de RDP sobre el kernel de Linux, se analiza y estudia un caso de aplicación que hace uso de una RDP que modela un productor y un consumidor involucrados en el uso de un mismo recurso compartido, representado como un buffer limitado. La figura 6.13 muestra la RDP mencionada.

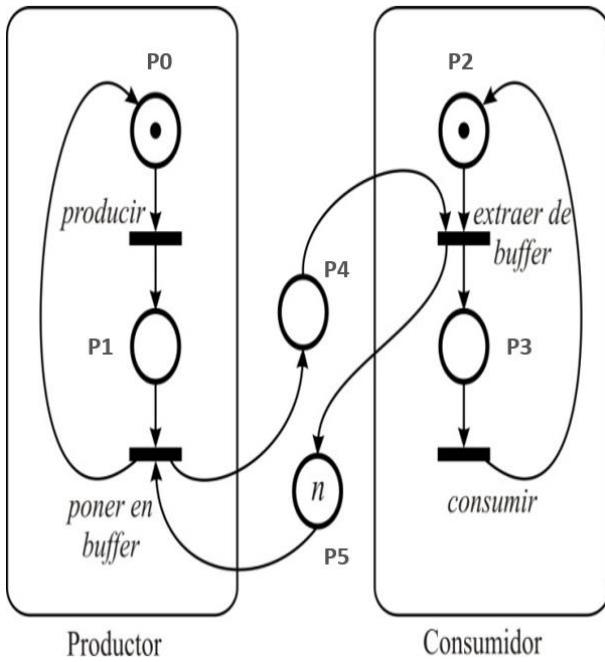


Figura 6.13 Modelo de RDP Productor/consumidor con buffer limitado a n.

6.5.1. ESTUDIO DE RDP PRODUCTOR-CONSUMIDOR

La Red de Petri de la figura 6.13 modela un productor y un consumidor que operan sobre un buffer de datos limitado. Por el lado del productor, este inicia como productor “listo para producir” (P0), para pasar al estado “productor produciendo” (P1) debe disparar la transición “producir”, para finalizar con el proceso de producción de un dato, deberá disparar la transición “poner en buffer” pero verificando que el buffer tenga disponible espacio (P5), de lo contrario debería esperar. Si existe espacio en el buffer se coloca el dato en el buffer (P4) y finaliza el proceso, luego continua produciendo de la misma manera sucesivamente.

Por el lado del consumidor, este inicia como consumidor “listo para consumir” (P2), si hay datos producidos en el buffer (P4), puede continuar con el consumo del mismo, pero si no hay datos este proceso deberá esperar. Suponiendo el caso de que existan datos, el consumidor debe disparar la transición “extraer de buffer” para pasar al estado de “consumidor consumiendo” (P3) en dicha transición se habilita nuevamente un espacio para producir (P5) ya que el consumidor extrajo un dato del buffer equivalente a liberar espacio en el buffer (P5), para finalizar con el proceso de consumo y volver al estado de “listo para consumir” deberá disparar la transición “consumir” y el proceso continua de esta manera sucesivamente.

6.5.2. COMPONENTES BASE DE LA RDP

Los componentes base de la RDP del caso de aplicación son:

- Matriz de incidencia I.
- Vector de marcado inicial.

Con los dos componentes de base se puede iniciar la creación de una nueva RDP.

Para mantener una mejor organización de plazas y transiciones se resume el nombre de las transiciones de la RDP en estudio, de la siguiente manera:

- **T0:** producir.
- **T1:** poner en buffer.
- **T2:** extraer de buffer.
- **T3:** consumir.

Matriz de incidencia I

	T0	T1	T2	T3
P0	-1	1	0	0
P1	1	-1	0	0
P2	0	0	-1	1
P3	0	0	1	-1
P4	0	1	-1	0
P5	0	-1	1	0

Vector de marcado inicial

P0	P1	P2	P3	P4	P5
1	0	1	0	0	10

6.5.3. MODO DE EJECUCIÓN DE LA RDP

Al igual que el caso de aplicación 1, en este caso de aplicación para la RDP modelada se puede ejecutar la RDP de la misma forma. En donde de manera análoga se propone que la ejecución se realice utilizando variables de condición para gestionar los procesos, y sin variables de condición ejecutando los disparos de cada transición conociéndolas de antemano. Para las ejecuciones de la RDP también se hace uso del monitor quien provee exclusión mutua en capa de aplicación y las variables de condición para la gestión de procesos.

6.6. RENDIMIENTO DEL SISTEMA

Optimizar el rendimiento del sistema desarrollado fue uno de los desafíos centrales del proyecto. Teniendo siempre esta idea en mente, el diseño e implementación de la librería del driver MatrixmodG se realizó de manera tal para contribuir a la obtención de los resultados de tiempos de respuesta de cada operación que realiza el driver de una forma simple de usar.

Las mediciones de tiempo se realizan todas desde el espacio usuario. Como la librería del driver se programó en C y en C++, se utilizaron dos herramientas para medir el tiempo, estas son:

- API OpenMP (omp.h).
- Librería time.h.

El uso de dos herramientas para la obtención de los resultados de tiempos da una mayor certeza a los resultados obtenidos y una doble medición para tener referencia desde dos herramientas diferentes como referencia.

Todos los resultados de tiempo se realizaron con la habilitación de las macros de mensajes de debug y luego con su deshabilitación para tener un parámetro de la mejora de rendimiento que se obtiene al no registrar movimientos en el espacio kernel como también en el espacio usuario.

6.6.1. TIEMPOS DE ASIGNACIÓN DE MEMORIA DINÁMICA

Con las macros de mensajes de debug habilitadas, los resultados obtenidos de las mediciones de tiempo sobre el driver MatrixmodG al asignar memoria dinámica en el kernel de Linux con las diferentes alternativas disponibles, tal como se describe en la tabla 6.1 de la sección 6.1.1, se presentan en las tablas 6.7 y 6.8:

Tabla 6.7: PROMEDIO DE TIEMPOS DE METODOS DE ALLOC EN DRIVER (CON API OPENMP)					
Tamaño de RDPG	KMALLOC	KZALLOC	KCALLOC	V_MALLOC	VZALLOC
RDPG 10x10	0,5 ms	0,4 ms	0,4 ms	1 ms	1,2 ms
RDPG 100x100	1,2 ms	1,1 ms	1,4 ms	2 ms	5 ms
RDPG 500x500	3 ms	2 ms	2 ms	11 ms	14 ms
RDPG 1000x1000	7 ms	5 ms	5 ms	27 ms	30 ms

Tabla 6.8: PROMEDIO DE TIEMPOS DE METODOS EN ALLOC DE DRIVER (CON LIB TIME)					
Tamaño de RDP	KMALLOC	KZALLOC	KCALLOC	V_MALLOC	VZALLOC
RDPG 10x10	0,2 ms	0,2 ms	0,3 ms	0,5 ms	0,7 ms
RDPG 100x100	0,4 ms	0,4 ms	0,4 ms	1,5 ms	3 ms
RDPG 500x500	2 ms	1,5 ms	1,5 ms	10 ms	12 ms
RDPG 1000x1000	6 ms	4 ms	4 ms	25 ms	29 ms

En la tabla 6.7 se muestran las mediciones de tiempo con la API OpenMP, mientras que en la tabla 6.8 las mediciones son de la librería time.h, en ambos casos se muestra el tiempo que demora el driver en asignar memoria dinámicamente en el kernel de Linux para el caso de RDPG de 10, 100, 500 y hasta 1000 plazas y transiciones. Todas las mediciones se encuentran en la unidad de milisegundos (ms).

Lo primero a destacar de los resultados es que entre la comparativa de los métodos de la familia kmalloc (asignación de memoria físicamente contigua) son mucho más rápido que los métodos de la familia vmalloc (asignación de memoria lógicamente contigua), este efecto se nota a medida que más memoria se necesita para una RDPG, por ejemplo para el caso de una RDPG de 1000 plazas y 1000 transiciones los métodos de vmalloc son muchos más lentos que los métodos de la familia kmalloc.

Por otro lado dentro de los métodos kmalloc, se puede decir que los mejores resultados se obtienen para los casos de kzalloc y kcalloc, que son las versiones optimizadas de la versión kmalloc por reservar la memoria y asignar los valores a cero a diferencia de la versión original que requiere de una asignación a cero manual de los valores luego de la asignación de memoria.

Con estos resultados se pudo identificar que los métodos kzalloc y kcalloc, son los mejores métodos para asignar memoria dinámicamente en el kernel de Linux de acuerdo a los objetivos buscados para el presente proyecto. De esta forma se decidió utilizar el método kzalloc como método por defecto en el driver matrixmodG para realizar las asignaciones de memoria dinámicamente en el kernel de Linux, pero se puede modificar por las versiones alternativas en cualquier momento.

De la misma forma que las tablas 6.7 y 6.8, en las tablas 6.9 y 6.10 se obtienen los resultados del tiempo en asignar memoria dinámicamente en el kernel con las diferentes alternativas pero en este caso realizando la deshabilitación de las macros de mensajes de debug del driver MatrixmodG, como de describe en la sección 4.2.4 del capítulo 4, los resultados son:

Tabla 6.9: PROMEDIO DE TIEMPOS DE METODOS DE ALLOC EN DRIVER (CON API OPENMP)

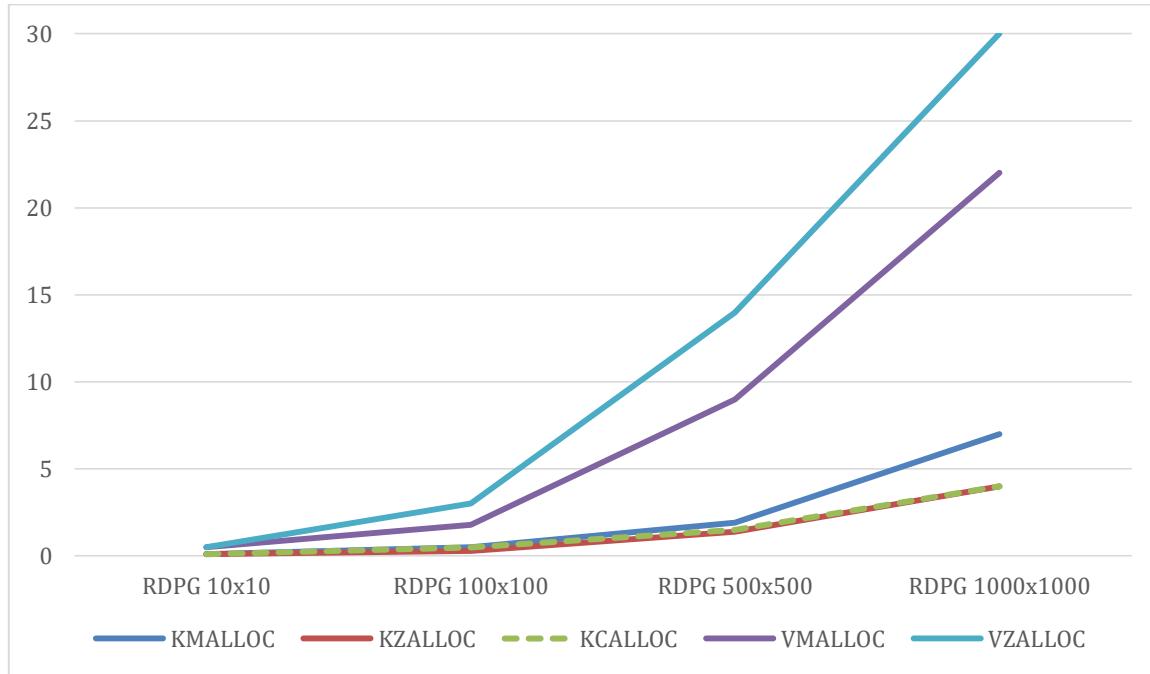
Tamaño de RDPG	KMALLOC	KZALLOC	KCALLOC	VMALLOC	VZALLOC
RDPG 10x10	0,1 ms	0,1 ms	0,1 ms	0,5 ms	0,5 ms
RDPG 100x100	0,5 ms	0,3 ms	0,5 ms	1,8 ms	3 ms
RDPG 500x500	1,9 ms	1,4 ms	1,5 ms	9 ms	14 ms
RDPG 1000x1000	7 ms	4 ms	4 ms	22 ms	30 ms

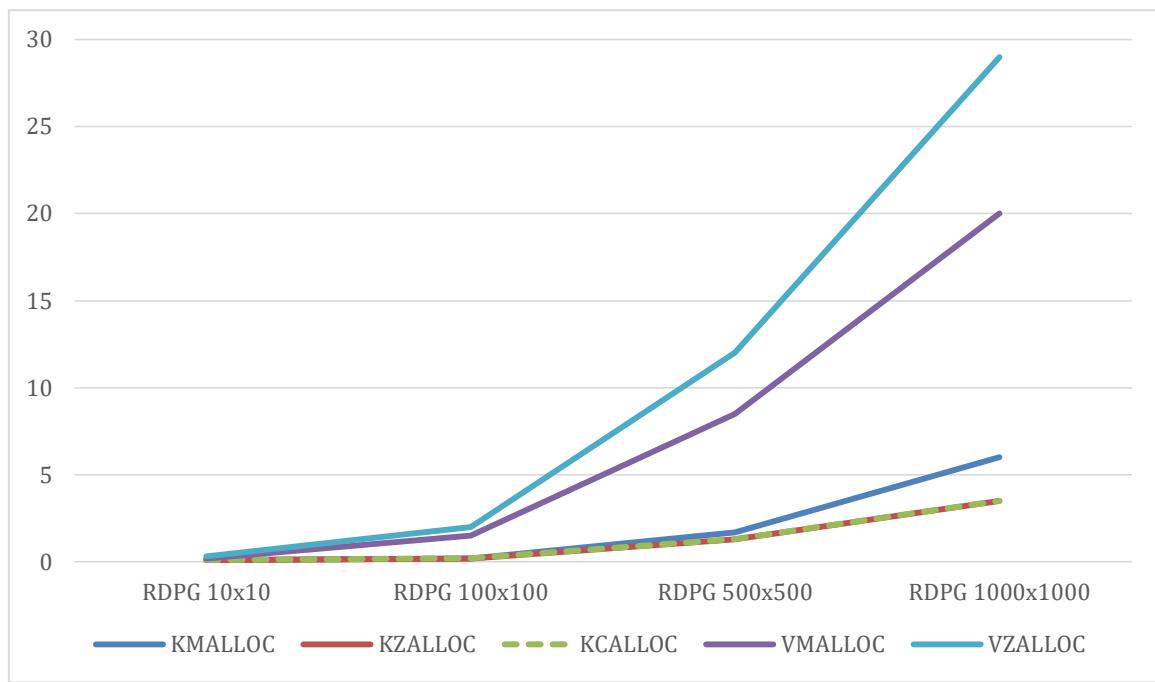
Tabla 6.10: PROMEDIO DE TIEMPOS DE METODOS EN ALLOC DE DRIVER (CON LIB TIME)

Tamaño de RDP	KMALLOC	KZALLOC	KCALLOC	VMALLOC	VZALLOC
RDPG 10x10	0,1 ms	0,1 ms	0,1 ms	0,2 ms	0,3 ms
RDPG 100x100	0,2 ms	0,2 ms	0,2 ms	1,5 ms	2 ms
RDPG 500x500	1,7 ms	1,3 ms	1,3 ms	8,5 ms	12 ms
RDPG 1000x1000	6 ms	3,5 ms	3,5 ms	20 ms	29 ms

Los resultados de las tablas 6.9 y 6.10 muestran el mismo comportamiento que las tablas 6.7 y 6.8 respectivamente, donde se observan mejoras significativas de los tiempos de asignación de memoria dinámica de todos los métodos, gracias a la desactivación de los mensajes de debug.

Las gráficas 6.1 y 6.2 muestran las curvas de tiempos para los diferentes tamaños que puede tener una RDPG en el kernel de Linux. La grafica 6.1 se corresponde a las mediciones de tiempo con la API OpenMP de la tabla 6.9. Por el lado de la gráfica 6.2, se corresponde con las mediciones de tiempo con la librería time.h de la tabla 6.10.

**Grafica 6.1** Tiempos de asignación de memoria dinámica en el kernel (API OpenMP).



Grafica 6.2 Tiempos de asignación de memoria dinámica en el kernel (lib time.h).

Analizando las gráficas 6.1 y 6.2, se puede observar más en detalle la mejora de rendimiento que tiene el uso de los métodos kzalloc y kcalloc para la asignación de memoria en comparación con el resto de los métodos de asignación.

Para la liberación de la memoria reservada de una RDPG en el kernel de Linux, se obtuvieron los resultados de las tablas 6.11 y 6.12:

Tabla 6.11: PROMEDIO DE TIEMPOS DE OPERACIÓN DELETE EN DRIVER (CON API OPENMP)					
Tamaño de RDP	KMALLOC	KZALLOC	KCALLOC	VMALLOC	VZALLOC
RDPG 10x10	0,06 ms	0,07 ms	0,06 ms	0,2 ms	0,2 ms
RDPG 100x100	0,1 ms	0,1 ms	0,1 ms	0,8 ms	0,9 ms
RDPG 500x500	0,4 ms	0,4 ms	0,4 ms	7 ms	7 ms
RDPG 1000x1000	0,8 ms	0,9 ms	0,7 ms	9 ms	10 ms

Tabla 6.12: PROMEDIO DE TIEMPOS DE OPERACIÓN DELETE EN DRIVER (CON LIB TIME)					
Tamaño de RDP	KMALLOC	KZALLOC	KCALLOC	VMALLOC	VZALLOC
RDPG 10x10	0,06 ms	0,07 ms	0,06 ms	0,2 ms	0,2 ms
RDPG 100x100	0,1 ms	0,1 ms	0,1 ms	0,8 ms	0,9 ms
RDPG 500x500	0,4 ms	0,4 ms	0,4 ms	3 ms	2 ms
RDPG 1000x1000	0,7 ms	0,7 ms	0,6 ms	5 ms	5 ms

Analizando las tablas 6.11 y 6.12 de liberación de memoria, se puede observar el mismo efecto que las tablas de asignación de memoria, en donde los métodos que utilizan las interfaces de la familia kmalloc son mejores en rendimiento, que las de la familia vmalloc.

6.6.2. TIEMPOS DE OPERACIONES DRIVER MATRIXMODG

Los resultados de las mediciones de tiempo de cada una de las operaciones del driver MatrixmodG con las macros de mensajes de debug habilitadas se pueden observar en las tablas 6.13 y 6.14:

Tabla 6.13: PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER MATRIXMODG (API OPENMP)						
Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG en el kernel	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	1,3 ms	1,5 ms	0,1 ms	3 ms	0,1 ms	0,1 ms
RDPG 100x100	10 ms	11 ms	0,4 ms	130 ms	0,2 ms	0,1 ms
RDPG 500x500	132 ms	133 ms	5 ms	130 ms	0,25 ms	0,5 ms
RDPG 1000x1000	480 ms	508 ms	55 ms	130 ms	0,3 ms	0,7 ms

Tabla 6.14 PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER MATRIXMODG (LIB TIME)						
Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG en el kernel	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,9 ms	1 ms	0,1 ms	1,5 ms	0,1 ms	0,1 ms
RDPG 100x100	9,6 ms	10,2 ms	0,2 ms	22 ms	0,1 ms	0,1 ms
RDPG 500x500	131 ms	132 ms	3 ms	22 ms	0,12 ms	0,5 ms
RDPG 1000x1000	476 ms	504 ms	35 ms	25 ms	0,15 ms	0,7 ms

Los resultados de las tablas 6.13 y 6.14 responde al rendimiento del espacio kernel. Analizando las tablas se puede notar que para la creación de las RDPG en el kernel de Linux mediante el driver MatrixmodG, lo que más trabajo lleva es la operación de asignación de valores de los componentes base de una RDPG, como también la creación completa de una RDPG en el kernel, lo importante es que estas operaciones solo se realizan una única vez para su creación, lo que puede implicar solo un pequeño costo inicial por hacer uso del driver MatrixmodG.

La operación de lectura de estado de la RDPG, está limitada a un número de visualización de elementos fijos como máximo, es por lo cual dicha operación tiene el mismo tiempo para RDPG de 100, 500 y 1000 plazas/transiciones ya que se debe utilizar los comandos de configuración de los atributos posVP, posVT y vdim para visualizar un estado realmente completo en redes de gran tamaño.

Extraer información acerca de las características del objeto RDPG cargado en el kernel es una operación que no demuestra demoras significativas a medida que crece una RDPG, es por lo cual dicha operación es independiente del tamaño de la RDPG.

Por ultimo para una de las operaciones más importantes, como lo es el disparo de una RDPG en el kernel de Linux, se puede notar que a medida que crece la RDPG es mayor el trabajo que debe realizar el kernel, esto es lógico ya que mientras más grande es una RDPG la ecuación de estado debe procesar vectores y matrices de mayor tamaño.

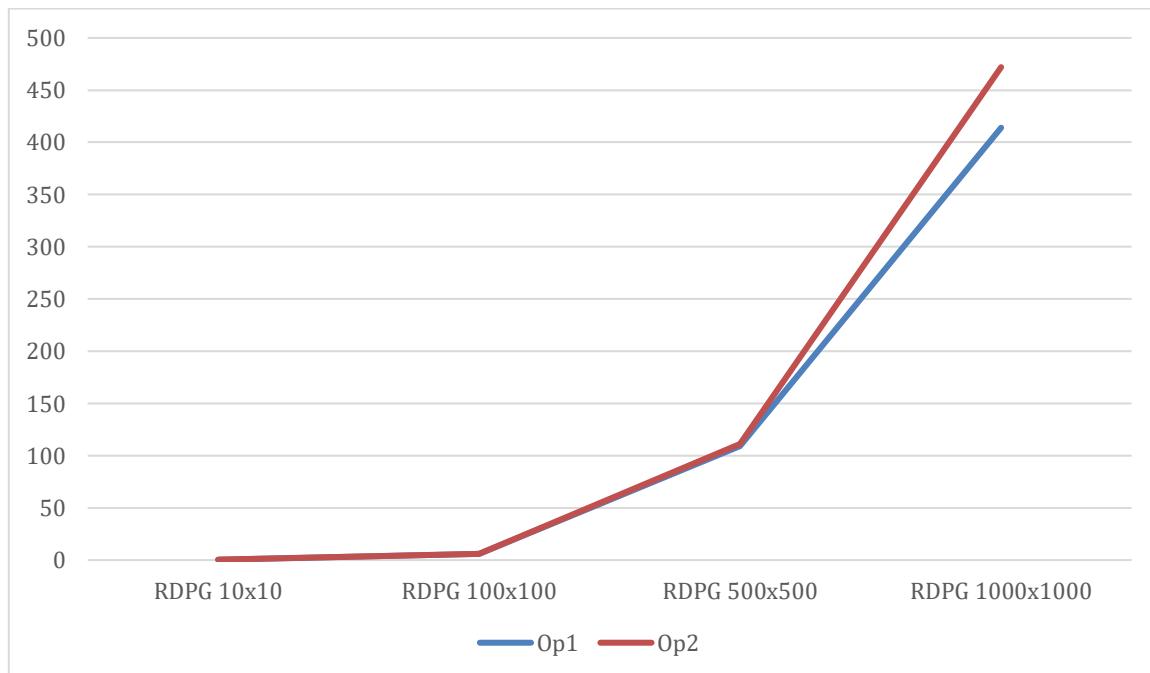
En las tablas 6.15 y 6.16 se observan los mismos resultados, solo que se registran los tiempos que se obtienen cuando se desactivan los mensajes de debug a través del control de las macros asociadas.

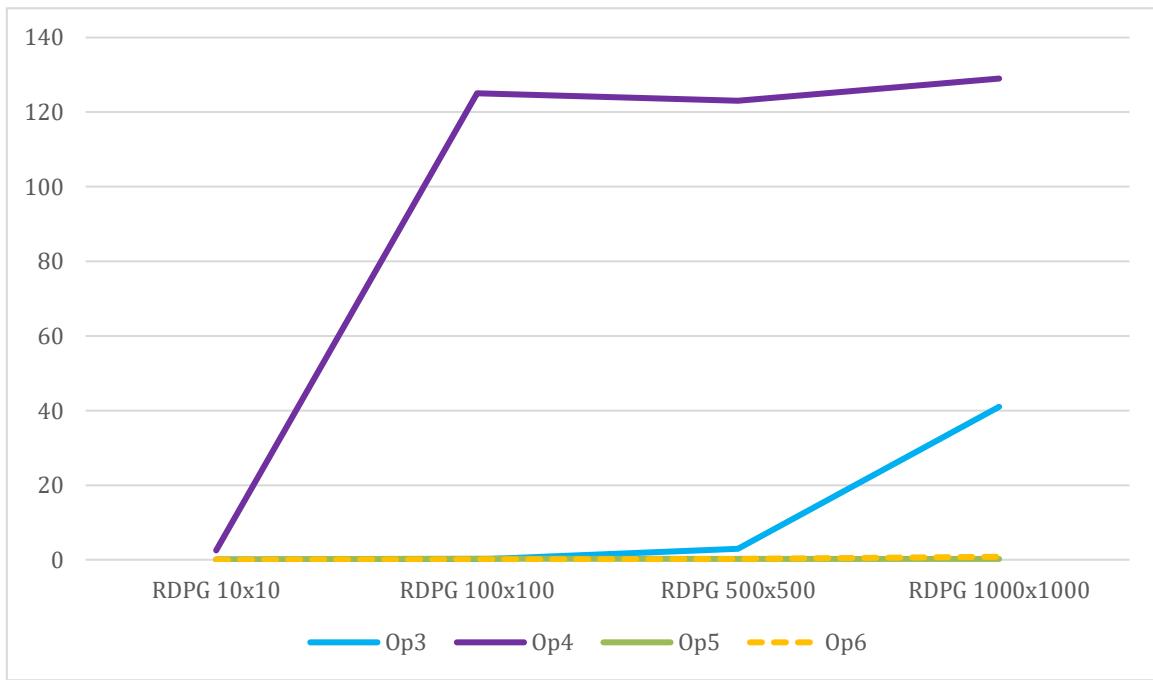
Tabla 6.15: PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER MATRIXMODG (API OPENMP)

Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG en el kernel	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,4 ms	0,4 ms	0,01 ms	2,5 ms	0,05 ms	0,01 ms
RDPG 100x100	5,8 ms	5,9 ms	0,1 ms	125 ms	0,2 ms	0,05 ms
RDPG 500x500	109 ms	111 ms	3 ms	123 ms	0,23 ms	0,27 ms
RDPG 1000x1000	414 ms	472 ms	41 ms	129 ms	0,3 ms	0,8 ms

Tabla 6.16 PROMEDIOS DE TIEMPOS DE OPERACIONES DE DRIVER MATRIXMODG (LIB TIME)

Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG en el kernel	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,35 ms	0,4 ms	0,01 ms	1,4 ms	0,05 ms	0,01 ms
RDPG 100x100	5,7 ms	5,9 ms	0,1 ms	40 ms	0,1 ms	0,05 ms
RDPG 500x500	108 ms	110 ms	3 ms	40 ms	0,12 ms	0,27 ms
RDPG 1000x1000	412 ms	468 ms	38 ms	41 ms	0,15 ms	0,8 ms

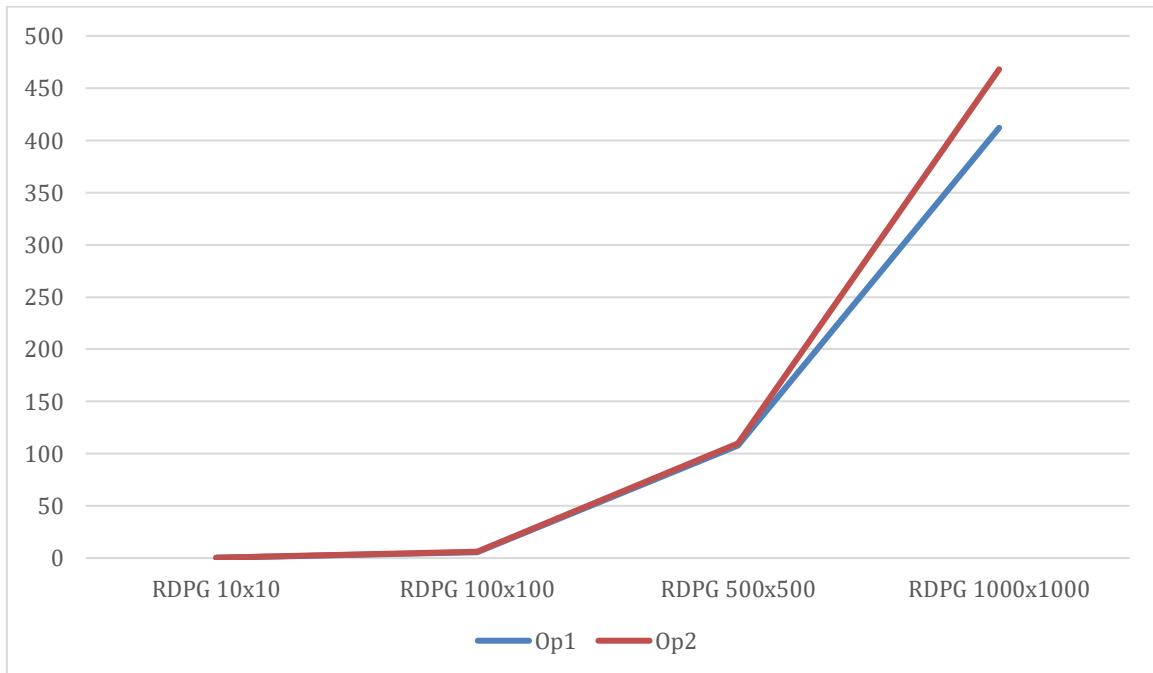


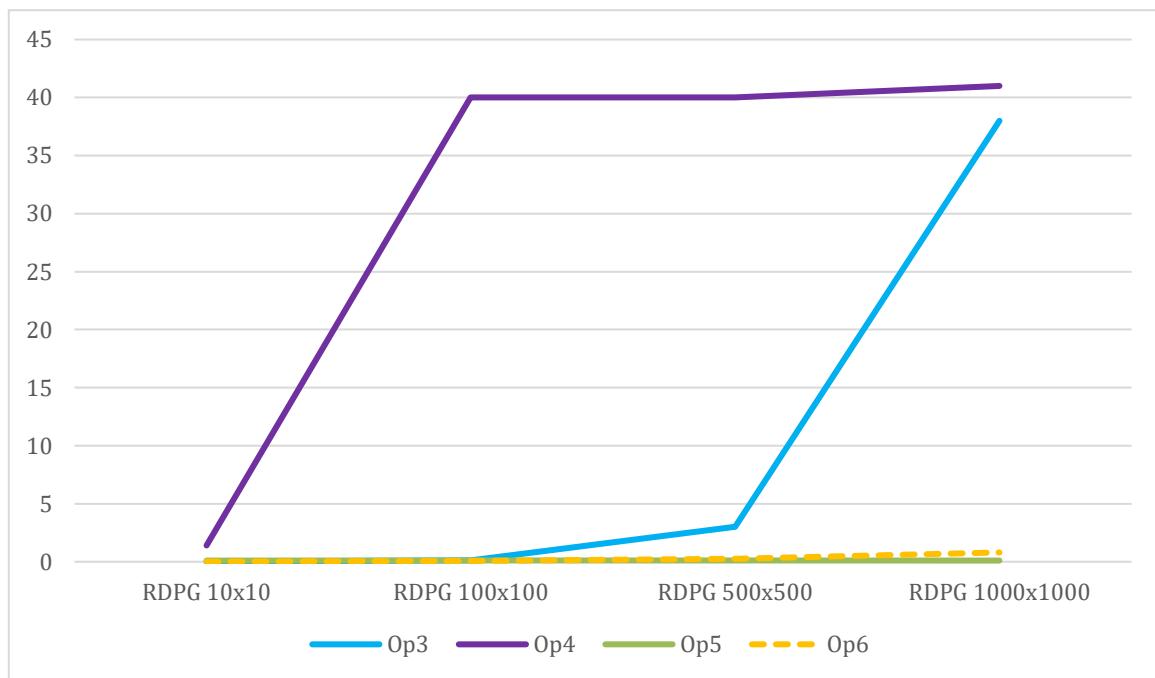


Grafica 6.3 Tiempos de operaciones driver MatrixmodG. (API OpenMP).

Las gráficas asociadas a las tablas 6.15 y 6.16 de los tiempos de operación del driver son la gráfica 6.3 y 6.4 respectivamente. En donde las operaciones del driver se referencian de la siguiente manera:

- **Op1:** Asignación de valores.
- **Op2:** Creación de RDPG en el kernel.
- **Op3:** Disparo de transición de la RDPG.
- **Op4:** Lectura de estado de la RDPG.
- **Op5:** Lectura de información de la RDPG.
- **Op6:** Eliminación de la RDPG.





Grafica 6.4 Tiempos de operaciones driver MatrixmodG. (lib time.h).

En las gráficas 6.3 y 6.4 de los tiempos de las operaciones del driver MatrixmodG, se observa que las operaciones de mayor consumo de tiempo son las asociadas a la creación de una RDPG en el kernel (Op1 y Op2) pero es un pequeño costo que se asume para trabajar con RDPG en el kernel de Linux. Por el lado de las operaciones más importantes a tener en cuenta para el rendimiento del driver son Op3 y Op4, de estas operaciones Op4 es la de mayor costo mientras que Op3 incrementa su tiempo mientras más grande es la RDPG con la que se opera. Las operaciones Op5 y Op6 son las operaciones que menor tiempo de consumo tienen.

6.6.3. TIEMPOS DE OPERACIONES APP C++

Al igual que la sección anterior, en esta sección se analizan los resultados pero para el espacio usuario desde una app C++ con las mismas características que las del driver MatrixmodG. Los resultados se presentan en las tablas 6.17 y 6.18, se tiene en cuenta que las macros de mensajes de debug en el espacio usuario están habilitadas:

Tabla 6.17: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON OPENMP)						
Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,2 ms	0,3 ms	0,05 ms	65 ms	0,1 ms	0,01 ms
RDPG 100x100	6 ms	8 ms	0,7 ms	660 ms	0,2 ms	0,2 ms
RDPG 500x500	108 ms	195 ms	18 ms	575 ms	0,3 ms	2 ms
RDPG 1000x1000	480 ms	675 ms	80 ms	630 ms	0,3 ms	6 ms

Tabla 6.18: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON LIB TIME)

Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,2 ms	0,3 ms	0,02 ms	65 ms	0,1 ms	0,01 ms
RDPG 100x100	5 ms	8 ms	0,7 ms	650 ms	0,2 ms	0,2 ms
RDPG 500x500	108 ms	180 ms	18 ms	575 ms	0,3 ms	2 ms
RDPG 1000x1000	475 ms	665 ms	74 ms	620 ms	0,3 ms	6 ms

Desactivando las macros de mensajes de debug desde el espacio usuario, se observan los resultados de las tablas 6.19 y 6.20:

Tabla 6.19: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON OPENMP)

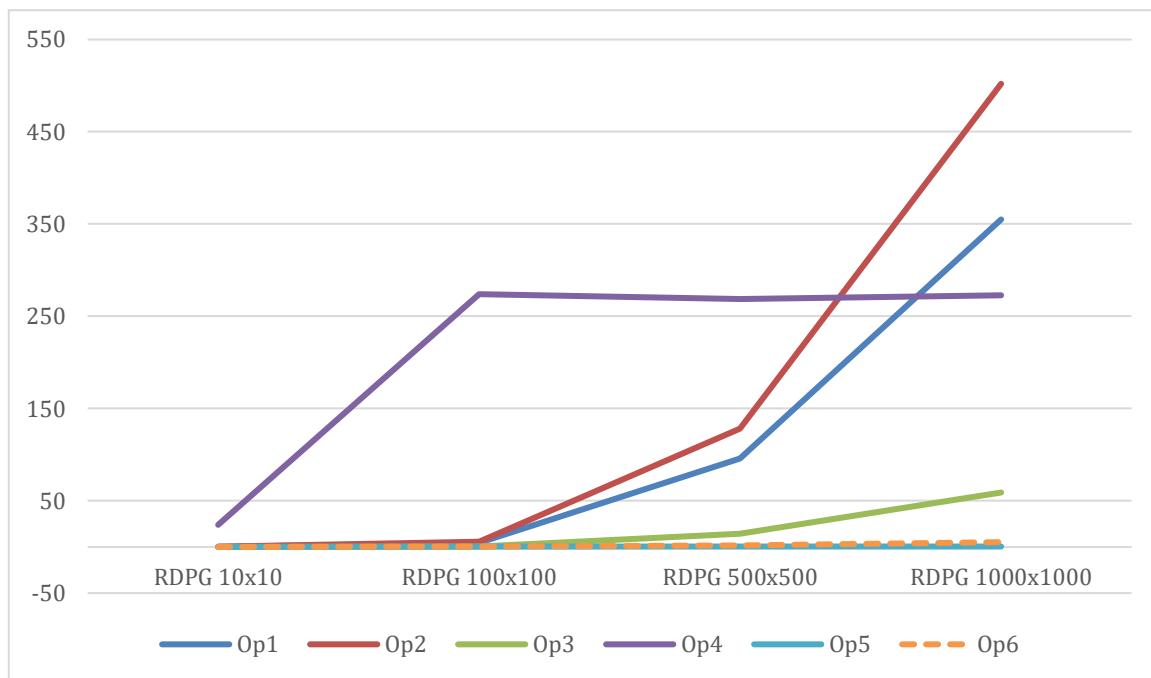
Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,173 ms	0,289 ms	0,01 ms	24 ms	0,04 ms	0,009 ms
RDPG 100x100	4,3 ms	5,6 ms	0,6 ms	274 ms	0,15 ms	0,2 ms
RDPG 500x500	96 ms	128 ms	14,2 ms	269 ms	0,2 ms	1,7 ms
RDPG 1000x1000	355 ms	502 ms	59 ms	273 ms	0,3 ms	5,2 ms

Tabla 6.20: PROMEDIOS DE TIEMPOS DE OPERACIONES DE APP C++ (CON LIB TIME)

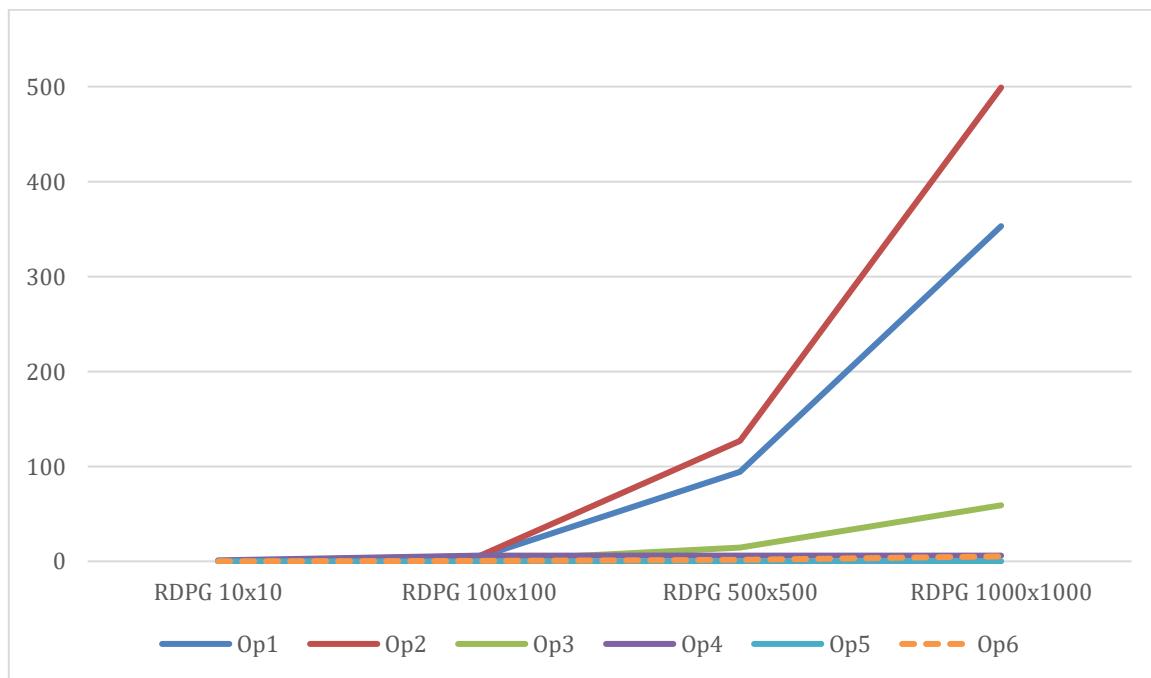
Tiempo de Operación	Asignación de valores (componentes base)	Creación de RDPG	Disparo de transición de RDPG	Lectura de estado de RDPG	Lectura de información de RDPG	Eliminación de RDPG
RDPG 10x10	0,171 ms	0,272 ms	0,01 ms	1 ms	0,04 ms	0,009 ms
RDPG 100x100	4,2 ms	5,4 ms	0,5 ms	6 ms	0,09 ms	0,16 ms
RDPG 500x500	94 ms	127 ms	14,2 ms	6 ms	0,11 ms	1,6 ms
RDPG 1000x1000	353 ms	499 ms	59 ms	6 ms	0,13 ms	5,2 ms

Las gráficas asociadas a las tablas 6.19 y 6.20 de las operaciones de la APP C++ son la gráfica 6.5 y 6.6 respectivamente. Las operaciones se referencian de la misma manera que se realizó con las gráficas 6.4 y 6.5.

Los efectos de las operaciones en la gestión de las RDPG que se notaron en el espacio kernel, también tiene el mismo comportamiento desde el espacio usuario, pero se notan algunos resultados con mayor tiempo de operación desde el espacio usuario, en la siguiente sección se analiza en detalle una comparativa entre cada una de los tiempos de operación desde los diferentes espacios.



Grafica 6.5 Tiempos de operaciones APP C++. (API OpenMp).



Grafica 6.6 Tiempos de operaciones APP C++. (lib time.h).

6.6.4. COMPARACIÓN DRIVER VS APLICACIÓN DE USUARIO

En esta sección se realiza la comparación de los tiempos de las operaciones más importantes para la gestión de RDPG, comparando los tiempos obtenidos por el driver MatrixmodG contra los tiempos obtenidos por la APP C++, determinando la mejora porcentual (o perdida) de rendimiento que produce el driver MatrixmodG.

Se consideran las operaciones más importantes para la gestión de las RDPG las siguientes:

- **Op3:** Disparo de transición de la RDPG.

- **Op4:** Lectura de estado de la RDPG.
- **Op5:** Lectura de información de la RDPG.

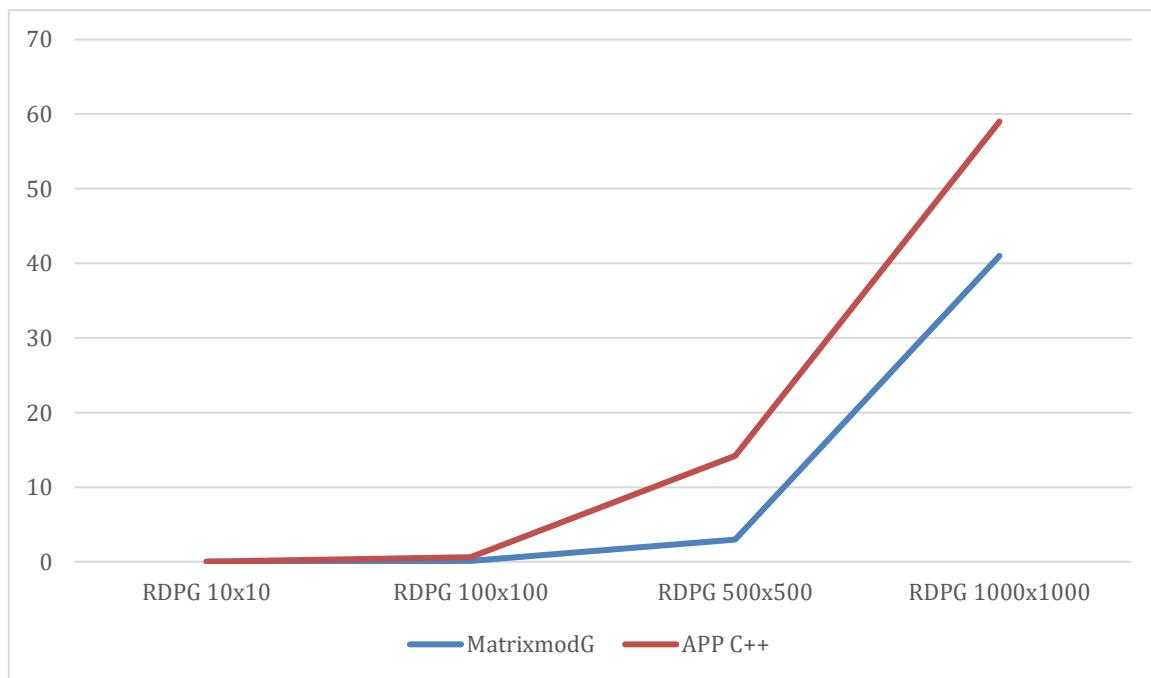
Las comparaciones se realizan en las tablas 6.21, 6.22 y 6.23 haciendo referencia a las operaciones Op3, Op4 y Op5 respectivamente. Se utilizan los tiempos medidos por la API OpenMP.

Tabla 6.21: Comparación Op3			
Tiempo de Operación	Driver MatrixmodG	APP C++	Mejora Porcentual
RDPG 10x10	0,01 ms	0,01 ms	0%
RDPG 100x100	0,1 ms	0,6 ms	80%
RDPG 500x500	3 ms	14,2 ms	75%
RDPG 1000x1000	41 ms	59 ms	30%
Mejora Porcentual promedio			45%

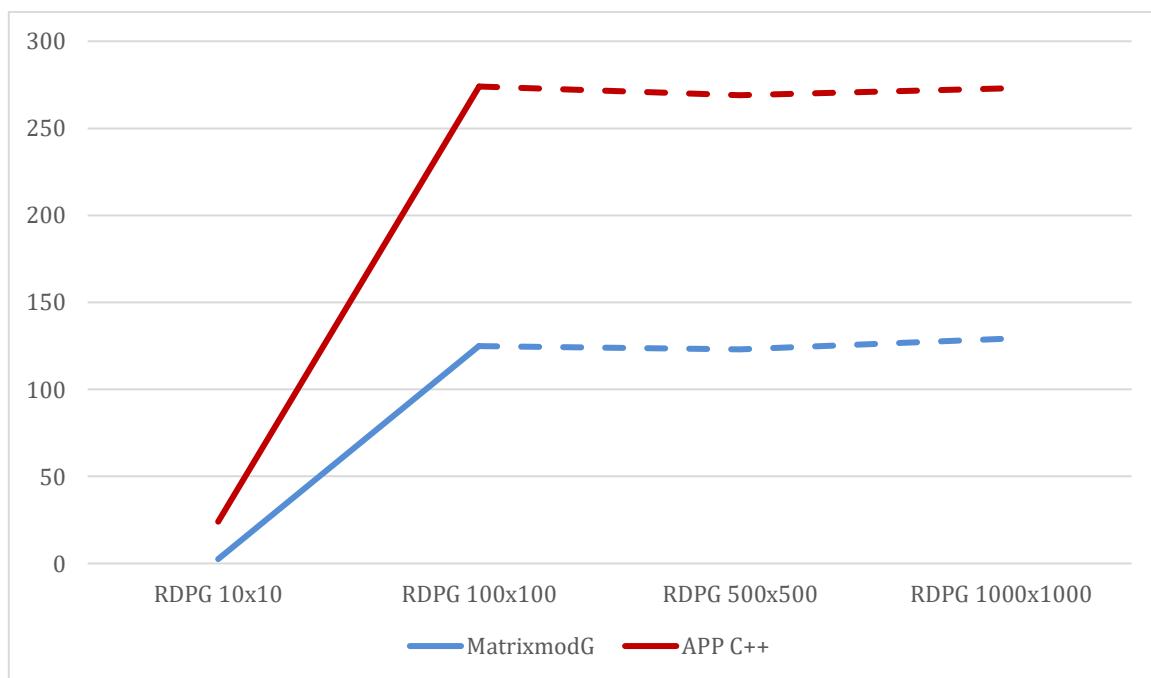
Tabla 6.22: Comparación Op4			
Tiempo de Operación	Driver MatrixmodG	APP C++	Mejora Porcentual
RDPG 10x10	2,5 ms	24 ms	89%
RDPG 100x100	125 ms	274 ms	54%
RDPG 500x500	123 ms	269 ms	54%
RDPG 1000x1000	129 ms	273 ms	52%
Mejora Porcentual promedio			62%

Tabla 6.23: Comparación Op5			
Tiempo de Operación	Driver MatrixmodG	APP C++	Mejora Porcentual
RDPG 10x10	0,05 ms	0,04 ms	0%
RDPG 100x100	0,2 ms	0,15 ms	0%
RDPG 500x500	0,23 ms	0,2 ms	0%
RDPG 1000x1000	0,3 ms	0,3 ms	0%
Mejora Porcentual promedio			0%

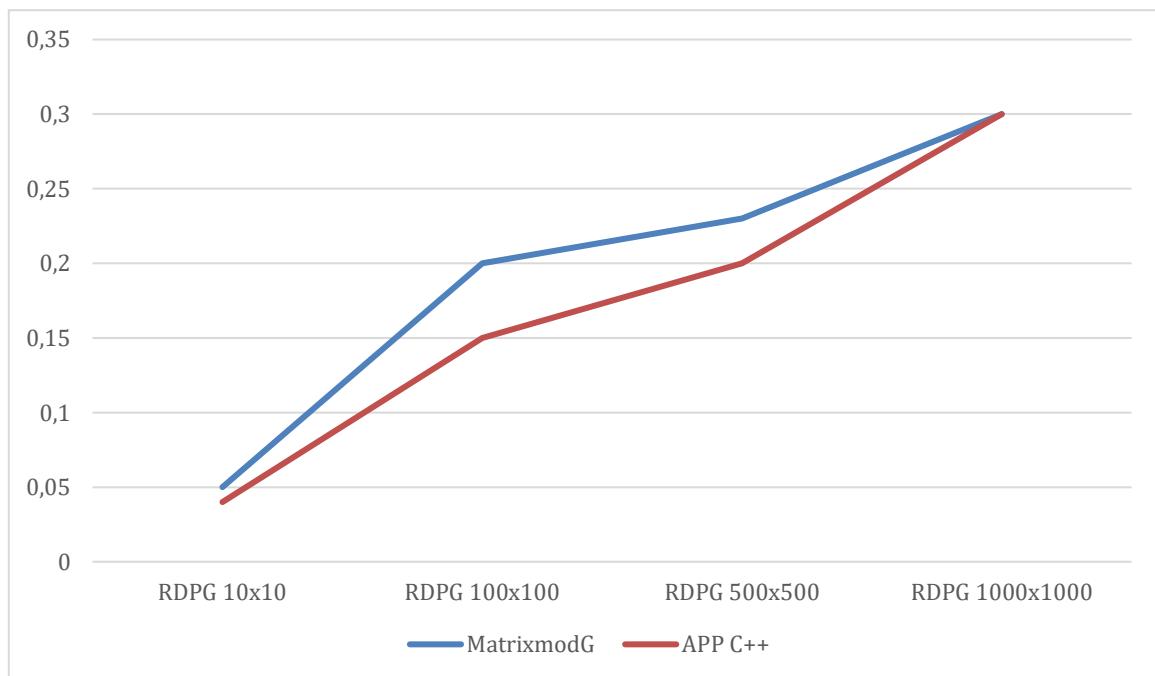
Las gráficas 6.7, 6.8 y 6.9 muestran las mejoras (o pérdidas) de las operaciones Op3, Op4 y Op5 respectivamente, solapando las dos curvas de tiempo para cada una de los sistemas comparados.



Grafica 6.7 Comparación Driver-APP operación de disparo de transición.



Grafica 6.8 Comparación Driver-APP operación de estado completo de RDPG.



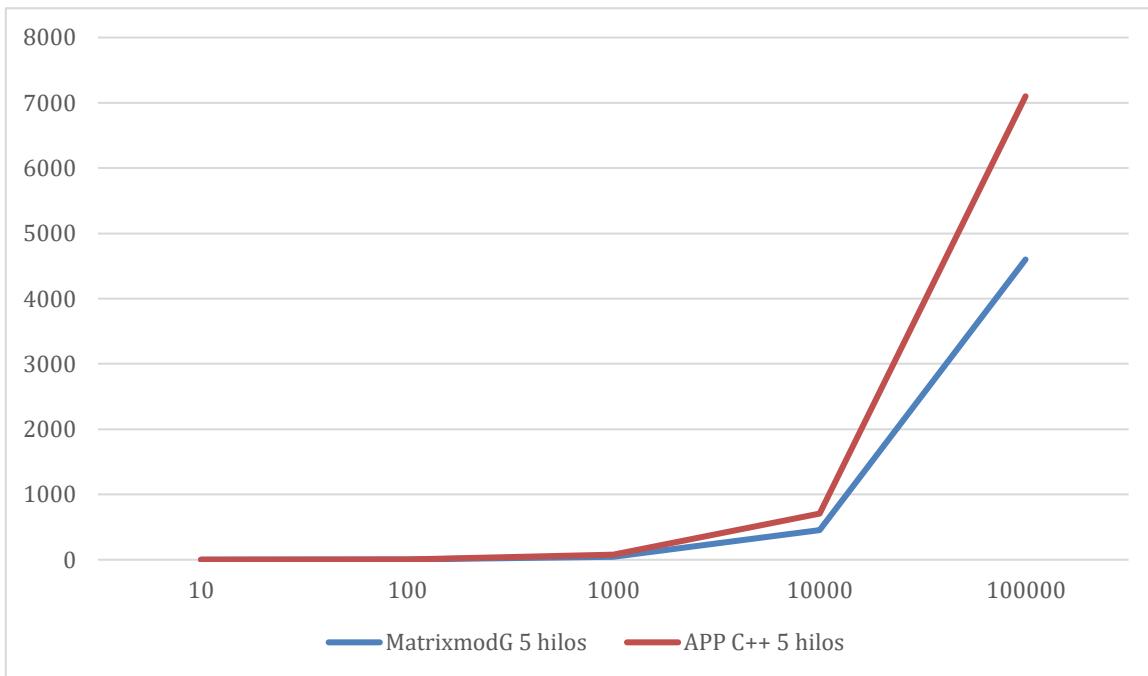
Grafica 6.7 Comparación Driver-APP operación de información RDPG.

6.6.5. TIEMPO CASO DE APLICACIÓN RDPG

De acuerdo al caso de aplicación de la sección 6.4, se realizó un análisis del rendimiento de la gestión de la RDPG del caso de aplicación asociado al procesador dual Core, gestionando la RDPG desde el driver MatrixmodG y desde la APP C++. Como el caso de aplicación gestiona tareas a procesar por los diferentes cores del procesador, se expuso la RDPG de acuerdo al modo de ejecución de la sección 6.4.7 con 5 hilos concurrentes para un conjunto de tareas que inician desde las 10 hasta las 100.000 tareas.

Tabla 6.24: Comparativa de tiempos de gestión de RDPG de Caso de Aplicación 1 con 5 hilos concurrentes.			
Tareas	MatrixmodG 5 hilos	APP C++ 5 hilos	Mejora porcentual
10	0,9 ms	1,1 ms	20%
100	5 ms	8 ms	37%
1000	47 ms	74 ms	36%
10000	455 ms	704 ms	35%
100000	4600 ms	7100 ms	35%
Mejora porcentual promedio			33%

La tabla 6.24 muestra que gestionar RDPG en el driver matrixmodG tiene mucha más rendimiento que realizarlo directamente desde el espacio usuario y mayor es la mejora a medida que más trabajo se realiza sobre la RDPG. Mediante las curvas de la gráfica 6.8 se puede notar dicho efecto.



Grafica 6.8 Comparación rendimiento en gestión de RDPG Driver vs. APP C++.

6.6.6. TIEMPO CASO DE APLICACIÓN RDP

De la misma forma que la sección anterior, y de acuerdo al caso de aplicación 2 de la sección 6.5, se realizó un análisis del rendimiento en la gestión de RDP asociado al problema del productor-consumidor. Como los productores y consumidores tienen trabajo asociado a un buffer controlado, se realiza la gestión de la RDP con dos hilos concurrentes, uno representa el productor y el otro representa el consumidor con una cantidad de trabajo para cada uno que comienza de 10 hasta 100.000.

Tabla 6.25: Comparativa de tiempos de gestión de la RDP del Caso de Aplicación 2 con 2 hilos concurrentes.

Trabajos	MatrixmodG 2 hilos	APP C++ 2 hilos
10	0,4 ms	0,4 ms
100	4 ms	2,5 ms
1000	38 ms	25 ms
10000	355 ms	255 ms
100000	4140 ms	3100 ms

La tabla 6.25 muestra que para el caso de gestionar RDP de pequeño tamaño no es recomendable utilizar el driver MatrixmodG, ya que se obtiene mejor rendimiento desde el espacio usuario.

7. CONCLUSIONES Y TRABAJOS FUTUROS

7.1. CONCLUSIONES

7.1.1. CONCLUSIONES REFERIDAS AL DRIVER MATRIXMODG

El uso de prácticas de patrones de diseño y el paradigma orientado a objetos en el kernel de Linux para el diseño e implementación del driver MatrixmodG fue de gran ayuda. Las prácticas brindaron organización, simplificación y mantenibilidad al código resultante, obteniendo mayor calidad en el producto final. Esta decisión de diseño e implementación también impactó positivamente sobre el proceso de pruebas, simplificando en gran medida la gestión de pruebas unitarias e integrales y permitiendo automatizarlas con frameworks de pruebas como lo son KTF y KEDR.

Quedo demostrado que el driver MatrixmodG sirve como una herramienta totalmente independiente del lenguaje de programación, funcionando como una extensión del kernel. El driver se puede entender como una herramienta del sistema operativo en general, disponible para cualquier aplicación de usuario. La creación de una librería de espacio usuario asociada, posibilita que la capa de aplicación obtenga todo el rendimiento del driver dentro de la programación de un lenguaje particular, quedando demostrado para el lenguaje C++.

Para explotar todos los beneficios del driver MatrixmodG se requiere de un lenguaje de usuario que permita gestionar hilos, haciendo que la gestión del driver sea sometida a la ejecución de múltiples hilos concurrentes, de esta forma se puede obtener eficiencia en la solución de problemas con RDPG. La ejecución a múltiples hilos se comprobó con el uso de la librería del driver en conjunto con monitores, que utilizan mutex y variables de condición, para obtener exclusión mutua entre los hilos y la posibilidad de gestionar los mismos de forma eficiente desde el espacio usuario.

Las operaciones asociadas a la creación de una RDPG en el kernel de Linux son las operaciones de mayor tiempo de ejecución, es un pequeño costo a cambio de trabajar con RDPG en el kernel. Estas operaciones pueden seguir mejorando en futuras versiones, si en lugar de mantener los componentes base de una RDPG en el espacio usuario (enfoque utilizado en lib matrixmodG), se realiza el envío de los datos directamente al ser leídos desde el conjunto de archivos .txt.

Los resultados de rendimiento obtenidos sobre el driver MatrixmodG para gestionar RDPG en el kernel de Linux, demuestran mejoras significativas en comparación a los resultados obtenidos por la aplicación de espacio usuario (aplicación C++). Desde los casos de aplicación ejecutados se pudo verificar que para el caso de aplicación 2, de una RDP pequeña (menos de 10 plazas y transiciones) no se detectan mejoras de rendimiento. Sin embargo para el caso de aplicación 1, de una RDPG mediana (más de 10 plazas y transiciones), se detectan mejoras significativas del 30% en promedio para la operación de disparos de transiciones.

De esta manera se puede concluir que la gestión de Redes de Petri Generalizadas desde el driver MatrixmodG tiene mejor rendimiento que su gestión directa desde una aplicación de usuario y mayor será la mejora mientras más grande sea la red utilizada.

La creación de la librería C++ asociada al driver MatrixmodG, para la gestión de RDPG en el kernel de Linux desde el espacio usuario, demuestra ser una librería de fácil uso, simple, entendible y adaptable a las necesidades que puede requerir un usuario final, realizando todas las operaciones con el driver MatrixmodG de una forma eficiente y totalmente transparente, gracias a la gestión de las RDPG como un objeto de espacio usuario. De esta forma la librería del driver MatrixmodG es una nueva herramienta que permite la solución de problemas computacionales complejos mediante la gestión de RDPG desde programas C++ con soporte al multiprocesamiento y brindando un alto rendimiento.

7.1.2. CONCLUSIONES REFERIDAS AL PROYECTO

Dentro de todas las etapas involucradas para la realización del presente proyecto se concluye sobre algunos aspectos positivos, que sumaron mucho al proyecto realizado, y aspectos negativos que también suman por permitir detectarlos y tener la posibilidad de exponerlos para seguir mejorando los procesos asociados. En ambos casos se consideran los aspectos más importantes como aportes de mejoras para futuros proyectos.

Comenzando con los puntos positivos, el primero que se destaca es de la etapa del plan de pruebas, capítulo 5 del proyecto, en este caso un plan de pruebas que se adapte a un proyecto en particular ayuda en gran medida a orientar que es lo que se debe probar, como probarlo y porque, llevando el proceso de prueba a los límites de acotación necesarios que se deseen para cada proyecto.

Otro punto positivo es el de incentivar el uso de herramientas complementos, como lo son los frameworks, librerías y cualquier otra herramienta que aporte al proyecto en cualquiera de las etapas (diseño, desarrollo, pruebas, etc.). Por ejemplo para el presente proyecto el uso del framework de prueba KTF simplificó el trabajo sobre la automatización de pruebas unitarias e integrales satisfactoriamente y permitió además obtener un producto resultante de mayor calidad.

Por el lado de los aspectos negativos, dentro de la etapa de estudio del sistema, capítulo 3 del proyecto, se realizó la trazabilidad de requerimientos, lo cual fue una herramienta muy útil para mantener una evolución y correlación de los requerimientos ante los cambios, con el uso de la técnica de matrices de trazabilidad, pero demanda mayor tiempo de actualización a medida que el proyecto crece. Como mejora se propone tener la posibilidad de utilizar herramientas de software que permita la gestión de la trazabilidad de los requerimientos con cualquiera de los elementos del proyecto de una manera más flexible y automatizada, permitiendo mantener siempre la trazabilidad actualizada y de gestión ágil, puntos que fueron complejos de llevarlos a cabo con el uso de matrices de trazabilidad durante la evolución del proyecto.

7.2. TRABAJOS FUTUROS

En esta sección se proponen trabajos futuros que se desprenden de las mejoras detectadas del presente proyecto y que no pudieron concretarse en el mismo.

- **Generación de nueva versión de driver MatrixmodG con soporte a RDPG de manera completa.** La versión del driver MatrixmodG desarrollada en el presente proyecto involucra a las RDPG abarcando las extensiones referidas a los arcos inhibidores, lectores y reset. Con respecto a las transiciones se contempla el uso de guardas y semántica temporal desde el espacio usuario, quedando pendiente la extensión referida a los eventos y las semánticas temporales desde el kernel. Para lograr la máxima capacidad de expresión de las RDPG con el driver MatrixmodG se propone una nueva versión que contemple las RDPG de manera completa desde el kernel de Linux.
- **Solución a la disponibilidad de múltiples aplicaciones.** De acuerdo con la sección 4.5, en la gestión de una RDPG con el paradigma de múltiples aplicaciones de lenguajes diferentes, se presenta el problema de que solo los subprocessos, del último lenguaje que peticiona al driver, serán los notificados al cambio del estado global del sistema modelado, ignorándose la notificación de las aplicaciones y subprocessos asociados del resto de lenguajes involucrados a la misma RDPG. Para esto se propone la generación de una nueva versión del driver que disponga de un conjunto de archivos de dispositivo extras asociados a cada aplicación para realizar la notificación a “subprocesos controladores” encargados de actualizar el estado global del sistema en cada aplicación de un lenguaje de manera particular cuando exista un cambio de estado en el sistema simulado mediante la RDPG del kernel.

- **Aumento del número de RDPG paralelas en el kernel.** El driver MatrixmodG permite crear solo una RDPG en el kernel, esta debe eliminarse para crear otra red. Mantener un conjunto de RDPG en el kernel es fácil de realizar con el modelo de código utilizado por el driver MatrixmodG, el mayor trabajo está en la gestión de los comandos para cada RDPG particular. Se propone como mejora una versión del driver en donde se readapte el conjunto de comandos para gestionar diferentes RDPG al mismo tiempo en el kernel de Linux.
- **Medición de tiempos de operación desde el espacio kernel por el propio driver.** Las mediciones de tiempo se realizan todas desde el espacio usuario. Se propone como mejora generar una versión del driver MatrixmodG que realice la medición del tiempo de operación desde el kernel para tener una referencia del rendimiento del driver dentro del espacio kernel.
- **Generación de librerías de espacio usuario sobre otros lenguajes de programación.** Las librerías de espacio usuario creadas para el driver MatrixmodG fueron sobre los lenguajes C y C++. Se propone como trabajo futuro realizar las librerías para otros lenguajes de espacio usuario, analizando con cuál de todos los lenguajes se obtienen los mejores resultados de rendimiento y que diferencias de rendimiento existen en relación a la gestión de RDPG con el driver MatrixmodG.
- **Creación de una interfaz gráfica.** La creación de una interfaz gráfica asociada al driver MatrixmodG permitirá conocer el estado de la RDPG cargada en el kernel de manera gráfica, lo cual es más cómodo para un usuario final. Además la interfaz gráfica con enfoque en la RDPG cargada en el kernel podrá ser controlada desde la ventana que permita su visión formando parte del conjunto de aplicaciones que operan simultáneamente con el DDL MatrixmodG.

8. ANEXOS

8.1. DESARROLLO DE SOFTWARE PROFESIONAL Y PROCESOS DE SOFTWARE

8.1.1. DESARROLLO DE SOFTWARE PROFESIONAL

La ingeniería de software busca apoyar el desarrollo de software profesional. El software profesional, es aquel desarrollo de software destinado a usarse por alguien más aparte de su desarrollador, se lleva a cabo por lo general por equipos, en vez de individualmente. Este tipo de software es mantenido y creado de la mejor forma para que el mismo cambie a lo largo de su vida [21].

Muchos suponen que el software es tan solo otra palabra para los programas de cómputo. No obstante, cuando se habla de ingeniería de software, esto no solo se refiere a los programas en sí, sino también a toda su documentación asociada y los datos de configuración requeridos para hacer que estos programas operen de manera correcta. Un sistema de software desarrollado profesionalmente es más que un solo programa. El sistema por lo regular consta de un número de programas separados y archivos de configuración que se usan para instalar dichos programas. Puede incluir documentación del sistema, que describe la estructura del sistema; documentación del usuario, que explica cómo usar el sistema y sitios web para que los usuarios descarguen información actualizada del producto.

Esta es una de las principales diferencias entre el desarrollo de software profesional y el desarrollo de un aficionado. Es decir que si creamos un software que otros usaran y que otros ingenieros cambiaran, entonces, en general debe ofrecerse información adicional como también el código del programa.

8.1.2. PROCESOS DE SOFTWARE

Un proceso de software es la serie de actividades relacionadas que conducen a la elaboración de un producto de software [21].

Existe una gran variedad de procesos de software diferentes, pero todos deben incluir cuatro actividades que son fundamentales para la ingeniería de software.

- **Especificación del software:** Tienen que definirse tanto la funcionalidad del software como las restricciones de su operación.
- **Diseño e implementación del software:** Debe desarrollarse el software para cumplir con las especificaciones.
- **Validación del software:** Hay que validar el software para asegurarnos de que cumple con lo que un cliente quiere.
- **Evolución del software:** El software tiene que evolucionar para satisfacer las necesidades cambiantes del cliente.

En cierta forma tales actividades forman parte de todos los procesos de software. Estos procesos de software se discuten y describen, por ejemplo, cuando hablamos de actividades como especificar un modelo de datos, diseñar una interfaz de usuario, etc., así como del orden de dichas actividades. Sin embargo, tanto las actividades como las descripciones de los procesos deben incluir:

- a. **Productos**, que son el resultado de una actividad del proceso.
- b. **Roles**, que reflejan las responsabilidades de las personas que intervienen en el proceso.

- c. **Precondiciones y poscondiciones**, qué son declaraciones válidas antes y después de que se realice una actividad del proceso o se cree un producto.

Los procesos de software son complejos y, como todos los procesos intelectuales y creativos, se apoyan en personas con capacidad de juzgar y tomar decisiones. No hay un proceso ideal.

8.1.3. CLASIFICACIÓN DE LOS PROCESOS DE SOFTWARE

En ocasiones, los procesos se clasifican como dirigidos por un plan o como procesos ágiles. Los procesos dirigidos por un plan son aquellos donde todas las actividades del proceso se planean por anticipado y el avance se mide contra dicho plan. En los procesos ágiles la planeación es incremental y es más fácil modificar el proceso para reflejar los requerimientos cambiantes del cliente. Como plantean Boehm y Tuner (2003), cada enfoque es adecuado para diferentes tipos de software [21].

Los enfoques ágiles en el desarrollo de software consideran el diseño y la implementación como las actividades centrales en el proceso del software. Incorporan otras actividades en el diseño y la implementación, como la adquisición de requerimientos y pruebas. En contraste, un enfoque basado en un plan para la ingeniería de software identifica etapas separadas en el proceso de software con salidas asociadas a cada etapa. Las salidas de una etapa se usan como base para planear la siguiente actividad del proceso.

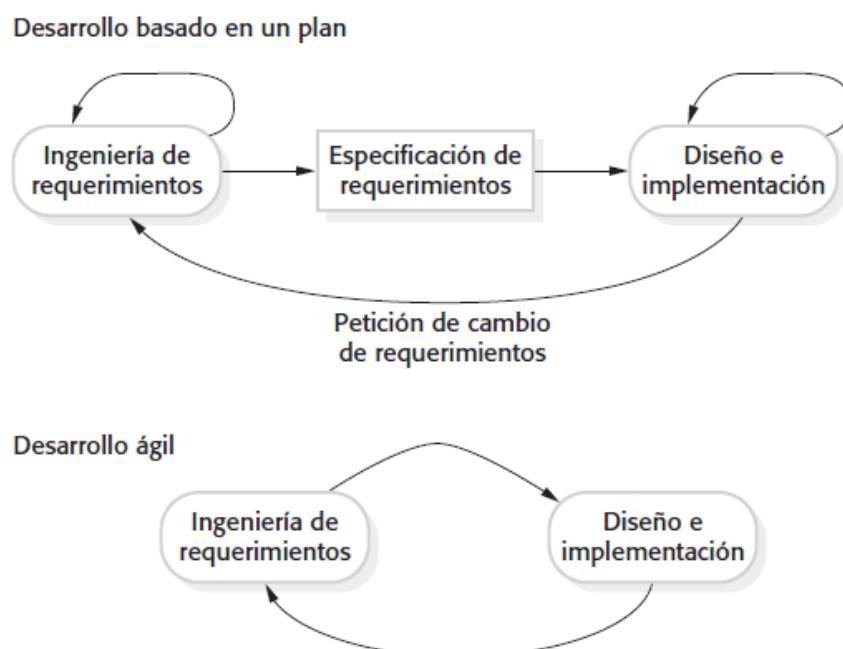


Figura 8.1 Especificación ágil y dirigida por un plan

Analizando las figuras, podemos notar que en un enfoque basado en un plan, la iteración ocurre dentro de las actividades con documentos formales usados para comunicarse entre etapas del proceso. Por ejemplo, los requerimientos evolucionaron y, a final de cuentas, se producirá una especificación de aquellos. Esto entonces es una entrada al proceso de diseño y la implementación. En un enfoque ágil, la iteración ocurre a través de las actividades. Por lo tanto, los requerimientos y el diseño se desarrollan en conjunto, no por separado.

Por lo general, en la mayoría de los proyectos de software se necesita encontrar un equilibrio entre procesos dirigidos por un plan y procesos ágiles. Para decidir sobre el equilibrio entre un enfoque basado en un plan y uno ágil, se deben responder algunas preguntas técnicas, humanas y organizacionales:

- a. ¿Es importante tener una especificación y un diseño muy detallado antes de dirigirse a la implementación?** Si es así, es probable que deba usar un enfoque basado en un plan.

Respuesta:

Si. Para la especificación y el diseño del sistema de software es importante un detalle, con lo cual queremos decir que debemos tener en claro ciertas especificaciones iniciales basadas en los temas a los cuales nos involucramos y como consecuencia afectarán al diseño. En este caso al tratarse de un sistema de software basado en el estudio de un tema en particular, como los son las RDPG, requerimos de los detalles bases del tema antes de dirigirnos a la implementación, esto nos permitirá tener una mayor visibilidad en puntos específicos y de esta manera podremos evitar posibles errores y demoras que podrían surgir en un futuro. Por esta razón la mejor opción en este caso es usar un enfoque basado en un plan para darnos un poco de previsibilidad y por tratarse un trabajo de investigación.

- b. ¿Es práctica una estrategia de entrega incremental, donde se entregue el software a los clientes y se obtenga así una rápida retroalimentación de ellos?** De ser el caso, considere el uso de métodos ágiles.

Respuesta:

Si. Creemos que cuanto más rápida sean las entregas incrementales y por consecuencia una rápida retroalimentación en cuanto a los avances que vayamos teniendo, es sin duda uno de los puntos clave. Por lo cual si buscamos aumentar los tiempos de trabajo eficiente, un enfoque ágil sería una buena idea como metodología de trabajo, para no perder este foco.

- c. ¿Qué tan grande es el sistema o proyecto que se desarrollara?** Los métodos ágiles son más efectivos cuando el sistema logra diseñarse con un pequeño equipo asignado que se comunique de manera informal. Esto sería imposible para los grandes sistemas que precisan equipos de desarrollo más amplios, de manera que tal vez se utilice un enfoque basado en un plan.

Respuesta:

El sistema de software a crear es pequeño a comparación de los grandes sistemas de software de negocios. Es un sistema que apunta a un tema en particular de investigación, que son la RDPG, y eso hace que se acote la cantidad de requerimientos aunque puede llegar a ser un poco complejo al tratarse de un tema de estudio e investigación que seleccionamos como trabajo final. Sobre este punto creemos que un enfoque ágil será una mejor opción.

- d. ¿Qué tipo de sistema se desarrollará?** Los sistemas que demandan mucho análisis antes de la implementación (por ejemplo, sistema en tiempo real con requerimientos de temporización compleja), por lo general, necesitan un diseño bastante detallado para realizar este análisis. En tales circunstancias es recomendable un enfoque basado en un plan.

Respuesta:

El sistema a desarrollar es un driver para un dispositivo (procesador de Redes de Petri) y la interfaz de usuario asociada para su utilización. Se trata de un sistema con complejidad alta ya que se basa en el estudio específico de las RDPG y complementa la investigación del mismo. Debido a esto, es posible que sea mejor opción usar un enfoque basado en un plan, ya que las problemáticas asociadas a las RDPG están bien fundamentadas en sus respectivos informes y documentos a tener en cuenta sobre las especificaciones de requerimientos.

- e. ¿Cuál es el tiempo de vida que se espera del sistema?** Los sistemas con lapsos de vida prolongados podrían requerir más documentación de diseño, para comunicar al equipo de apoyo los propósitos originales de los desarrolladores del sistema. Sin embargo, los defensores de los métodos ágiles fundamentan acertadamente que con frecuencia la documentación no se conserva actualizada, ni se usa mucho para el mantenimiento del sistema de largo plazo.

Respuesta:

Con los conocimientos aprendidos y por la experiencia del director del proyecto, el sistema garantiza 10 años de vida útil como mínimo, ya que si se extiende sus características, proporcionando grandes cambios, de esa

manera se prolonga su vida útil aún más. También al tratarse de un tema de estudio avanzado, su documentación siempre va ser requerida, por lo que para este punto podríamos decir que buscamos que exista la documentación y pensamos que un enfoque basado en un plan será una buena opción.

- f. **¿Qué tecnologías se hallan disponibles para apoyar el desarrollo del sistema?** Los métodos ágiles se auxilian a menudo de buenas herramientas para seguir la pista de un diseño en evolución. Si se desarrolla un sistema con un IDE sin contar con buenas herramientas para visualización y análisis de programas, entonces posiblemente se requiera más documentación de diseño.

Respuesta:

Actualmente se han creado un conjunto de herramientas bastante amplio que ayudan al desarrollo del software profesional por lo que haciendo uso de todas estas herramientas podría disminuir parte de documentaciones en base al desarrollo. En el proyecto se hará uso de la herramienta Trello que nos ayuda a gestionar y controlar el flujo de tareas vinculadas al software, Git como sistema de control de versiones en repositorio local, y Bitbucket como plataforma de desarrollo colaborativo y repositorio remoto. En este punto creemos que un enfoque ágil es una buena opción.

- g. **¿Cómo está organizado el equipo de desarrollo?** Si el equipo de desarrollo está distribuido, o si parte del desarrollo se subcontrata, entonces tal vez se requiera elaborar documentos de diseño para comunicarse a través de los equipos de desarrollo. Quizás se necesite planear por adelantado cuáles son esas partes distribuidas.

Respuesta:

El equipo de desarrollo está organizado por dos personas. Un Director y un alumno desarrollador, el desarrollo del software será realizado completamente por este equipo, de esta forma la comunicación será de manera directa entre cada miembro del equipo y se puede complementar dicha comunicación con las herramientas de desarrollo y trabajo de manera que un enfoque ágil se adapta muy bien en este punto.

- h. **¿Existen problemas culturales que afecten el desarrollo del sistema?** Las organizaciones de ingeniería tradicionales presentan una cultura de desarrollo basada en un plan, pues es una norma de ingeniería. Esto requiere comúnmente una amplia documentación de diseño, en vez del conocimiento informal que se utiliza en los procesos ágiles.

Respuesta:

Al tratarse de un trabajo final debemos cumplir con la documentación asociada a nuestras especificaciones, diseño e implementaciones, validaciones y pruebas por lo que en este punto es de gran ayuda un enfoque basado en un plan.

- i. **¿Qué tan buenos son los diseñadores y programadores en el equipo de desarrollo?** Se argumenta en ocasiones que los métodos ágiles, requieren niveles de habilidad superiores a los enfoques basados en un plan, en que los programadores simplemente traducen un diseño detallado en un código. Si se cuenta con un equipo con niveles de habilidad relativamente bajos, es probable que se necesite el mejor personal para desarrollar el diseño, siendo otros los responsables de la programación.

Respuesta:

El director del proyecto cuenta con suficiente experiencia relacionada al tema específico en que se enfoca el sistema de software y una gran experiencia en cuanto al desarrollo de software. Como estudiante avanzado de la carrera de Ing. en Computaciónuento con varios años sobre los cuales adquirí experiencia tanto en el desarrollo de software como en el conocimiento de las herramientas de desarrollo y las nuevas tecnologías que están marcando nuevas tendencias actualmente. Podríamos decir que somos un equipo de desarrollo con conocimientos avanzados y con bastante experiencia por lo que un enfoque ágil es una buena opción.

- j. **¿El sistema está sujeto a regulaciones externas?** Si un regulador externo tiene que aprobar el sistema, entonces tal vez se requiera documentación detallada como parte del sistema de seguridad.

Respuesta:

El sistema a desarrollar no está sujeto a regulaciones externas por lo cual no será necesario generar documentación extra como parte de regulaciones. De esta manera un enfoque ágil es adecuado gracias a la flexibilidad que existe en el sistema y su libertad en cuanto a regulaciones externas.

Del cuestionario obtuvimos los siguientes resultados en tabla que nos brinda como consecuencia el equilibrio entre procesos asociado al desarrollo del sistema de software del presente proyecto.

Característica del proceso	Proceso ágil	Proceso basado en un plan
a. Especificación y diseño detallado.		✓
b. Entregas incrementales.	✓	
c. Tamaño del sistema a desarrollar.	✓	
d. Tipo de sistema a desarrollar.		✓
e. Tiempo de vida del sistema.		✓
f. Tecnologías disponibles que ayudan al desarrollo.	✓	
g. Organización del equipo de desarrollo.	✓	
h. Problemas culturales que afectan el desarrollo.		✓
i. Experiencia del equipo.	✓	
j. Sistema sujeto a regulación externa.	✓	
Porcentaje	60 %	40 %



Figura 8.2 Porcentajes de inclinación a una metodología ágil y basada en un plan

8.1.4. METODOLOGÍA DE TRABAJO ÁGIL KANBAN

Kanban es un método para gestionar el trabajo intelectual, con énfasis en la entrega justo a tiempo, mientras no se sobrecarguen los miembros del equipo. En este enfoque, el proceso, desde la definición de una tarea hasta su entrega al cliente, se muestra para que los participantes lo vean y los miembros del equipo tomen el trabajo de una cola (backlog) [28].

Kanban se puede dividir en dos partes:

- **Kanban:** Un sistema de gestión de proceso visual que le indica qué producir, cuándo producir, y cuánto producir.
- **El método Kanban:** Una aproximación a la mejora del proceso evolutivo e incremental para las organizaciones.

El método Kanban

En el desarrollo de software, se utiliza el sistema Kanban virtual para limitar el trabajo en curso. A pesar de que el nombre se origina del idioma japonés "Kanban", y se traduce aproximadamente como "tarjeta de señal", y hay tarjetas utilizadas en la mayoría de las implementaciones de Kanban en desarrollo de software, estas tarjetas no funcionan en realidad como señales para realizar más trabajo. Representan los elementos de trabajo. De ahí el término "virtual" porque no existe una tarjeta física [28].

El método Kanban formulado por David J. Anderson es una aproximación al proceso gradual, evolutivo y al cambio de sistemas para las organizaciones. Utiliza un sistema de extracción limitada del trabajo en curso como mecanismo básico para exponer los problemas de funcionamiento del sistema (o proceso) y estimular la colaboración para la mejora continua del sistema. Un ejemplo del sistema de extracción es el sistema Kanban, y es después de esta popular forma de trabajo en curso, que se ha denominado el método.

Los principios del método Kanban

El método Kanban tiene sus raíces en cuatro principios básicos [28]:

- Comience con lo que hace ahora:** El método Kanban se inicia con las funciones y procesos que ya se tienen y estimula cambios continuos, incrementales y evolutivos a su sistema.
- Se acuerda perseguir el cambio incremental y evolutivo:** La organización (o equipo) deben estar de acuerdo que el cambio continuo, gradual y evolutivo es la manera de hacer mejoras en el sistema y debe apegarse a ello. Los cambios radicales pueden parecer más eficaces, pero tienen una mayor tasa de fracaso debido a la resistencia y el miedo en la organización. El método Kanban anima a los pequeños y continuos cambios incrementales y evolutivos a su sistema actual.
- Respetar el proceso actual, los roles, las responsabilidades y los cargos:** Tenemos que facilitar el cambio futuro; acordando respetar los roles actuales, responsabilidades y cargos, eliminamos los temores iniciales. Esto nos debería permitir obtener un mayor apoyo a nuestra iniciativa Kanban.
- Liderazgo en todos los niveles:** Se debe alentar hechos de liderazgo en todos los niveles de la organización de los contribuyentes individuales a la alta dirección.

Características del método Kanban

Anderson identificó cinco características básicas que habían sido observadas en cada implementación correcta del método Kanban. Posteriormente fueron etiquetadas como prácticas y se ampliaron con la adición de una sexta característica [28].

- Visualizar:** Visualizar el flujo de trabajo y hacerlo visible es la base para comprender cómo avanza el trabajo. Sin comprender el flujo de trabajo, realizar los cambios adecuados es más difícil. Una forma común de visualizar el flujo de trabajo es el uso de columnas. Las columnas representan los diferentes estados o pasos en el flujo de trabajo.
- Limitar el trabajo en curso:** Limitar el trabajo en curso implica que un sistema de extracción se aplica en la totalidad o parte del flujo de trabajo. El sistema de extracción actúa como uno de los principales estímulos para los cambios continuos, incrementales y evolutivos en el sistema.

- c. **Dirigir y gestionar el flujo:** Se debe supervisar, medir y reportar el flujo de trabajo a través de cada estado. Al gestionar activamente el flujo, los cambios continuos, graduales y evolutivos del sistema pueden ser evaluados para tener efectos positivos o negativos.
- d. **Hacer las Políticas de Proceso Explícitas:** Configure las reglas y directrices de su trabajo. Entienda las necesidades y asegúrese de seguir las reglas. Las políticas definirán cuándo y por qué una tarjeta debe pasar de una columna a otra. Escríbalas. Cambie las reglas cuando la realidad cambie.
- e. **Utilizar modelos para reconocer oportunidades de mejora:** Cuando los equipos tienen un entendimiento común de las teorías sobre el trabajo, el flujo de trabajo, el proceso y el riesgo, es más probable que sea capaz de construir una comprensión compartida de un problema y proponer acciones de mejora que puedan ser aprobadas por consenso. El método Kanban sugiere que un enfoque científico sea utilizado para implementar los cambios continuos, graduales y evolutivos. El método no prescribe un método científico específico para utilizarlo.

8.1.5. DESARROLLO ITERATIVO INCREMENTAL

El desarrollo iterativo incremental es un proceso de desarrollo de software creado en respuesta a las debilidades del modelo tradicional de cascada.

Básicamente este modelo de desarrollo, que no es más que un conjunto de tareas agrupadas en pequeñas etapas repetitivas (iteraciones), es uno de los más utilizados en los últimos tiempos ya que, como se relaciona con novedosas estrategias de desarrollo de software y una programación extrema, es empleado en diversas metodologías [29].

El modelo consta de diversas etapas de desarrollo en cada incremento, las cuales inician con el análisis y finalizan con la instauración y aprobación del sistema [29].

Concepto de desarrollo iterativo incremental

Se planifica un proyecto en distintos bloques temporales que se denominan iteración. En una iteración se repite un determinado proceso de trabajo que brinda un resultado más completo para un producto final, de forma que quien lo utilice reciba beneficios de este proyecto de manera creciente.

Para llegar a lograr esto, cada requerimiento debe tener un completo desarrollo en una única iteración que debe de incluir pruebas y una documentación para que el equipo pueda cumplir con todos los objetivos que sean necesarios y esté listo para ser dado al cliente. Así se evita tener arriesgadas actividades en el proyecto finalizado.

Lo que se busca es que en cada iteración los componentes logren evolucionar el producto dependiendo de los completados de las iteraciones antecesoras, agregando más opciones de requerimientos y logrando así una mejora más completa.

Una manera muy primordial para dirigir al proceso iterativo incremental es la de priorizar los objetivos y requerimientos en función del valor que ofrece el cliente [30].

Ciclo de vida

La idea principal detrás de la mejora iterativa es desarrollar un sistema de programas de manera incremental, permitiéndole al desarrollador sacar ventaja de lo que se ha aprendido a lo largo del desarrollo anterior, incrementando, versiones entregables del sistema. El aprendizaje viene de dos vertientes: el desarrollo del sistema, y su uso (mientras sea posible). Los pasos claves en el proceso son comenzar con una implementación simple de los requerimientos del sistema, e iterativamente mejorar la secuencia evolutiva de versiones hasta que

el sistema completo esté implementado. En cada iteración, se realizan cambios en el diseño y se agregan nuevas funcionalidades y capacidades al sistema.

Básicamente este modelo se basa en dos premisas:

- Los usuarios nunca saben bien que es lo que necesitan para satisfacer sus necesidades. En el caso del driver MatrixmodG se tienen objetivos claros, pero existe flexibilidad para realizar cambios que permitan la evolución del sistema.
- En el desarrollo, los procesos tienden a cambiar. En el driver constantemente se busca optimizar las funciones en busca de mejoras de rendimiento.

El proceso en sí mismo consiste de las siguientes etapas:

- **Etapa de inicialización:** Se crea una versión del sistema. La meta de esta etapa es crear un producto con el que el usuario pueda interactuar, y por ende retroalimentar el proceso. Debe ofrecer una muestra de los aspectos claves del problema y proveer una solución lo suficientemente simple para ser comprendida e implementada fácilmente. Para guiar el proceso de iteración se crea una lista de control de proyecto, que contiene un historial de todas las tareas que necesitan ser realizadas. Incluye cosas como nuevas funcionalidades para ser implementadas, y áreas de rediseño de la solución ya existente. Esta lista de control se revisa periódica y constantemente como resultado de la fase de análisis.
- **Etapa de iteración:** Esta etapa involucra el rediseño e implementación de una tarea de la lista de control de proyecto, y el análisis de la versión más reciente del sistema. La meta del diseño e implementación de cualquier iteración es ser simple, directa y modular, para poder soportar el rediseño de la etapa o como una tarea añadida a la lista de control de proyecto. El código puede, en ciertos casos, representar la mayor fuente de documentación del sistema. El análisis de una iteración se basa en la retroalimentación del usuario y en el análisis de las funcionalidades disponibles del programa. Involucra el análisis de la estructura, modularidad, usabilidad, confiabilidad, eficiencia y eficacia (alcanzar las metas). La lista de control del proyecto se modifica bajo la luz de los resultados del análisis.
- **Lista de control de proyecto.**

Características

Usando análisis y mediciones como guías para el proceso de mejora es una diferencia mayor entre las mejoras iterativas y el desarrollo rápido de aplicaciones, principalmente por dos razones:

- Provee de soporte para determinar la efectividad de los procesos y de la calidad del producto.
- Permite estudiar y después mejorar y ajustar el proceso para el ambiente en particular.

Estas mediciones y actividades de análisis pueden ser añadidas a los métodos de desarrollo rápido existentes.

De hecho, el contexto de iteraciones múltiples conlleva ventajas en el uso de mediciones. Las medidas a veces son difíciles de comprender en lo absoluto, aunque en los cambios relativos en las medidas a través de la evolución del sistema puede ser muy informativo porque proveen una base de comparación. Por ejemplo, un vector de medidas m_1, m_2, \dots, m_n puede ser definido para caracterizar varios aspectos del producto en cierto punto, como pueden ser el esfuerzo total realizado, los cambios, los defectos, los atributos lógico, físico y dinámico, consideraciones del entorno, etcétera. Así el observador puede decir como las características del producto como el tamaño, la complejidad, el acoplamiento y la cohesión incrementan o disminuyen en el tiempo. También puede monitorearse el cambio relativo de varios aspectos de un producto o pueden proveer los límites de las medidas para apuntar a problemas potenciales y anomalías.

Ventajas del desarrollo iterativo incremental

- En este modelo los usuarios no tienen que esperar hasta que el sistema completo se entregue para hacer uso de él. El primer incremento cumple los requerimientos más importantes de tal forma que pueden utilizar el software al instante.
- Los usuarios pueden utilizar los incrementos iniciales como prototipos y obtener experiencia sobre los requerimientos de los incrementos posteriores del sistema.
- Existe muy pocas probabilidades de riesgo en el sistema. Aunque se pueden encontrar problemas en algunos incrementos, lo normal es que el sistema se entregue sin inconvenientes al usuario.
- Ya que los sistemas de más alta prioridad se entregan primero, y los incrementos posteriores se integran entre ellos, es muy probable que los sistemas más importantes sean a los que se les hagan más pruebas. Esto quiere decir que es menos probable que los usuarios encuentren fallas de funcionamiento del software en las partes más importantes del sistema [21].
- En el desarrollo de este modelo se da la retroalimentación muy temprano a los usuarios.
- Permite separar la complejidad del proyecto, gracias a su desarrollo por parte de cada iteración o bloque.
- El producto es consistente y puntual en el desarrollo.
- Los productos desarrollados con este modelo tienen una menor probabilidad de fallar.
- Se obtiene un aprendizaje en cada iteración que es aplicado en el desarrollo del producto y aumenta las experiencias para próximos proyectos [29].

8.2. TRAZABILIDAD DE REQUERIMIENTOS

La trazabilidad de requerimientos es la asociación de un requerimiento con otros requerimientos y las diferentes instancias o artefactos con que se relaciona, se refiere a la "capacidad para describir y seguir la vida de un requerimiento en ambas direcciones, hacia delante (forward) y hacia atrás (backward) esto es, desde su origen, durante su desarrollo y especificación, hasta su desarrollo y uso, y a lo largo de todos los períodos de refinamiento en curso e iteración en alguna de estas fases". La trazabilidad está condicionada por los cambios y las validaciones que los participantes del proyecto hagan al sistema durante el proceso de desarrollo [31].

Es necesario seleccionar aquellas asociaciones que son de interés relevante para el análisis, para su posterior análisis ante un posible cambio en los elementos que se puedan ver afectados. Debido a esto, resulta fundamental que la trazabilidad siempre esté actualizada y refleje la realidad del proyecto en tiempo real [32].

Disponer de una buena trazabilidad de requerimientos nos permite realizar el control y apoyo para la toma de decisiones en el proyecto. Por ejemplo, una de las ventajas principales que nos ofrece la trazabilidad es poder determinar si todos los requerimientos han sido considerados y si las instancias que han sido generadas pueden asociarse con un requerimiento válido.

Gracias a la trazabilidad de requerimientos tenemos la posibilidad de identificar el origen de cada requerimiento y realizar el seguimiento de cada cambio que se realice sobre el mismo. Además, al trazar los requerimientos con otros artefactos como pruebas, casos de uso, código, etc., será posible responder a los cambios de forma más controlada y con más información, y en consecuencia anticiparnos a lo que un cambio puede significar.

Se han creado diferentes técnicas y modelos para soportar la práctica de trazabilidad durante el proceso de desarrollo de software. La técnica más común y aplicable a cualquier modelo de desarrollo es la construcción de matrices de trazabilidad, que hacen posible el análisis de la correlación entre elementos de una misma etapa y entre etapas en diferentes niveles de abstracción. Por ejemplo, en la figura 8.1 se muestra una de las matrices más comunes de trazado. Esta es la matriz de Requerimientos y Casos de Uso, la cual permite verificar en qué casos de uso son representados y especificados los requerimientos funcionales.

		Casos de uso			
		CU1	CU2	CU3	CU4
Requerimientos	RF1	✓			✓
	RF2		✓		
	RF3				
	...				
	RFn	✓			

Figura 8.3 Matriz de trazabilidad de Requerimientos vs. Casos de uso.

La construcción de estas matrices trae beneficios, más allá de un simple registro de la correlación o dependencia entre los elementos de las etapas. A partir de ellas es posible analizar características tales como el nivel de especificación de los requerimientos, el nivel de participación de los usuarios, el costo asociado a cada fase de desarrollo, la arquitectura requerida, el plan de las pruebas, etc. Otra de las ventajas de estas matrices es que, si se actualizan continuamente, pueden servir como herramienta de soporte muy útil para el personal encargado del mantenimiento y las pruebas del sistema.

Esta práctica puede ser costosa y tediosa para sistemas de gran tamaño o complejos. Algunas herramientas posibilitan la generación automática de estas matrices o de forma automática generan elementos de una etapa a otra en diferentes niveles de abstracción (p. ej. la etapa de análisis a la etapa de diseño), para obtener así una correlación de trazado; sin embargo, no todos los elementos pueden ser correlacionados de forma automática o simplemente no tienen elementos definidos en las etapas destino.

8.3. DISEÑO ARQUITECTÓNICO

El diseño arquitectónico se interesa por entender cómo debe organizarse un sistema y cómo tiene que diseñarse la estructura global de ese sistema. En el modelo del proceso de desarrollo de software, el diseño arquitectónico es la primera etapa en el proceso de diseño del software. Es el enlace crucial entre el diseño y la ingeniería de requerimientos, ya que identifica los principales componentes estructurales en un sistema y la relación entre ellos. La salida del proceso de diseño arquitectónico consiste en un modelo arquitectónico que describe la forma en que se organiza el sistema como un conjunto de componentes en comunicación [21].

En los procesos ágiles, por lo general se acepta que una de las primeras etapas en el proceso de desarrollo debe preocuparse por establecer una arquitectura global del sistema. Usualmente no resulta exitoso el desarrollo incremental de arquitecturas. Mientras que la refactorización de componentes en respuesta a los cambios suele ser relativamente fácil, tal vez resulte costoso refactorizar una arquitectura de sistema.

Para ayudar a comprender lo que se entiende por arquitectura del sistema, tome en cuenta la figura 8.4. En ella se presenta un modelo abstracto de la arquitectura para un sistema de robot de empaquetado, que indica los componentes que tienen que desarrollarse.

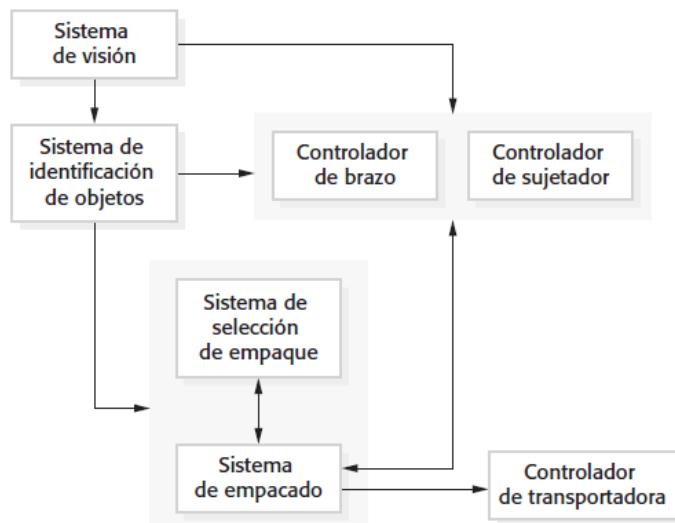


Figura 8.4 Arquitectura de un sistema de control para un robot.

En la práctica, hay un significativo traslape entre los procesos de ingeniería de requerimientos y el diseño arquitectónico. De manera ideal, una especificación de sistema no debe incluir cierta información de diseño. Esto no es realista, excepto para sistemas muy pequeños. La descomposición arquitectónica es por lo general necesaria para estructurar y organizar la especificación. Por lo tanto, como parte del proceso de ingeniería de requerimientos, usted podría proponer una arquitectura de sistema abstracta donde se asocien grupos de funciones de sistemas o características con componentes o subsistemas a gran escala. Luego, puede usar esta descomposición para discutir con los participantes sobre los requerimientos y las características del sistema.

Las arquitecturas de software se diseñan en dos niveles de abstracción, que en este texto se llaman arquitectura en pequeño y arquitectura en grande:

- La arquitectura en pequeño se interesa por la arquitectura de programas individuales. En este nivel, uno se preocupa por la forma en que el programa individual se separa en componentes.
- La arquitectura en grande se interesa por la arquitectura de sistemas empresariales complejos que incluyen otros sistemas, programas y componentes de programa. Tales sistemas empresariales se distribuyen a través de diferentes computadoras, que diferentes compañías administran y poseen. En los capítulos 18 y 19 se cubren las arquitecturas grandes; en ellos se estudiarán las arquitecturas de los sistemas distribuidos.

La arquitectura de software es importante porque afecta el desempeño y la potencia, así como la capacidad de distribución y mantenimiento de un sistema (Bosch, 2000). Como afirma Bosch, los componentes individuales implementan los requerimientos funcionales del sistema. Los requerimientos no funcionales dependen de la arquitectura del sistema, es decir, la forma en que dichos componentes se organizan y comunican. En muchos sistemas, los requerimientos no funcionales están también influidos por componentes individuales, pero no hay duda de que la arquitectura del sistema es la influencia dominante. Bass y sus colaboradores (2003) analizan tres ventajas de diseñar y documentar de manera explícita la arquitectura de software:

- **Comunicación con los participantes:** La arquitectura es una presentación de alto nivel del sistema, que puede usarse como un enfoque para la discusión de un amplio número de participantes.
- **Análisis del sistema:** En una etapa temprana en el desarrollo del sistema, aclarar la arquitectura del sistema requiere cierto análisis. Las decisiones de diseño arquitectónico tienen un efecto profundo sobre si el sistema puede o no cubrir requerimientos críticos como rendimiento, fiabilidad y mantenibilidad.
- **Reutilización a gran escala:** Un modelo de una arquitectura de sistema es una descripción corta y manejable de cómo se organiza un sistema y cómo interoperan sus componentes. Por lo general, la arquitectura del sistema es la misma para sistemas con requerimientos similares y, por lo tanto, puede soportar reutilización de software a gran escala.

Hofmeister y sus colaboradores (2000) proponen que una arquitectura de software sirve en primer lugar como un plan de diseño para la negociación de requerimientos de sistema y, en segundo lugar, como un medio para establecer discusiones con clientes, desarrolladores y administradores. También sugieren que es una herramienta esencial para la administración de la complejidad. Oculta los detalles y permite a los diseñadores enfocarse en las abstracciones clave del sistema.

Las arquitecturas de sistemas se modelan con frecuencia usando diagramas de bloques simples, como en la figura 8.4. Cada recuadro en el diagrama representa un componente. Los recuadros dentro de recuadros indican que el componente se dividió en subcomponentes. Las flechas significan que los datos y/o señales de control pasan de un componente a otro en la dirección de las flechas.

Los diagramas de bloque presentan una imagen de alto nivel de la estructura del sistema e incluyen fácilmente a individuos de diferentes disciplinas que intervienen en el proceso de desarrollo del sistema.

Hay dos formas en que se utiliza un modelo arquitectónico de un programa:

- **Como una forma de facilitar la discusión acerca del diseño del sistema:** Una visión arquitectónica de alto nivel de un sistema es útil para la comunicación con los participantes de un sistema y la planeación del proyecto, ya que no se satura con detalles. Los participantes pueden relacionarse con él y entender una visión abstracta del sistema. En tal caso, analizan el sistema como un todo sin confundirse por los detalles. El modelo arquitectónico identifica los componentes clave que se desarrollarán, de modo que los administradores pueden asignar a individuos para planear el desarrollo de dichos sistemas.
- **Como una forma de documentar una arquitectura que se haya diseñado:** La meta aquí es producir un modelo de sistema completo que muestre los diferentes componentes en un sistema, sus interfaces y conexiones. El argumento para esto es que tal descripción arquitectónica detallada facilita la comprensión y la evolución del sistema.

Los diagramas de bloque son una forma adecuada para describir la arquitectura del sistema durante el proceso de diseño, pues son una buena manera de soportar las comunicaciones entre las personas involucradas en el proceso. En muchos proyectos, suele ser la única documentación arquitectónica que existe.

8.3.1. DECISIONES EN EL DISEÑO ARQUITECTÓNICO

El diseño arquitectónico es un proceso creativo en el cual se diseña una organización del sistema que cubrirá los requerimientos funcionales y no funcionales de éste. Puesto que se trata de un proceso creativo, las actividades dentro del proceso dependen del tipo de sistema que se va a desarrollar, los antecedentes y la experiencia del arquitecto del sistema, así como de los requerimientos específicos del sistema. Por lo tanto, es útil pensar en el diseño arquitectónico como un conjunto de decisiones a tomar en vez de una secuencia de actividades [21].

Durante el proceso de diseño arquitectónico, los arquitectos del sistema deben tomar algunas decisiones estructurales que afectarán profundamente el sistema y su proceso de desarrollo.

Aunque cada sistema de software es único, los sistemas en el mismo dominio de aplicación tienen normalmente arquitecturas similares que reflejan los conceptos fundamentales del dominio. Cuando se diseña una arquitectura de sistema, debe decidirse qué tienen en común el sistema y las clases de aplicación más amplias, con la finalidad de determinar cuánto conocimiento se pueden reutilizar de dichas arquitecturas de aplicación.

La arquitectura de un sistema de software puede basarse en un patrón o un estilo arquitectónico particular.

- Un patrón arquitectónico es una descripción de una organización del sistema (Garlan y Shaw, 1993), tal como una organización cliente-servidor o una arquitectura por capas. Los patrones arquitectónicos captan

la esencia de una arquitectura que se usó en diferentes sistemas de software. Usted tiene que conocer tanto los patrones comunes, en que éstos se usen, como sus fortalezas y debilidades cuando se tomen decisiones sobre la arquitectura de un sistema.

- Estilo y patrón llegaron a significar lo mismo. Es necesario elegir la estructura más adecuada, como cliente-servidor o estructura en capas, que le permita satisfacer los requerimientos del sistema. Para descomponer las unidades del sistema estructural, usted opta por la estrategia de separar los componentes en subcomponentes. Los enfoques que pueden usarse permiten la implementación de diferentes tipos de arquitectura. Finalmente, en el proceso de modelado de control, se toman decisiones sobre cómo se controla la ejecución de componentes. Se desarrolla un modelo general de las relaciones de control entre las diferentes partes del sistema.

Debido a la estrecha relación entre los requerimientos no funcionales y la arquitectura de software, el estilo y la estructura arquitectónicos particulares que se elijan para un sistema dependerán de los requerimientos de sistema no funcionales, como:

- Rendimiento.
- Seguridad.
- Protección.
- Disponibilidad.
- Mantenibilidad.

Evaluar un diseño arquitectónico es difícil porque la verdadera prueba de una arquitectura es qué tan bien el sistema cubre sus requerimientos funcionales y no funcionales cuando está en uso. Sin embargo, es posible hacer cierta evaluación al comparar el diseño contra arquitecturas de referencia o patrones arquitectónicos genéricos.

8.3.2. VISTAS ARQUITECTÓNICAS

Es imposible representar toda la información relevante sobre la arquitectura de un sistema en un solo modelo arquitectónico, ya que cada uno presenta únicamente una vista o perspectiva del sistema. Ésta puede mostrar cómo un sistema se descompone en módulos, cómo interactúan los procesos de tiempo de operación o las diferentes formas en que los componentes del sistema se distribuyen a través de una red. Todo ello es útil en diferentes momentos de manera que, para el diseño y la documentación, por lo general se necesita presentar múltiples vistas de la arquitectura de software [21].

Existen diferentes opiniones relativas a qué vistas se requieren. Krutchen (1995), en su bien conocido modelo de vista 4+1 de la arquitectura de software, sugiere que deben existir cuatro vistas arquitectónicas fundamentales, que se relacionan usando casos de uso o escenarios. Las vistas que él sugiere son:

- Una **vista lógica**, que indique las abstracciones clave en el sistema como objetos o clases de objeto. En este tipo de vista se tienen que relacionar los requerimientos del sistema con entidades.
- Una **vista de proceso**, que muestre cómo, en el tiempo de operación, el sistema está compuesto de procesos en interacción. Esta vista es útil para hacer juicios acerca de las características no funcionales del sistema, como el rendimiento y la disponibilidad.
- Una **vista de desarrollo**, que muestre cómo el software está descompuesto para su desarrollo, esto es, indica la descomposición del software en elementos que se implementen mediante un solo desarrollador o equipo de desarrollo. Esta vista es útil para administradores y programadores de software.
- Una **vista física**, que exponga el hardware del sistema y cómo los componentes de software se distribuyen a través de los procesadores en el sistema. Esta vista es útil para los ingenieros de sistemas que planean una implementación de sistema.

8.3.3. ARQUITECTURA EN CAPAS

El patrón de arquitectura en capas es otra forma de lograr separación e independencia. Aquí, la funcionalidad del sistema está organizada en capas separadas, y cada una se apoya sólo en las facilidades y los servicios ofrecidos por la capa inmediatamente debajo de ella.

Este enfoque en capas soporta el desarrollo incremental de sistemas. Conforme se desarrolla una capa, algunos de los servicios proporcionados por esta capa deben quedar a disposición de los usuarios. La arquitectura también es cambiante y portátil. En tanto su interfaz no varíe, una capa puede sustituirse por otra equivalente. Más aún, cuando las interfaces de capa cambian o se agregan nuevas facilidades a una capa, sólo resulta afectada la capa adyacente. A medida que los sistemas en capas localizan dependencias de máquina en capas más internas, se facilita el ofrecimiento de implementaciones multiplataforma de un sistema de aplicación. Sólo las capas más internas dependientes de la máquina deben reimplantarse para considerar las facilidades de un sistema operativo o base de datos diferentes.

La figura 8.5 es un ejemplo de una arquitectura en capas con cuatro capas. La capa inferior incluye software de soporte al sistema, por lo general soporte de base de datos y sistema operativo. La siguiente capa es la de aplicación, que comprende los componentes relacionados con la funcionalidad de la aplicación, así como los componentes de utilidad que usan otros componentes de aplicación. La tercera capa se relaciona con la gestión de interfaz del usuario y con brindar autenticación y autorización al usuario, mientras que la capa superior proporciona facilidades de interfaz de usuario. Desde luego, es arbitrario el número de capas. Cualquiera de las capas en la figura 8.5 podría dividirse en dos o más capas.

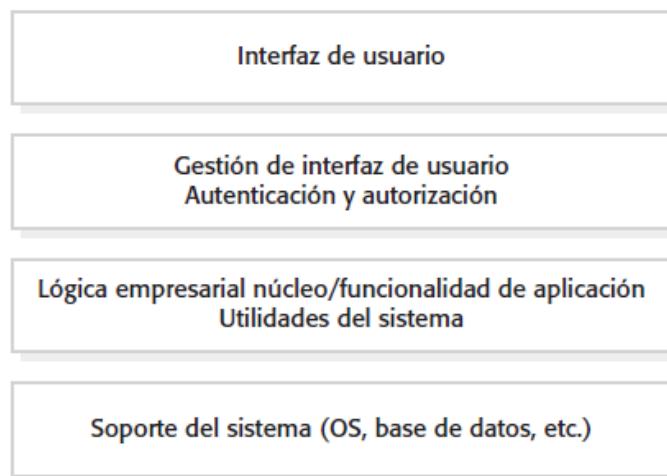


Figura 8.5 Arquitectura genérica en capas.

8.4. GESTIÓN DE RIESGOS EN PROYECTOS DE SOFTWARE

La gestión del riesgo es una de las tareas más sustanciales para un administrador de proyecto. La gestión del riesgo implica anticipar riesgos que pudieran alterar el calendario del proyecto o la calidad del software a entregar, y posteriormente tomar acciones para evitar dichos riesgos. Podemos considerar un riesgo como algo que es preferible que no ocurra [21].

La gestión del riesgo es particularmente importante para los proyectos de software, debido a la incertidumbre inherente que enfrentan la mayoría de proyectos. Ésta se deriva de requerimientos vagamente definidos, cambios de requerimientos que obedecen a cambios en las necesidades del cliente, dificultades en estimar el tiempo y los recursos requeridos para el desarrollo de software, o bien, se deriva de diferencias en las habilidades

individuales. Es necesario anticipar los riesgos; comprender el efecto de estos riesgos sobre el proyecto, el producto y la empresa; y dar los pasos adecuados para evitar dichos riesgos. Tal vez se necesite diseñar planes de contingencia de manera que, si ocurren los riesgos, se puedan tomar acciones inmediatas de recuperación.

8.4.1. TIPOS DE RIESGOS

Los riesgos pueden amenazar el proyecto, el software que se desarrolla o a la organización. Por lo tanto, existen tres categorías relacionadas de riesgo:

- **Riesgos del proyecto:** Los riesgos que alteran el calendario o los recursos del proyecto. Un ejemplo de riesgo de proyecto es la renuncia de un diseñador experimentado. Encontrar un diseñador de reemplazo con habilidades y experiencia adecuadas puede demorar mucho tiempo y, en consecuencia, el diseño del software tardará más tiempo en completarse. Estos están relacionados con:
 - Presupuesto: se necesita una mayor inversión.
 - Planificación: se necesita más tiempo.
 - Personal: se necesitan más o mejores cualificados.
 - Recursos: se necesitan más o mejores instrumentos.
 - Requerimientos: se necesitan más condiciones.
- **Riesgos del producto:** Los riesgos que afectan la calidad o el rendimiento del software a desarrollar. Un ejemplo de riesgo de producto es la falla que presenta un componente que se adquirió al no desempeñarse como se esperaba. Esto puede afectar el rendimiento global del sistema, de modo que es más lento de lo previsto. Estos están relacionados con:
 - Requerimientos: se necesitan más condiciones.
 - Diseño: se necesita una mejor arquitectura.
 - Implementación: se necesita más tiempo o personal para finalizarse.
 - Interfaz: se necesita un mejor estudio de usabilidad e interacción.
 - Verificación: se necesita realizar más pruebas.
 - Mantenimiento: se necesita más dedicación después de la entrega final.
 - Incertidumbre técnica: se necesita mayor conocimiento sobre el área.
 - Tecnologías desconocidas: se necesita mayor conocimiento sobre cómo realizar el proyecto de forma óptima.
- **Riesgos empresariales:** Riesgos que afectan a la organización que desarrolla o adquiere el software. Por ejemplo, un competidor que introduce un nuevo producto es un riesgo empresarial. La introducción de un producto competitivo puede significar que las suposiciones hechas sobre las ventas de los productos de software existentes sean excesivamente optimistas. Los principales riesgos empresariales son:
 - Riesgo de mercado.
 - Riesgo estratégico.
 - Riesgo de ventas.
 - Riesgo de dirección.
 - Riesgo de presupuesto.

Desde luego, estos tipos de riesgos se traslanan. Si un programador experimentado abandona un proyecto, esto puede ser un riesgo del proyecto porque, incluso si se sustituye de manera inmediata, el calendario se alterará. Siempre se requiere tiempo para que un nuevo miembro del proyecto comprenda el trabajo realizado, de manera que no puede ser inmediatamente productivo. En consecuencia, la entrega del sistema podría demorarse. La salida de un miembro del equipo también puede ser un riesgo del producto, porque un sustituto tal vez no sea

tan experimentado y, por lo tanto, podría cometer errores de programación. Finalmente, puede ser un riesgo empresarial, porque la experiencia de dicho programador es vital para obtener nuevos contratos.

Es necesario registrar los resultados del análisis del riesgo en el plan del proyecto, junto con un análisis de consecuencias, que establece las consecuencias del riesgo para el proyecto, el producto y la empresa. La gestión de riesgos efectiva facilita hacer frente a los problemas y asegurar que éstos no conduzcan a un presupuesto inaceptable o a retrasos en el calendario.

Los riesgos específicos que podrían afectar un proyecto dependen del proyecto y el entorno de la organización donde se desarrolla el software. Sin embargo, también existen riesgos comunes que no se relacionan con el tipo de software a desarrollar y que pueden ocurrir en cualquier proyecto. En la siguiente tabla se muestran algunos de estos riesgos comunes.

Riesgo	Repercute en	Descripción
Rotación de personal	Proyecto	Personal experimentado abandonará el proyecto antes de que éste se termine.
Cambio administrativo	Proyecto	Habrá un cambio de gestión en la organización con diferentes prioridades.
Indisponibilidad de hardware	Proyecto	Hardware, que es esencial para el proyecto, no se entregará a tiempo.
Cambio de requerimientos	Proyecto y producto	Habrá mayor cantidad de cambios a los requerimientos que los anticipados.
Demoras en la especificación	Proyecto y producto	Especificaciones de interfaces esenciales no están disponibles a tiempo.
Subestimación del tamaño	Proyecto y producto	Se subestimó el tamaño del sistema.
Cambio tecnológico	Empresa	La tecnología subyacente sobre la cual se construye el sistema se sustituye con nueva tecnología.
Competencia de productos	Empresa	Un producto competitivo se comercializa antes de que el sistema esté completo.

Los objetivos de la gestión de riesgos son identificar, dirigir y eliminar las fuentes de riesgo antes de que empiecen a afectar a la finalización satisfactoria de un proyecto software [33].

El riesgo siempre implica dos características:

- **Incertidumbre:** el acontecimiento que caracteriza al riesgo puede o no puede ocurrir.
- **Pérdida:** si el riesgo se convierte en una realidad, ocurrirán consecuencias no deseadas o pérdidas.

Para cuantificar el nivel de incertidumbre y el grado de pérdidas asociado con cada riesgo se consideran las diferentes categorías de riesgos vistas anteriormente. Se puede hacer otra categorización de los riesgos en función de su facilidad de detección:

- **Riesgos conocidos:** son aquellos que se pueden predecir después de una evaluación del plan del proyecto, del entorno técnico y otras fuentes de información fiables.
- **Riesgos predecibles:** se extrapolan de la experiencia de proyectos anteriores.
- **Riesgos impredecibles:** pueden ocurrir, pero es extremadamente difícil identificarlos por adelantado.

La gestión continuada de los riesgos permite aumentar su eficiencia:

- Evaluar continuamente lo que pueda ir mal.
- Determinar qué riesgos son importantes o implementar estrategias para resolverlos.
- Asegurar la eficacia de las estrategias.

8.4.2. PROCESO DE GESTIÓN DE RIESGOS

En la figura 8.6 se ilustra una idea general del proceso de gestión del riesgo. Comprende varias etapas:

- **Identificación del riesgo:** Hay que identificar posibles riesgos para el proyecto, el producto y la empresa.
- **Análisis de riesgos:** Se debe valorar la probabilidad y las consecuencias de dichos riesgos.
- **Planeación del riesgo:** Es indispensable elaborar planes para enfrentar el riesgo, evitarlo o minimizar sus efectos en el proyecto.
- **Monitorización del riesgo:** Hay que valorar regularmente el riesgo y los planes para atenuarlo, y revisarlos cuando se aprenda más sobre el riesgo.

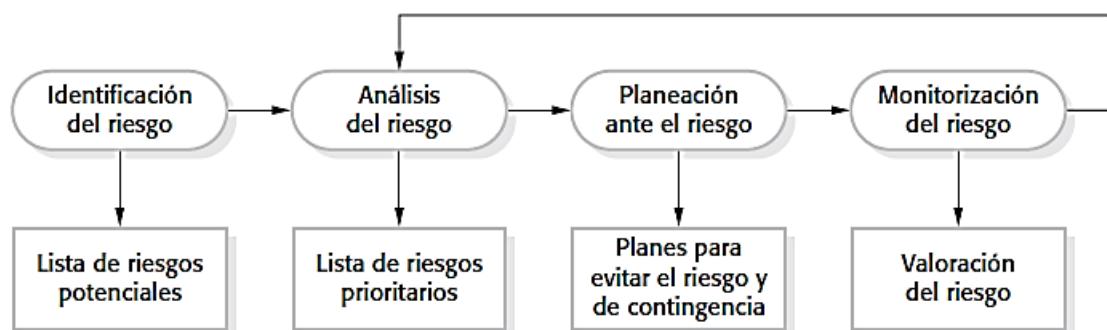


Figura 8.6 Proceso de gestión de riesgos.

Es preciso documentar los resultados del proceso de gestión del riesgo en un plan de gestión del riesgo. Éste debe incluir un estudio de los riesgos que enfrenta el proyecto, un análisis de dichos riesgos e información de cómo se gestionará el riesgo cuando es probable que se convierta en un problema.

El proceso de gestión del riesgo es un proceso iterativo que continúa a lo largo del proyecto. Una vez desarrollado un plan de gestión del riesgo inicial, se monitoriza la situación para detectar riesgos emergentes. Conforme está disponible más información referente a los riesgos, habrá que volver a analizar los riesgos y decidir si cambió la prioridad del riesgo. Entonces tal vez sea necesario cambiar los planes para evitar el riesgo y gestionar la contingencia.

Identificación del riesgo

La identificación del riesgo es la primera etapa del proceso de gestión del riesgo. Se ocupa de identificar los riesgos que pudieran plantear una mayor amenaza al proceso de ingeniería de software, al software a desarrollar, o a la organización que lo desarrolla. La identificación del riesgo puede ser un proceso de equipo en el que este último se reúne para pensar en posibles riesgos. O bien, el administrador del proyecto, con base en su experiencia, identifica los riesgos más probables o críticos.

Como punto de partida para la identificación del riesgo, se recomienda utilizar una lista de verificación de diferentes tipos de riesgo. Existen al menos seis tipos de riesgos que pueden incluirse en una lista de verificación:

- **Riesgos tecnológicos:** Se derivan de las tecnologías de software o hardware usadas para desarrollar el sistema.
- **Riesgos personales:** Se asocian con las personas en el equipo de desarrollo.
- **Riesgos organizacionales:** Se derivan del entorno organizacional donde se desarrolla el software.
- **Riesgos de herramientas:** Resultan de las herramientas de software y otro software de soporte que se usa para desarrollar el sistema.

- **Riesgos de requerimientos:** Proceden de cambios a los requerimientos del cliente y del proceso de gestionarlos.
- **Riesgos de estimación:** Surgen de las estimaciones administrativas de los recursos requeridos para construir el sistema.

Al concluir el proceso de identificación de riesgos, se tendrá una larga lista de eventualidades que podrían ocurrir y afectar al producto, al proceso y a la empresa. Entonces se necesita reducir esta lista a un tamaño razonable. Si existen demasiados riesgos, será prácticamente imposible seguir la huella de todos ellos.

Análisis de riesgo

Durante el proceso de análisis de riesgos, hay que considerar cada riesgo identificado y realizar un juicio acerca de la probabilidad y gravedad de dicho riesgo. No hay una forma sencilla de hacer esto. Usted debe apoyarse en su propio juicio y en la experiencia obtenida en los proyectos anteriores y los problemas que surgieron en ellos. No es posible hacer valoraciones precisas y numéricas de la probabilidad y gravedad de cada riesgo. En vez de ello, habrá que asignar el riesgo a una de ciertas bandas:

1. La probabilidad del riesgo puede valorarse como muy baja (< 10%), baja (del 10 al 25%), moderada (del 25 al 50%), alta (del 50 al 75%) o muy alta (> 75%).
2. Los efectos del riesgo pueden estimarse como catastróficos (amenazan la supervivencia del proyecto), graves (causarían grandes demoras), tolerables (demoras dentro de la contingencia permitida) o insignificantes.

Luego hay que tabular los resultados de este proceso de análisis mediante una tabla clasificada de acuerdo con la gravedad del riesgo. Desde luego, aquí la valoración de la probabilidad y seriedad son arbitrarias. Para hacer esta valoración, se necesita información detallada del proyecto, el proceso, el equipo de desarrollo y la organización.

Tanto la probabilidad como la valoración de los efectos de un riesgo pueden cambiar conforme se disponga de más información acerca del riesgo y a medida que se implementen planes de gestión del riesgo. Por lo tanto, esta tabla se debe actualizar durante cada iteración del proceso de riesgo.

Una vez analizados y clasificados los riesgos, valore cuáles son los más significativos. Su juicio debe depender de una combinación de la probabilidad de que el riesgo surja junto con los efectos de dicho riesgo. En general, los riesgos catastróficos deben considerarse siempre, así como los riesgos graves con más de una probabilidad moderada de ocurrencia.

El número correcto de riesgos a monitorizar debe depender del proyecto. Pueden ser cinco o 15. Sin embargo, el número de riesgos elegidos para monitorizar debe ser manejable. Un número de riesgos muy grande requeriría recopilar demasiada información.

Planeación del riesgo

El proceso de planeación del riesgo considera cada uno de los riesgos clave identificados y desarrolla estrategias para manejarlos. Para cada uno de los riesgos, usted debe considerar las acciones que puede tomar para minimizar la perturbación del proyecto si se produce el problema identificado en el riesgo. También debe pensar en la información que tal vez necesite recopilar mientras observa el proyecto para que pueda anticipar los problemas.

Nuevamente, no hay un proceso simple que pueda seguirse para la planeación de contingencias. Se apoya en el juicio y la experiencia del administrador del proyecto. Las estrategias se establecen en tres categorías:

- **Estrategias de evitación:** Seguir estas estrategias significa que se reducirá la probabilidad de que surja el riesgo.
- **Estrategias de minimización:** Seguir estas estrategias significa que se reducirá el efecto del riesgo.
- **Planes de contingencia:** Seguir estas estrategias significa que se está preparado para lo peor y se tiene una estrategia para hacer frente a ello.

Aquí se observa una clara analogía con las estrategias utilizadas en los sistemas críticos para garantizar fiabilidad, seguridad y protección, cuando hay que evitar, tolerar o recuperarse de las fallas. Desde luego, es mejor usar una estrategia que evitar el riesgo. Si esto no es posible, se debe usar una estrategia que reduzca las posibilidades de que el riesgo cause graves efectos. Finalmente, se debe contar con estrategias para enfrentar el riesgo cuando éste surja. Tales estrategias deben reducir el efecto global de un riesgo en el proyecto o el producto.

Monitorización del riesgo

La monitorización del riesgo es el proceso para comprobar que no han cambiado sus suposiciones sobre riesgos del producto, el proceso y la empresa. Hay que valorar regularmente cada uno de los riesgos identificados para decidir si este riesgo se vuelve más o menos probable. También se tiene que considerar si los efectos del riesgo han cambiado o no. Para hacer esto, observe otros factores, como el número de peticiones de cambio de requerimientos, lo que da pistas acerca de la probabilidad del riesgo y sus efectos. Dichos factores dependen claramente de los tipos de riesgos.

Los riesgos deben monitorizarse comúnmente en todas las etapas del proyecto. En cada revisión administrativa, es necesario reflexionar y estudiar cada uno de los riesgos clave por separado. También hay que decidir si es más o menos probable que surja el riesgo, y si cambiaron la gravedad y las consecuencias del riesgo.

8.5. MATRICES Y VECTORES DINÁMICOS

El lenguaje de programación C proporciona la posibilidad de manejar matrices en forma estática por lo que se debe conocer el tamaño de la matriz en tiempo de compilación. En caso de matrices de dos dimensiones, el compilador debe conocer el número de filas y columnas. Si el tamaño de la matriz no se conoce hasta el tiempo de ejecución el lenguaje de programación C nos permite manipular matrices en forma dinámica [34].

En particular, se posibilitará:

- Crear matrices en forma dinámica (en tiempo de ejecución).
- Eliminar matrices en forma dinámica.
- Tener acceso a la matriz mediante índices o punteros.

Para poder trabajar la matriz en forma dinámica se requiere una variable del tipo doble puntero. El primer puntero hará referencia al inicio de un arreglo de punteros y cada puntero del arreglo de punteros tendrá la referencia del inicio de un arreglo de enteros u otro tipo de variable, la figura 8.7 muestra esta idea.

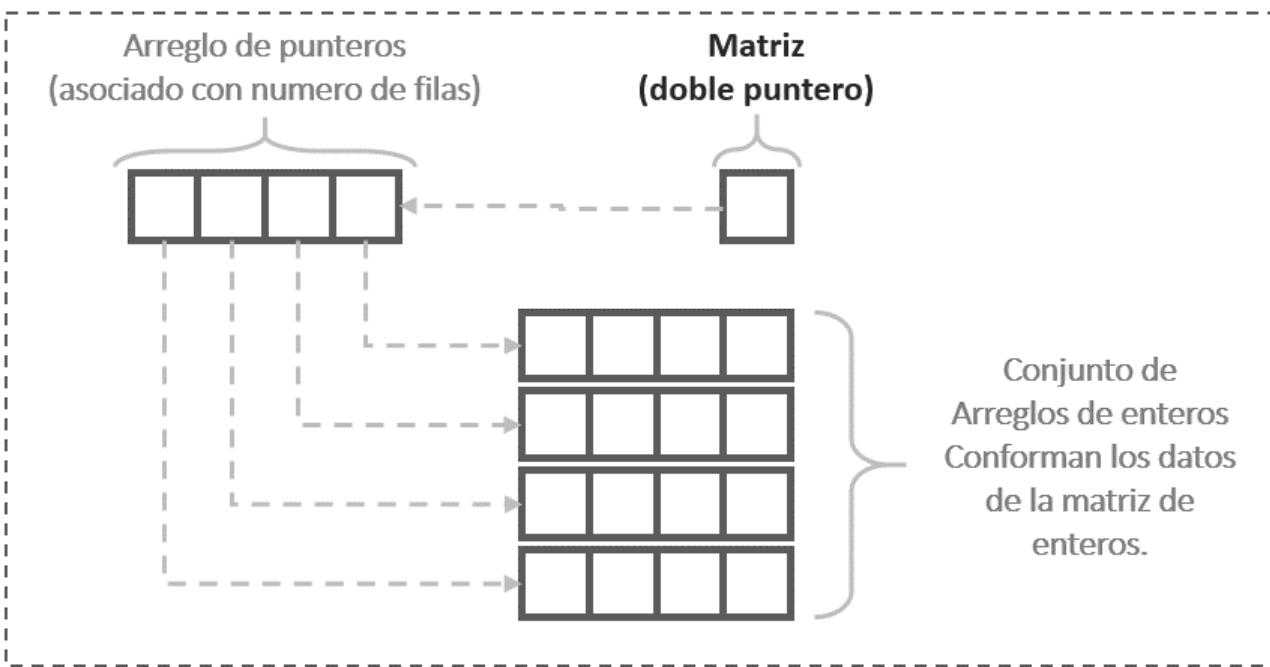


Figura 8.7 Estructura de una matriz dinámica.

La figura 8.7 muestra los elementos necesarios para construir una matriz en tiempo de ejecución. Es decir que al momento de conocer el número de filas y el número de columnas en tiempo de ejecución, se utiliza una variable doble puntero, se crea un arreglo de punteros con el número de filas, luego por cada fila se crea un arreglo del tipo de dato a utilizar utilizando el número de columnas. De esta manera el número de filas se asocia con el número de punteros que existe en el arreglo de punteros, y el número de elementos de cada arreglo, que apunta cada puntero del arreglo de punteros, es el número de columnas. Por lo tanto el doble puntero siempre conoce la dirección inicial del arreglo de punteros, conociendo esta dirección se suma el número de fila para avanzar sobre las diferentes filas, cada fila es un puntero a la dirección inicial de un arreglo de enteros (o el tipo de datos a utilizar) por lo que sumando el número de columnas se avanza sobre las diferentes columnas de una misma fila. La sintaxis para el uso de la matriz dinámica es igual que con una matriz estática, lo que cambia es que una matriz dinámica se debe crear en tiempo de ejecución respetando estos elementos para su creación, luego de su creación, todo es igual.

Por analogía para un vector se puede utilizar la misma lógica. Solo que en este caso con una variable del tipo puntero simple es suficiente, ya que no es necesario un arreglo de punteros. La idea sería como se muestra en la figura 8.8.

De esta forma en el momento de conocer el número de elementos del vector, se crea un arreglo del tipo de dato a utilizar utilizando dicho número. El puntero simple siempre conoce la primera dirección del arreglo de enteros (o tipo de dato a utilizar) y sumando el número de elemento necesario se recorre cada uno de los elementos del vector. Al igual que con las matrices dinámicas, cuando se realiza la creación del vector dinámicamente, se lo trata con la misma sintaxis que un vector estático.

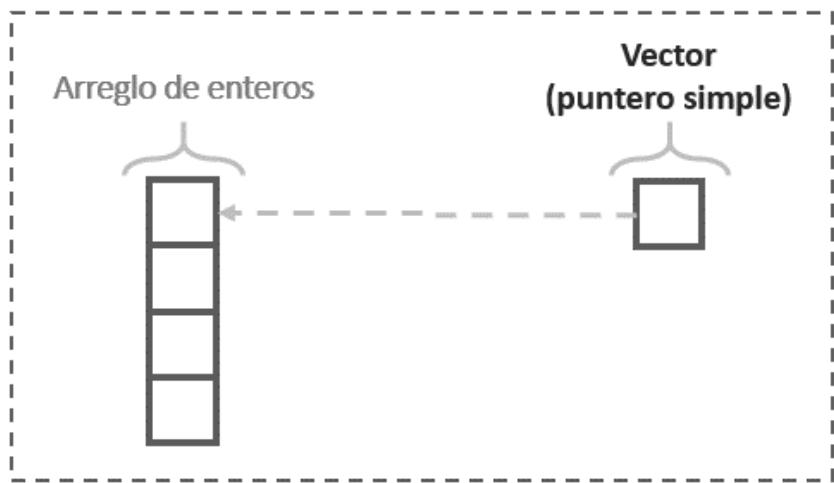


Figura 8.8 Estructura de un vector matriz dinámico.

8.5.1. IMPLEMENTACIÓN DE UNA MATRIZ DINÁMICA

Como se vio al comienzo de la sección, se debe tener en cuenta que la implementación de la asignación dinámica en matrices bidimensionales en el lenguaje C se basa de un proceso que se divide en dos partes:

- Asignar memoria a un arreglo de punteros, cuyos elementos referenciaran cada una de las filas de la matriz de dos dimensiones que se desea crear.
- Asignar memoria para cada una de las filas. El número de elementos de cada fila puede ser variable.

De esta forma, el bloque de código necesario de una función para manipular cada uno de los punteros para la creación dinámica de una matriz bidimensional, es el siguiente:

```

int **m;
int i = 0;

// Reserva de memoria para filas m (creación de arreglo de punteros)
m = (int **)vmalloc(filas*sizeof(int*));

// Reserva de memoria para columnas de m (creación de arreglos de enteros por cada m fila)
for (i=0;i<x->filas;i++)
    m[i] = (int*)vmalloc(columnas*sizeof(int));

```

Como se nota, la línea del código necesaria para crear el arreglo de punteros es la siguiente:

```
// Reserva de memoria para filas m (creación de arreglo de punteros)
m = (int **)vmalloc(filas*sizeof(int*));
```

Aquí la función sizeof() entrega el número de bytes requerido para un puntero tipo entero, al multiplicarse por el número de filas lo que se tiene es el tamaño real del arreglo de punteros para que la función vmalloc() realice la reserva de memoria con la magnitud indicada. Al final se realiza un casteo para poder entregar la dirección inicial del arreglo de punteros a la variable del tipo doble puntero entero.

Por otro lado para poder crear cada uno de los arreglos de enteros, se requerirá una instrucción cíclica como es el “for” ya que buscará espacio en forma independiente para cada arreglo de enteros. La función sizeof() retornará el tamaño en bytes requerido por cada entero, al multiplicarse por el número de columnas se tendrá el tamaño total de cada arreglo de enteros. La función vmalloc() buscará espacio en el HEAP para el tamaño solicitado por la función sizeof(), al final se realiza un casteo tipo puntero entero. El puntero que hace referencia al inicio del arreglo -m[i]- se entrega la variable tipo puntero entero.

```
// Reserva de memoria para columnas de m (creación de arreglos de enteros por cada m fila)
for (i=0;i< filas;i++)
    m[i] = (int*)vmalloc(columnas*sizeof(int));
```

La asignación de memoria mediante la función vmalloc() se utiliza para el espacio kernel, para el espacio usuario se reemplaza por la función malloc().

8.5.2. IMPLEMENTACIÓN DE UN VECTOR DINÁMICO

Análogamente la implementación de la asignación dinámica en vectores en el lenguaje C se basa de un proceso más simple de una sola parte:

- Asignar memoria para un arreglo de entero (o el tipo de dato a utilizar) con el número de elementos del vector.

De esta forma, el bloque de código necesario de una función para la creación dinámica de un vector, es el siguiente:

```
int *v;
int i = 0;

// Reserva de memoria de elementos de v (creación de arreglo de enteros)
v = (int*)vmalloc(columnas*sizeof(int));
```

Aquí la función sizeof() entrega el número de bytes requerido para un puntero tipo entero, al multiplicarse por el número de elementos que se tiene es el tamaño real del arreglo de enteros para que la función vmalloc() realice la reserva de memoria con la magnitud indicada. Al final se realiza un casteo para poder entregar la dirección inicial del arreglo de enteros a la variable del tipo puntero simple entero.

Se reitera que la asignación de memoria mediante la función vmalloc() se utiliza para el espacio kernel, mientras que para el espacio usuario se reemplaza por la función malloc().

9. REFERENCIAS BIBLIOGRÁFICAS

1. Micolini Orlando, Cebollada M, Eschoyez M, Ventre Luis O, Schild M, editors. Ecuación de estado generalizada para redes de Petri no autónomas y con distintos tipos de arcos. XXII Congreso Argentino de Ciencias de la Computación (CACIC 2016); 2016.
2. Brams GW. Las redes de Petri: teoría y práctica. Modelización y aplicaciones: Masson; 1986.
3. Desel J, Esparza J. Free choice Petri nets: Cambridge university press; 2005.
4. David R, Alla H. Petri nets and Grafset: tools for modelling discrete event systems. 1992.
5. Iordache M, Antsaklis PJ. Supervisory control of concurrent systems: a Petri net structural approach: Springer Science & Business Media; 2007.
6. Love Robert. Linux Kernel Development: Linux Kernel Development: Pearson Education; 2010.
7. Tanenbaum Andrew S. Sistemas operativos modernos: Pearson Educación; 2003.
8. Bovet Daniel P, Cesati Marco. Understanding the Linux Kernel: from I/O ports to process management: "O'Reilly Media, Inc."; 2005.
9. Corbet J, Rubini A, Kroah Hartman G. Linux Device Drivers: Where the Kernel Meets the Hardware: "O'Reilly Media, Inc."; 2005.
10. Salzman Peter Jay, Burian M, Pomerantz O. The linux kernel module programming guide. 2007.
11. Yadav V. Linux Kernel Symbols. (21/09/2012). Accedido a la fecha 10/06/2019. Consultado desde: <http://venkateshabbarapu.blogspot.com/2012/09/kernel-symbols.html>.
12. Comunidad de desarrollo del Kernel. Memory allocation guide. www.kernel.org. Accedido a la fecha 15/03/2019. Consultado desde: <https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html>.
13. Colaboradores de Wikipedia. Thrashing (computer science). In Wikipedia, The Free Encyclopedia. (2018, October 22). Accedido a la fecha 18/03/2019. Consultado desde: [https://en.wikipedia.org/w/index.php?title=Thrashing_\(computer_science\)&oldid=865151829](https://en.wikipedia.org/w/index.php?title=Thrashing_(computer_science)&oldid=865151829).
14. Colaboradores de Wikilibros. Programación en C/Manejo dinámico de memoria. Wikilibros. (2018, Noviembre 27). Accedido a la fecha 02/04/2019. Consultado desde: https://es.wikibooks.org/w/index.php?title=Programaci%C3%B3n_en_C/Manejo_din%C3%A1mico_de_memoria&oldid=360127.
15. Colaboradores de Wikipedia. Pérdida de memoria. Wikipedia, La Enciclopedia Libre. (2019, 1 de marzo). Accedido a la fecha 29/05/2019. Consultado desde: https://en.wikipedia.org/wiki/Memory_leak.
16. Neil Brown. Object-oriented design patterns in the kernel. (June 1, 2011). Accedido a la fecha 28/01/2019. Consultado desde: <https://lwn.net/Articles/444910/>.
17. Colaboradores de Wikipedia. Archivo de cabecera. Wikipedia, La enciclopedia libre. (2019, 20 de febrero). Accedido a la fecha 05/06/2019. Consultado desde: https://es.wikipedia.org/wiki/Archivo_de_cabecera.
18. Microsoft. Basic concepts. Declarations and definitions. Aliases and typedefs. (2017). Accedido a la fecha 29/05/2019. Consultado desde: <https://docs.microsoft.com/es-es/cpp/cpp/aliases-and-typedefs-cpp?view=vs-2017#typedefs>.
19. Estructuras opacas. Programación orientada a objetos en C. Accedido a la fecha 28/01/2019. Consultado desde: <https://trucosinformaticos.wordpress.com/tag/estructura-opaca/>.
20. Microsoft. UML Use Cases Diagram: Reference. docs.microsoft.com. Accedido a la fecha 25/04/2019. Consultado desde: <https://docs.microsoft.com/en-us/visualstudio/modeling/uml-use-case-diagrams-reference?view=vs-2015>.
21. Sommerville Ian. Ingeniería del software. Pearson Educación. 2011.;Novena Edición.
22. Fernando CH GA. Plan de pruebas de software. (18/02/2017). Accedido a la fecha 14/06/2019. Consultado desde: https://mundotesting.com/plan-de-pruebas-de-software/#El_Plan_de_Pruebas_de_Software_lo_integrar%F3n_las_siguientes_secciones.
23. Colaboradores de PMOinformatica. Paso para eleborar un plan de pruebas. (18/01/2016). Accedido a la fecha 15/06/2019. Consultado desde: <http://www.pmoinformatica.com/2016/01/elaborar-plan-pruebas-software.html>.
24. Knut Omang. Kernel Test Framework documentation. © Copyright 2017. Accedido a la fecha 10/12/2018. Consultado desde: <http://heim.ifi.uio.no/~knuto/ktf/>.
25. Åhman Niro. Testing of Linux Kernel Modules. 2017.
26. Eugene Shatokhin, Andrey Tsvarev. KEDR 0.5 Reference Manual. (May 12, 2014). Accedido a la fecha 18/06/2019. Consultado desde: https://github.com/euspectre/kedr/wiki/kedr_manual.
27. Shareef Ali, Zhu Yifeng, editors. Energy modeling of wireless sensor nodes based on Petri nets. 2010 39th International Conference on Parallel Processing; 2010: IEEE.

28. Anderson David J. Kanban: Cambio evolutivo exitoso para su negocio de tecnología: Blue Hole Press; 2010.
29. Gestión de proyectos y desarrollo de software. Desarrollo de software: ciclo de vida iterativo incremental. Accedido a la fecha 07/05/2019. Consultado desde: <https://jummp.wordpress.com/2011/03/31/desarrollo-de-software-ciclo-de-vida-iterativo-incremental/>.
30. Desarrollo iterativo e incremental. Accedido a la fecha 07/05/2019. Consultado desde: <https://proyectosagiles.org/desarrollo-iterativo-incremental/>.
31. Tabares-Betancur MS, Arango F, Anaya-Hernandez R. Una revisión de modelos y semánticas para la trazabilidad de requisitos. 2014.
32. Jose Sevilla. Trazabilidad de requerimientos (Tecnología). (12 de Julio 2016). Accedido a la fecha 29/04/2019. Consultado desde: <http://www.overti.es/tecnologia/313-trazabilidad-de-requisitos>.
33. Gestión de proyectos de software: Planificación de proyecto. Accedido a la fecha 10/05/2019. Consultado desde: <https://sites.google.com/site/gestiondeproyectossoftware/unidad-3-planificacion-de-proyecto/3-5-1-tipos-de-riesgos>.
34. Ayala de la Vega J, Aguilar Juárez I, Zarco Hidalgo A, Gómez Ayala H. Memoria dinámica en el lenguaje de programación C.