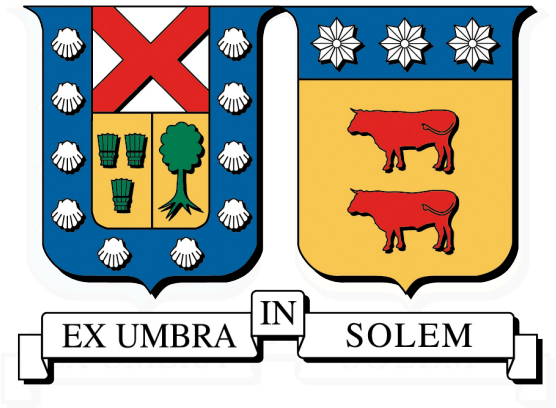
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
VALPARAISO - CHILE



INCORPORACIÓN DE EXPERIENCIAS PRÁCTICAS CON KERNEL REALES EN LA ENSEÑANZA DE SISTEMAS OPERATIVOS

CRISTIAN SEBASTIÁN FUENTES GARRIDO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

PROFESOR GUÍA : SR. JAVIER CAÑAS
PROFESOR CORREFERENTE : SR. HORST VON BRAND

FEBRERO 2016

EX UMBRA
IN
SOLEM

Agradecimientos

A mi familia

Por todo el apoyo tango monetario como anímico lo que me hizo estar y mantener mis estudios en la V Región, claramente sin esto no se puede lograr una meta como ésta, en especial a mis padres y a mis hermanos quienes se dieron el trabajo de ayudarme a costa de sus intereses.

A mi pareja

Lily, por todo el ánimo y apoyo que me diste día a día y por aguantar todos estos viajes sin poder vernos de una forma continua, el cariño que nos tenemos es un sustento de vida.

A mis amigos

Acá tengo a varios, en primer lugar a Andrea quien me ha aguantado todos estos años y a pesar de todo continúa esta amistad. A mis compañeros de departamento en especial a Roberto por todas esas pizzas que servían como energía en esos días de estudio y no estudio. A Maximiliano, Alondra, Alejandro Sazo, Alejandro Díaz y Daniela por la amistado brindada en todo este periodo académico. Y a todos mis compañeros con los cuales se compartió y se logró finalizar las asignaturas como corresponde.

A mis profesores

A mi profesor guía Javier Cañas, quien me llevó en buen camino para finalizar esta memoria. A mi profesor correferente Horst von Brand por darse el tiempo de encontrar cada detalle. Y a cada uno de los profesores de cada asignatura dada por la enseñanza que dejaron para convertirme en un profesional de esta universidad.

”Eres como un pez grande en un estanque pequeño.”

Edward Bloom - Big fish (2003)

RESUMEN EJECUTIVO

El trabajo consiste en la investigación de métodos de aprendizaje utilizados para la enseñanza de la construcción de un sistema operativo. Se busca que el alumno pueda tomar un kernel para poder modificar y aprender con la práctica, con ello se evalúan diez cursos distintos, se escoge uno de ellos según ciertos discriminantes y se trabaja en la modificación de éste para que esté adaptado a la realidad del Departamento de Informática de la Universidad Técnica Federico Santa María.

El trabajo concluye con la elaboración de experiencias prácticas de carácter evolutivo en el cual se toma el esqueleto de un kernel real, se construye uno más robusto durante el periodo que transcurre el curso y así se aprende en la marcha.

Palabras Clave. Xv6, MIT, Núcleo, Sistemas Operativos.

ABSTRACT

The work consists in the investigation of learning methods used to teach the construction of an operativing system. You have to look for the pupil can take a "kernel"to modify and learn with practice, with this you can evaluate ten differet groups, it is chosen one of them according to certain things and you work in the modification of this group to be addapted to the reality of the Departamento de Informática of the Universidad Técnica Federico Santa Maria.

The work ends with the summery of practice experiences of evolutive character in wich you can take the squeleton of a real kernel, so you can build a stronger one during the period that last the course so in that way you they learn.

Índice de Contenidos

EX UMBRA
IN
SOLEM

Índice de Tablas

1 | Introducción

Actualmente en la carrera de Ingeniería Civil en Informática de nuestra Universidad, en el área de Infraestructuras TIC, más específico en la asignatura de Sistemas Operativos, se da un enfoque teórico sobre la construcción de un sistema operativo como tal[10]. Es clara la separación entre lo teórico y práctico donde los alumnos no alcanzan a experimentar que es la modificación y elaboración de un sistema como éste[9].

Se busca lograr una asignatura para el Departamento de Informática que solucione esta brecha y que permita a los alumnos interesados ver el lado práctico de la construcción de un sistema operativo como tal, más específico, experimentar con módulos de kernels.

El propósito del curso que se elaborará consiste en darle al alumno el conocimiento necesario de cómo trabajan realmente los computadores a tal bajo nivel, dándole las herramientas requeridas para que, con un lenguaje de alto nivel (Como el lenguaje C)[19], se consiga la creación o modificación de módulos de kernel, los cuales puedan experimentar en un ambiente de prueba real.

El alumno experimentará los fundamentos de los engranajes de un sistema operativo, tales como la memoria virtual, el modo usuario, llamadas de sistemas, threads, interrupciones, comunicación interprocesos, coordinación, concurrencia y la interfaz presente el software y el hardware lo cual es lo más importante en general de todo este proceso. Se busca que el alumno aprecie todo lo que es el proceso de diseño y vea la importancia de la reducción de la complejidad y la integridad conceptual. Para poder abarcarlos, es necesario analizar un kernel de sistema operativo, el cual está definido como la parte en donde se ejecutan todos los comandos que están en modo kernel y permite un acceso seguro al hardware mediante llamadas al sistema[13]. También es el encargado de la gestión de recursos como el análisis de los dispositivos de E/S[13].

La investigación se centrará en la búsqueda de métodos de aprendizaje ya aplicados en otras partes del mundo; esto engloba a instituciones educacionales, organizaciones sin fines de lucro y tutoriales simples de Internet. Se evaluarán sus contenidos y métodos de enseñanza para poder converger a uno o más métodos que se vean factibles para el ambiente del Departamento de Informática.

1.1. Objetivos

1.1.1. Objetivo Principal

- Proporcionar herramientas para unir teoría y práctica en el aprendizaje de sistemas operativos.

1.1.2. Objetivos Específicos

- Seleccionar uno o más Sistemas Operativos utilizados para el proceso educativo.
- Elaborar un plan de enseñanza para el aprendizaje.
- Establecer tópicos de carácter acumulativo para las distintas experiencias del taller.

1.2. Estructura del Documento

Comenzamos con la situación actual, en donde vemos como otras instituciones de educación han abordado este tema y cuales son algunos de los Sistemas Operativos que han sido creados para el aprendizaje del diseño, todo esto en el capítulo 2. Luego realizamos una comparación de todos estos cursos, mediante el método de evaluación de incorporación de tópicos relevantes para la construcción de un Sistema Operativo, los cursos son comparados por igual y se discrimina según criterios nombrados; ésto lo encontramos en el capítulo 3. Las compilaciones logradas de los sistemas operativos se encuentran documentadas en el Anexo A. Posterior a la evaluación y como resultado de ésta, tenemos el desarrollo del curso escogido con la construcción de uno especial para el Departamento de Informática,

el desarrollo del curso lo encontramos en el Anexo B y la construcción del curso se encuentra en el capítulo 4. Finalmente tenemos la conclusión del trabajo, la cual engloba los resultados obtenidos luego de toda esta investigación, lo que se encuentra en el capítulo 5. Las experiencias construidas para el curso se encuentran en el Anexo digital.

EX UMBRA
IN
SOLEM

2 | Estado del Arte

En este capítulo se investigará sobre la existencia de métodos de aprendizaje que se han utilizado para poder interiorizarse en lo que es un sistema operativo a bajo nivel. Se encontraron diez métodos de los cuales dos de ellos son tutoriales de construcción de un sistema operativo desde cero, uno es un kernel no compilado que está construido por partes para ir explicando su proceso, y siete cursos de distintas universidades con sus respectivos objetivos y métodos de enseñanza.

2.1. Cursos Implementados en Otras Universidades

2.1.1. Curso de Massachusetts Institute of Technology

El curso 6.828 de Massachusetts Institute of Technology (MIT) está orientado principalmente al estudio de ideas sobre el diseño y la implementación de un sistema operativo basado en UNIX[22]. El curso es dictado por Frans Kaashoek, profesor del *Department of Electrical Engineering and Computer Science*, miembro del *MIT Computer Science and Artificial Intelligence Laboratory* y principal colaborador del libro *"xv6, a simple, Unix-like teaching operating system"*[29] junto a Russ Cox y Robert Morris.

El plan de estudio se centra en el sistema operativo Xv6, el cual fue desarrollado en

Semanas de duración	16 Semanas
Horas/sesiones de clase por semana	2 Sesiones
Horas/sesiones de taller por semana	-
Horas estimadas de dedicación adicional (semanas)	4 Horas
Horas promedio dedicadas por taller	9 Horas

Tabla 2.1: Horas de dedicación Curso de Massachusetts Institute of Technology

verano de 2006 para este curso, siendo una modificación del ya obsoleto Unix V6[18] en donde los estudiantes experimentaban con código de hace 30 años atrás. Se moldeó de tal forma que pasó a ser multiplataforma y con soporte para multiprocesadores, de tal manera que V6 fue reemplazado[24].

La evaluación del curso consiste en la realización de dos pruebas; una a mediados del semestre y otra en las últimas semanas, cinco laboratorios de trabajo individual y dos laboratorios de trabajo grupal. Las ponderaciones de cada trabajo se adjuntan en la tabla 2.2. Los laboratorios están construidos de tal forma que la experiencia sea incremental y el código sea reutilizado para los siguientes laboratorios; se basan en la construcción de un SO con un kernel de tipo exokernel el cual está principalmente escrito en lenguaje C. Es necesario tener finalizado el laboratorio 1 para proseguir con el 2, y así sucesivamente[22]. Los tópicos abarcados por laboratorio son:

	Ponderación
Pruebas	30 %
Labs 1 - 5	40 %
Tareas, revisiones, Participación en clases	15 %
Retos y Proyecto final	15 %

Tabla 2.2: Ponderaciones curso Xv6 de MIT.

Lab 1 ***Booting a PC*** La experiencia se separa en tres partes. La primera corresponde a un acercamiento a las herramientas a utilizar durante todo el curso, tal como el emulador QEMU x86 para correr el sistema operativo, lenguaje assembly y el sistema de booteo de un computador. La segunda parte corresponde al booteo desde el mismo kernel. Y la última corresponde a la construcción del template inicial para el kernel a construir.

Lab 2: ***Memory Manager*** El objetivo principal de la experiencia es escribir en código el manejo de memoria del sistema operativo, realizando el trabajo tanto para la memoria física y la memoria virtual. Para la memoria física el trabajo consiste en tener estructuras que guarden qué espacios están libres o no junto el lugar en donde se

encuentran localizadas y cuántos procesos comparten dicho espacio y cuáles son. Por otro lado, para la memoria virtual se busca que se construyan las tablas de páginas según se estime conveniente para las siguientes experiencias.

Lab 3 ***User Environments*** Se espera que al finalizar esta experiencia se tenga un sistema protegido para utilizar el modo usuario y así poder correr procesos sin inconvenientes. Como finalidad, el trabajo debe ser ejecutado junto a un proceso en él. Se trabaja el manejo de excepciones e interrupciones, errores de páginas y llamadas al sistema.

Lab 4 ***Preemptive Multitasking*** Se busca implementar múltiples y simultáneos ambientes de modo usuario. La experiencia se divide en tres partes, la primera corresponde a la extensión del kernel para el soporte de multiprocesos y nuevas llamadas al sistema para que en modo Usuario se puedan crear nuevos ambientes, implementar un sistema cooperativo de tipo round-robin para la ejecución de procesos y la implementación de un sistema interrumpible para que el kernel pueda volver a retomar el control de la CPU luego de un tiempo determinado, incluso si el entorno no coopera. La segunda parte consiste en la implementación de fork() en modo Usuario. En la tercera parte se trabaja la construcción de la comunicación entre procesos, de tal forma que distintos modo Usuarios se comuniquen y sincronicen entre sí.

Lab 5 ***File system, Spawn and Shell*** La experiencia se basa en la construcción de la biblioteca *spawn* para permitir leer y escribir archivos en el disco duro. También dejar preparado el sistema para poder correr una terminal por consola.

Lab 6 ***Network Driver (default final project)*** Es un trabajo individual y corresponde a la primera parte del proyecto final. El objetivo es conectar el sistema operativo a la red; para ello es necesario escribir un driver para la interfaz de red y hacer funcionar bajo QEMU transmitiendo paquetes.

Lab 7 ***Final JOS project*** Consiste en extender todo el trabajo realizado hasta el momento a lo que el/los alumnos estimen conveniente. Es un trabajo grupal como individual.

El curso completo, tanto los laboratorios como el código fuente, están bajo la licencia

tradicional del MIT[27] con lo cual es posible la utilización y adaptación de toda su información mientras se referencie su origen.

2.1.2. Curso de Carnegie Mellon University

La universidad *Carnegie Mellon* en su departamento de *Computer Science*, implementa el curso 15-410 llamado *Operating System Design and Implementation*. El curso se basa en la experiencia de escribir una pequeña unidad de kernel de Unix en lenguaje C y x86 Intel Assembly[7].

Semanas de duración	16 Semanas
Horas/sesiones de clase por semana	2 Sesiones
Horas/sesiones de taller por semana	-
Horas estimadas de dedicación adicional (semanas)	3 Horas
Horas promedio dedicadas por taller	7,5 Horas

Tabla 2.3: Horas de dedicación Curso de Carnegie Mellon University

El curso está implementado para ser desarrollado en grupos de dos personas y se enfoca directamente en la modificación y creación de módulos para un sistema operativo base. Dichos módulos se trabajan con el lenguaje de programación C. El curso ofrece un sistema operativo base con el sistema de booteo incluido[7].

	Ponderación
Proyecto 0	5 %
Proyecto 1	5 %
Proyecto 2	15 %
Proyecto 3	25 %
Proyecto 4	5 %
Primer examen	15 %
Segundo examen	20 %
Tareas y reportes	10 %

Tabla 2.4: Ponderaciones curso de Carnegie Mellon University

Durante el transcurso del curso se califican cinco proyectos grupales y dos exámenes. La información requerida para cada uno de los tópicos es bastante densa la cual requiere de mucha lectura. Las ponderaciones están en la tabla 2.4 y los tópicos del proyecto son los siguientes:

Proyecto 0 ***Traceback*** El objetivo es crear la librería de *traceback* con la intención de poder leer memoria de un nivel determinado de una pila. La idea es obtener un sistema que nos permita conocer un error cuando exista dicha excepción. La creación de esta librería permitiría conocer la información que dispone la tarea predefinida para este caso.

Proyecto 1 ***Nonogram*** Consiste en la creación de tres drivers fundamentales para la interacción del usuario en el sistema operativo, la creación de una consola de terminal que permita imprimir carácteres por pantalla, crear el controlador para el teclado para lograr capturar las teclas presionadas, y lo último es el manejo del tiempo y su captura para el uso en el sistema.

Proyecto 2 ***Hand-in*** La experiencia se basa en la construcción del sistema de reserva de memoria de tipo Mutex con variables de condición, la creación de la librería de Thread y librerías extras de rutina como semáforos y r/w lock.

Proyecto 3 ***Introduction to the Kernel Project*** Es la parte del proyecto en donde se requiere la mayor cantidad de conocimientos. Se busca crear el sistema de multitasking, el manejo múltiple de direcciones virtuales, un aumento notorio de las llamadas a sistemas y un aumento en general del proyecto en sí. Pretende que los alumnos analicen lo que pueden agregar al sistema.

Proyecto 4 ***DMA+CLLFS*** Se centra en la creación del driver de un disco IDE para dar la posibilidad de usar interrupciones y tener acceso directo a la memoria, además de la creación completa de un sistema distinto de *file system* con tal de utilizar la llamada a sistema `readfile()` y ser utilizada para otras nuevas llamadas necesarias para el completo funcionamiento[7].

El curso está bajo la licencia CMU[36] la que permite su uso, copia, modificación y distribución mientras que se comente el origen de éste. Pero se debe tener en cuenta que en dicho curso utilizan el simulador Simics, el cual necesita de una licencia especial de *Wind River* (www.windriver.com) la que ofrecen de forma gratuita para propósitos educativos con previa solicitud.

2.1.3. Curso de Standford University

La universidad de Standford imparte el curso *CS 140: Operating System* dictado por el profesor John Ousterhout. El curso se basa en la modificación de los módulos del sistema operativo PintOS[38] con tal de entender el funcionamiento en sus distintos componentes. PintOS es un framework de sistema operativo para arquitecturas 80x86, soporta threads, carga de programas y *file system*, todos implementados en su forma más simple posible. El curso se centra en hacer PintOS un sistema operativo más robusto[37].

	Ponderación
Problemas	5 %
Proyecto	50 %
Prueba 1	15 %
Prueba 2	30 %

Tabla 2.5: Ponderaciones curso de Standford University

El curso se compone de dos pruebas, una a mediados del curso y otra al finalizar, cuenta con problemas tipo desafío y el proyecto, el cual se separa en cuatro trabajos que se reparten durante todo el semestre, puede tomar entre 10 a 20 horas semanales con la posibilidad de ser realizado entre 2 a 3 personas[37].

Semanas de duración	10 Semanas
Horas/sesiones de clase por semana	3 Sesiones
Horas/sesiones de taller por semana	Algunas Sesiones*
Horas estimadas de dedicación adicional (semanas)	10-20 Horas
Horas promedio dedicadas por taller	25 Horas

Tabla 2.6: Horas estimadas Curso de Standford University

Los problemas de desafío son ejercicios de trabajo individual que deben ser desarrollados dentro de PintOS y abordan el tópico de sincronización. Se debe programar en lenguaje C dos funciones que utilizan funciones propias de PintOS con tal de familiarizarse con las bibliotecas que trae implementadas[37]. Por otro lado, el proyecto consta de los siguientes tópicos según número de actividad a realizar:

Proyecto 1 *Threads* Se les entrega a los alumnos un sistema mínimo funcional de threads bajo PintOS que deben implementar en todo el sistema de sincronización de threads

y manejo de prioridades, además de implementar la función `timer_sleep()` para proporcionar esperas en los procesos.

Proyecto 2 ***User Program*** Los alumnos deben simular un disco duro con las propiedades que trae PintOS, se utilizan las funciones de *file system* para probar esta experiencia. Se debe implementar un sistema que nos indique cuándo un proceso ejecutado por un usuario es finalizado, implementar además la función `process_execute()` con paso de argumentos (ya que la que se encuentra no lo permite) y crear distintas llamadas a sistemas tales como `wait, halt, exec, exit, open` entre otros.

Proyecto 3 ***Virtual Memory*** El objetivo es quitar la limitación por hardware de la utilización de memoria implementando memoria virtual. Se debe crear un sistema de paginación de segmentos cargados de ejecutables con tal de aproximarse al sistema LRU, implementar el Stack Growth que aloje estas páginas de segmentos y los permisos necesarios para su funcionamiento, y crear las llamadas a sistemas `mapid_t` y `munmap` para el mapeo de memoria.

Proyecto 4 ***File Systems*** Los alumnos deben modificar los archivos ya existentes de *file system* con tal de agregar una nueva indexación de archivos según estimen conveniente, crear un manejo de subdirectorios y las llamadas a sistema correspondientes tales como `mkdir, isdir` y `chdir`, crear un Buffer Cache que mantenga los bloques de archivos utilizados recientemente, y por último crear un sistema de sincronización de lectura/escritura de los archivos del disco.

El curso se encuentra bajo licencia Creative Commons[5] con atribución mientras se de el crédito correspondiente. La documentación del sistema operativo se encuentra libre y todas las presentaciones de la asignatura se encuentran disponibles para la lectura en su página web[37].

2.1.4. Curso de Harvard, School of Engineering and Applied Sciences

En la Universidad de Harvard en la escuela de ingenieros, se dicta el curso *CS 161: Operating Systems*[6] dictado por el profesor Margo Seltzer. El curso se basa en la construc-

ción y modificación del kernel OS161 complementando con clases lectivas y un proyecto separado por cinco partes que envuelven los distintos tópicos a estudiar.

Semanas de duración	16 Semanas
Horas/sesiones de clase por semana	2 Sesiones
Horas/sesiones de taller por semana	-
Horas estimadas de dedicación adicional (semanas)	3 Horas por clase + 2 Proyecto
Horas promedio dedicadas por taller	10 Horas

Tabla 2.7: Horas estimadas Curso de Harvard, School of Engineering and Applied Sciences

El sistema operativo OS161 es provisto por la universidad y está basado en el sistema UNIX. El kernel está construido con lenguaje Assembly x86 y C y viene con los requisitos mínimos necesarios para su funcionamiento y acceso a su propia terminal[6].

	Ponderación
Participación en clases	10 %
Tareas asignadas	50 %
Prueba 1	15 %
Prueba 2	25 %

Tabla 2.8: Ponderaciones curso de Harvard

El curso cuenta con videos explicativos y con diapositivas disponibles de forma pública. El temario de las experiencias es el siguiente:

Tarea 0 ***An Introduction to OS/161*** Es un trabajo individual y consiste en un acercamiento al código que utilizará el alumno durante todo el curso, configurar la máquina virtual, clonar los repositorios, configurar y construir el kernel, además de la utilización de git para almacenar todo el proyecto.

Tarea 1 ***Synchronization*** Trabajo en equipo de dos personas el cual consiste en conocer como actualmente OS161 está implementando semáforos. Además deben implementar sistemas primitivos de sincronización tales como *Locks* y *Condition Variables* para luego probar dicho código en problemas planteados de sincronización.

Tarea 2 ***System Calls and Processes*** En un equipo de dos personas, los alumnos deben comenzar a construir las piezas faltantes de las llamadas a sistemas que dispone

OS161 en lenguaje C para luego correr SYS161 en modo usuario y lograr utilizarlas sin problemas. Los estudiantes deben construir más de 14 funciones que se encuentran incompletas.

Tarea 3 ***Virtual Memory*** Como en la experiencia anterior se posibilitó la utilización de varios programas, el problema ahora es la falta de memoria para ejecutarlos todos, por lo que es necesario adaptar OS161 para que implemente todo un sistema de memoria virtual y sistema de paginación. Finalmente el trabajo debe ser funcional al momento de probar con un programa provisto por el curso que utiliza las funciones de C `malloc` y `free`.

Tarea 4 ***File Systems*** OS161 trae implementado un sistema llamado SFS (*Simple File System*). Lo que se espera de la experiencia es que los alumnos modifiquen este sistema y agreguen indexación, sincronización de datos y un sistema de permita volver a funcionar luego de recibir algún tipo de error de lectura/escritura.

El curso cuenta con toda la documentación disponible en la red, con contenidos generales por tópicos, materiales para antes de clases y los repositorios git disponibles públicamente para su utilización[6].

2.1.5. Curso de University of Illinois at Chicago

El curso impartido por la Universidad de Illinois corresponde a *CS 486: Secure Operating System Design and Implementation*. El curso, al igual que los anteriores, corresponde a la construcción de módulos de kernel e implementación de un sistema operativo (este sistema se espera que corra en Xen). Los alumnos deben programar en lenguaje C un sistema operativo seguro, ya que el programa se basa en Ethos[31] el cual es un sistema operativo *open source* especializado para evitar ataques externos[30].

El curso se basa principalmente en la construcción del proyecto. Las clases presenciales consisten en presentaciones de avances más la presentación del proyecto final, además de una prueba que envuelve todos los temas tratados durante la construcción del sistema operativo[30].

Semanas de duración	* No disponible *
Horas/sesiones de clase por semana	* No disponible *
Horas/sesiones de taller por semana	* No disponible *
Horas estimadas de dedicación adicional (semanas)	* No disponible *
Horas promedio dedicadas por taller	* No disponible *

Tabla 2.9: Horas estimadas Curso de University of Illinois at Chicago

El syllabus no se encuentra disponible públicamente, pero sí se encuentran las seis etapas del proyecto a realizar durante el curso las cuales son:

Tarea 0 ***Install VMware on your computer*** Consiste en implementar la zona de trabajo para toda la asignatura para poder utilizar máquinas virtuales Xen en VMware. Se recomienda trabajar bajo Fedora 20.

Tarea 1 **`memory/string/printf`** El trabajo consiste en crear, en lenguaje C, el código necesarias para hacer funcionar las rutinas `memcpy, memset, memcmp, strcat, strncat, strcmp, strncmp, strcpy, strncpy` y `printf`. Las funciones deben ser probadas por un programa provisto por el curso.

Tarea 2 **`library/malloc/free`** Se basa en crear las funciones para el manejo de memoria de las aplicaciones a utilizar. Este trabajo se debe complementar con la Tarea 1 y debe ser testeado con lo provisto en el curso.

Tarea 3 **`console/timer/events`** Se debe construir una consola para el sistema operativo, más rutinas de programación para tener un sistema interactivo de sincronización.

Tarea 4 **`page walk`** Fase de creación de páginas de memorias. El alumno debe implementar la función `tableWalk(vaddr)` la que permitirá la utilización de memoria virtual.

Tarea 5 **`page table create`** Se requiere la implementación de la función `virtualMap` quien inicializará el sistema de memoria virtual al arrancar el sistema operativo. Se deben tener en cuenta todos los funcionamientos que éste requiera[30].

Las presentaciones en diapositivas con la información necesaria del curso, parte del código y las explicaciones de cada tarea están disponibles para todo público mientras que

se cite su origen. No cuenta con una licencia de distribución por lo que no se tienen los permisos necesarios para su utilización.

2.1.6. Curso de University of California

En la Universidad de California, en la división de Berkeley Computer Science, se imparte el curso *CS194-24: Advanced Operating Systems Structures and Implementation*[16] con clases lectivas presenciales en donde se explica cada detalle del proceso de diseño y construcción de un sistema operativo en conjunto con un laboratorio acumulativo de seis etapas en el que se modifica un sistema operativo basado en Unix creado para el curso el cual pueda ser utilizado como *webserver*[16].

	Ponderación
Prueba 1	25 %
Prueba 2	25 %
Proyecto	45 %
Participación en clases	5 %

Tabla 2.10: Ponderaciones curso de University of California

Semanas de duración	17 Semanas
Horas/sesiones de clase por semana	2 Clases
Horas/sesiones de taller por semana	-
Horas estimadas de dedicación adicional (semanas)	8 Horas
Horas promedio dedicadas por taller	20 Horas

Tabla 2.11: Horas estimadas Curso de University of California

Los temas abarcados en las clases presenciales se enfocan directamente en lo que se necesita implementar en el laboratorio. Se comienza con las definiciones correspondientes, las formas de correr el sistema operativo a utilizar y se debate sobre la estructura de un sistema operativo en sí, todo ello en la primera etapa del curso para no topar con la elaboración del laboratorio[16]. Para ello se forman grupos de dos a tres y los tópicos de cada uno son los siguientes:

Lab 0 ***Introduction*** El objetivo es familiarizar a los alumnos al ambiente en el que trabajarán durante toda la asignatura y configurar la máquina virtual, se instala VMware

y se configura de tal forma que corra el sistema operativo UNIX base que ofrece el curso.

Lab 0.5 ***Fish*** El objetivo del laboratorio es poder ver por terminal la animación de un pescado, con ello se es necesario el conocimiento de llamadas al sistema, mapeo de memoria, algunas rutinas primitivas de Linux y la infraestructura como tal. Existe un archivo base en lenguaje C con las funciones del pescado a implementar utilizando las pocas herramientas que se tienen.

Lab 1 ***httpd*** La experiencia de laboratorio consiste en la creación de puertos de red y la configuración de este sistema operativo para que pueda ser un servidor http. Es necesario familiarizarse con el entorno de desarrollo POSIX.

Lab 2 ***fs*** en este laboratorio es importante la creación de *file system* con la posibilidad de actualizar información, cifrar storage, calcular checksum y rápida recuperación ante fallos.

Lab 3 ***Scheduling*** El objetivo es crear el sistema de rutinas de los procesos del sistema operativo. Es importante resaltar que debe funcionar como corresponde ya que trabajará como web server.

Lab 4 ***Device Drivers*** Se pide la implementación de un driver de red para el sistema operativo. Es necesario escribirlo desde cero y se debe realizar para la red proporcionada por el sistema de virtualización, en este caso se trabaja bajo ETH194[16].

El curso no cuenta con información sobre su distribución, pero sí con la documentación de todo el curso y con los repositorios git disponibles para quien los quiera utilizar. Es posible realizar un curso utilizando el mismo material[16].

2.1.7. Curso de University of Florida

En la Universidad de Florida, en el *Department of Computer and Information Science and Engineering* se imparte el curso *COP 4600: Operating Systems* realizado por el Doctor *Sumi Helal*[32]. El curso se centra en el aprendizaje de sistemas operativos, separando

dicho aprendizaje en clases presenciales y experiencias de laboratorio utilizando el pequeño sistema operativo Xinu[8]. El curso se focaliza en el diseño y funcionamiento interno de un sistema operativo computacional. Cubre los conceptos, principios, mecanismos, políticas, funcionalidades, diseño e implementación de un sistema operativo que soporte el funcionamiento de procesos con el principal objetivo de vivir la experiencia de construir uno propio.

Semanas de duración	15 Semanas
Horas/sesiones de clase por semana	2 Clases
Horas/sesiones de taller por semana	-
Horas estimadas de dedicación adicional (semanas)	* No Disponible *
Horas promedio dedicadas por taller	* No Disponible *

Tabla 2.12: Horas estimadas Curso de University of Florida

La universidad realiza dos evaluaciones escritas realizadas al inicio y a la mitad del semestre, un examen final, cuatro laboratorios y un proyecto grupal en los cuales se trabaja con Xinu.

	Ponderación
Laboratorios	20 %
Exámenes	60 %
Proyecto	20 %

Tabla 2.13: Ponderaciones curso de University of Florida

Este curso no proporciona los métodos de evaluación para experimentar con el código del sistema operativo Xinu. Lo único que revela son sus tópicos a pasar en el semestre, los cuales muestran claramente que son ordenados para la realización de los laboratorios al tener una gran similitud con los temas de los cursos nombrados anteriormente. Los temas son:

- The Hardware and Run-Time Environment
- Data structures for Operating Systems
- Processes, Scheduling and Context Switching
- Process Management

- Inter-process Communication
- Basic Memory Management
- Virtual Memory
- Device-independent Input/Output
- The Shell Project review
- Real-time Clock Management
- File Systems

Mientras el curso se efectúa, los alumnos deben de ir leyendo ciertos capítulos del libro *Operating System Design - The XINU Approach*[3], el cual corresponde a las instrucciones de construcción y funcionamiento del sistema operativo utilizado en este curso.

2.2. Sistemas Operativos Creados para el Aprendizaje no utilizados en cursos

2.2.1. CROCOS 0.2

CROCOS es un sistema operativo en construcción el cual se está confeccionando para su utilización en la educación. Es un pequeño kernel de UNIX para sistemas x86/x64, su distribución está bajo la licencia GNU versión 3[33] y se encuentra almacenado mediante pasos según el tópico en construcción[11].

Por el momento sólo cuenta con un sistema de administrador de tareas que corre dentro de un proceso de Linux. Más adelante se espera tener más módulos en el kernel como un sistema de manejo de memorias, *filesystem* y más[11].

La construcción del sistema operativo ha pasado por las siguientes fases:

- Fase 1: kernel libc
- Fase 2: Administrador de tareas

  - Fase 2.1: Un simple administrador de tareas
  - Fase 2.2: Jerarquía de procesos y tareas iniciales
  - Fase 2.3: Señales
  - Fase 2.4: Pila de Kernel y de Usuario
  - Fase 2.5: Sincronización de procesos (Mutex)
- Fase 3: *File system* de sólo lectura
  - Fase 3.1: Leyendo la estructura de *file system*
  - Fase 3.2: Descriptor de archivos
  - Fase 3.3: Leyendo archivos regulares
  - Fase 3.4: Leyendo directorios
  - Fase 3.5: \proc *file system*

Y como trabajo futuro la planificación es:

- Fase 4: Escribir en *file system*
- Fase 5: Secuencia de Boot (para funcionar fuera de un proceso)
- Fase 6: Memoria virtual
- Fase 7: Carga de binarios
- Fase 8: driver tty
- Fase 9: Utilidades y *newlib*

El trabajo que se lleva hasta ahora puede ser descargado en la página SourceForge y se encuentra separado según las fases de desarrollo nombradas anteriormente. De esta forma el estudiante puede ir analizando paso a paso el trabajo de construcción que se lleva a cabo para tener un sistema operativo.

No existe hasta el momento una actividad realizable con este proyecto, lamentablemente éste dejó de ser actualizado en Abril del año 2013[11] y no se ve que exista una continuidad de parte de los desarrolladores.

2.3. Alternativas para el Aprendizaje Buscado

Existen varios tutoriales en la red que permiten la elaboración de un sistema operativo simple partiendo desde cero, los cuales permiten tener la noción de las necesidades que se requieren para la construcción de un sistema más robusto.

2.3.1. JamesM's kernel development tutorials

Existe un tutorial del año 2008 creado por James Molly[25] en el cual en 10 etapas nos permiten tener un sistema operativo. En pocas palabras, esto es lo que se logra en cada paso:

1. Creación de la base del sistema operativo, directorios, Makefile y configuración con Bochs
2. El código de booteo para el sistema operativo
3. Configuración de la pantalla, teclado y mouse
4. Tablas de memorias
5. Manejo de interrupciones
6. Paginación y memoria virtual
7. Creación del Heap
8. *Filesystem* y *initrd*
9. Multitasking
10. Modo Usuario

El tutorial nos permite crear el sistema con lenguaje C y assembly x86 en ciertos lugares. Lo creado es simplemente una de dar a conocer las partes que componen un sistema operativo[25] .

2.3.2. Bran's Kernel Development

Es un tutorial de año 2005 que se basa en la creación de un sistema operativo para procesadores superior o igual a 385. Su construcción es en lenguaje C y assembly x86, es similar al tutorial indicado en 2.3.1 pero está más centrado en el hardware y está explicado con mayor detalle. Los pasos para la construcción se basan en los siguientes temas:

1. La entrada de Kernel y el script linker
2. Creación de main() y el link del código C
3. Imprimir por pantalla
4. Implementación de Global Descriptor Table
5. Implementación de Interrupt Descriptor Table
6. Rutinas de interrupciones
7. Peticiones de Interrupción (IRQ) y controles programables de interrupción (PIC)
8. Implementación de System Clock
9. Teclado

Luego el tutorial lo deja a libre disposición del usuario, con ideas tales como agregar multitasking, memoria virtual, configurar nuevos drivers VGA para obtener más colores o VESA para videos y mayor resoluciones[12].

Mientras se realizan los pasos de este tutorial, este mismo va explicando de forma superficial los tópicos mínimos necesarios para conocer que es lo que se está haciendo. Se estima un trabajo de no más de una hora por cada uno de los temas.

EX UMBRA
IN
SOLEM

3 | Evaluación de Sistemas

Cada método de aprendizaje nombrado en el capítulo 2 será evaluado según la información que es posible aprender, los temas a los cuales aborda y qué tiene y qué no tiene que hacer el alumno para avanzar en dichos cursos, además de evaluar la cantidad de líneas de código que se esperan construir, siempre y cuando los cursos nombrados tengan disponible su información de forma liberada.

Los cursos fueron comparados mediante la existencia de los siguientes temas y la participación del alumno en cada una de ellas:

Sistema Operativo Base Corresponde a la existencia de archivos en los cuales basarse para la construcción del Sistema Operativo. Estos pueden ser archivos con cabeceras de funciones como también carpetas. Sólo corresponde a su estructura y no a código.

Sistema de Booteo Base Código correspondiente al iniciador del sistema operativo. Puede estar completo o no. En caso de que código se encuentre presente, éste debe escoger correctamente el Sistema Operativo a mostrar.

Kernel Base para Sistema Operativo Código que corresponde a un kernel base con posibilidades de ser modificado.

Stack Base para Sistema Operativo Sistema básico de manejo de memoria para ser utilizado por el Sistema Operativo y sus procesos[40].

Gestión de Páginas Físicas Código base para el principio de memoria virtual. La forma con la cual los procesos están repartidos en forma de páginas para ser cargados, ayudando al hardware de gestión de memoria y optimizando el espacio. [40]

Memoria Virtual Código que corresponde a la Memoria Virtual. Configuración que es necesaria cuando la memoria es ocupada en su totalidad y se requiere de algún método de mitigación para seguir utilizándola con ayuda de un almacenamiento secundario[40].

Memoria de Kernel Código que corresponde a la reserva de memoria asignada netamente al uso del Kernel del Sistema Operativo.

Manejo de Interrupciones y Excepciones Corresponde a la construcción de un sistema que permita interactuar con procesos que requieran necesariamente una interrupción del sistema, con la implementación de la lista IDT, lo mismo para el manejo de excepciones. Debe soportar tanto las interrupciones, las cuales son de origen externo, y las excepciones que son de origen interno[40].

Soporte de Multiprocesos Corresponde a la creación de código que permita la existencia de más de un proceso funcionando a la vez en el Sistema Operativo. Éste puede ser simétrico o asimétrico[28].

Procesos y Fork Código que corresponde a la utilización del concepto 'Proceso' para un trabajo. Además también de la creación del sistema Fork para la creación de procesos hijos.

Comunicación entre procesos Parte del sistema operativo que permita la interacción entre los procesos, involucrando los tópicos anteriores de Fork, Multiprocesos e Interrupciones y Excepciones.

File System Creación de un sistema que interactue con archivos. Código que sirva a nivel de usuario para crear, verificar e incluso usar archivos y directorios.

Llamadas a Sistema Código que corresponde a las aplicaciones que se encargan de realizar la comunicación hacia el kernel para realizar operaciones a petición del usuario.

La Interfaz Teclado Código que permita la interacción de la interfaz I/O de Teclado.

La Interfaz Mouse Código que permita interactuar con el Mouse en el Sistema Operativo.

La Interfaz VGA Código que abarca la utilización de una pantalla VGA, siendo ésto el manejo de colores, resoluciones y otros temas a fines de visualización.

La Terminal y Modo Usuario Corresponde a la creación de una Terminal que contenga aplicaciones seguras para el Sistema Operativo que tienen llamadas a sistema.

Servicio Network Configuraciones mediante código para la creación de la interacción del Sistema Operativo para su utilización mediante conexión de red.

Ideas libres del alumno Espacio para que el alumno pueda agregar lo que desee al Sistema Operativo trabajado.

Para cada tabla resolutiva sólo se adjuntarán los temas que abarca cada curso y su respectivo laboratorio en el cual se encuentra, también se mostrará como es la participación del alumno en cada uno de ellos.

Símbolo	Corresponde a
○	Implementado totalmente por el alumno
●	Proporcionado por el sistema
○●	Proporcionada una base, desarrollada por el alumno

Tabla 3.1: Símbolos para representar participación del alumno por cada tema.

Por otro lado, los cursos serán evaluados según su disponibilidad de información y su útlima fecha de actualización, para así tener una clara definición de cual curso se puede utilizar para nuestro propósito.

3.1. Cursos para comparar

3.1.1. Curso de Massachusetts Institute of Technology

El curso basado en la modificación del sistema operativo XV6 está disponible para su libre uso mientras que éste se le referencie su origen de procedencia. El código fue descargado y compilado según la documentación anexada en la Sección A.1 sin presentar inconveniente alguno. Se logra encontrar un código incompleto preparado para que se puedan realizar las experiencias listadas en documentos separados.

El curso además cuenta con la materia disponible (Reflejado en la Tabla 3.2) según el orden el cual es pasada teniendo una clara similitud entre lo pasado en clases y lo experimentado. Se da a conocer que el curso consta de horas de clases y horas de trabajo para las experiencias prácticas[22].

La última modificación conocida en el curso fue realizada el mes de Noviembre del año 2015 esencialmente en el código público git[22].

Laboratorio Tema	1	2	3	4	5	6	7
Sistema Operativo Base	●						
Sistema de Booteo Base	●						
Kernel Base para Sistema Operativo	●						
Stack Base para Sistema Operativo	●						
Gestión de Páginas Físicas		○●					
Memoria Virtual		○●					
Memoria de Kernel		○●					
Manejo de Interrupciones y Excepciones			○●				
Soporte de Multiprocesos				○●			
Procesos y Fork				○●			
Comunicación entre procesos				○●			
File system					○●		
La Interfaz VGA					○●		
La Interfaz Teclado					○●		
La Terminal y Modo Usuario					○●		
Servicio Network						○●	
Ideas libres del alumno							○

Tabla 3.2: Tabla comparativa de temas abordados por el curso de Massachusetts Institute of Technology

3.1.2. Curso de Carnegie Mellon University

Este curso es bastante completo, da a su disposición todo su material referente a lo pasado en clases, además de ofrecer presentaciones en ppt. El código se encuentra disponible sólo para realizar las primeras dos experiencias; para las demás se encuentran especialmente para uso de los estudiantes de su universidad.

Cuenta con dos clases presenciales semanalmente por 16 semanas y se enfoca netamente en el aprendizaje de Sistemas Operativos basados en el libro *Operating Systems: Principles and Practic* y las experiencias prácticas están contempladas para realizarse fuera de clases por lo que son de baja dificultad[1][7].

Como no se pudo indagar en el contenido de las experiencias 3 y 4 del curso, la Tabla 3.3 quedó incompleta y la evaluación fue realizada con la información disponible en las experiencias 1 y 2.

La última modificación conocida para este curso fue realizado el mes de Noviembre del año 2015 y fue principalmente en la actualización de los archivos pdf disponibles públicamente[7].

Laboratorio Tema	1	2	3	4
Inspección de lenguaje C	○●			
Sistema Operativo Base	●			
Sistema de Booteo Base		●		
Kernel Base para Sistema Operativo		●		
Stack Base para Sistema Operativo		●		
La Interfaz Teclado		○●		
La Interfaz Mouse		○●		
La Terminal y Modo Usuario		○		
Manejo de Interrupciones y Excepciones		○●		

Tabla 3.3: Tabla comparativa de temas abordados por el curso de Carnegie Mellon University

3.1.3. Curso de Standford University

El desarrollo de las experiencias prácticas en este curso se basa principalmente en la modificación del sistema operativo PintOS en su expresión más pequeña posible, de tal forma que los alumnos puedan hacerlo robusto. Dichas experiencias son de carácter acumulativo y están planificadas para ser desarrolladas fuera del transcurso de clases[38].

Cuenta con horarios de clases semanales y horarios de consulta para el desarrollo de las experiencias. El código se encuentra disponible y fue posible ser compilado según la documentación adjunta en la Sección A.3, encontrándonos con un kernel base para modificar, ya que sin modificar tenemos una terminal con operaciones básicas incompletas. Cada una de las experiencias toca un punto importante esencial que debe contener un kernel funcional, tal como lo podemos ver en la Tabla 3.4[38].

El curso presenta una última actualización para Diciembre del año 2009. [38]

Laboratorio Tema	0	1	2	3	4
Sistema Operativo Base		●	●	●	●
Comunicación entre procesos	○●				
Soporte de Multiprocesos		○●			
La Terminal y Modo Usuario			○●		
Memoria Virtual				○●	
File System					○●

Tabla 3.4: Tabla comparativa de temas abordados por el curso de Standford University

3.1.4. Curso de Harvard, School of Engineering and Applied Sciences

El curso cuenta con 5 experiencias prácticas separadas por tópicos según la Tabla 3.5 en las cuales se trabaja en base a la modificación del sistema *OS 161* el cual permite su modificación agregando código en lenguaje C. El curso cuenta con toda la información disponible de forma pública, incluso con repositorios especiales para ello para dar la oportunidad de que cualquier persona pueda aportar a su código[6].

Además cuenta con una máquina virtual que trabaja en VMWare la cual es posible descargar y contiene las herramientas necesarias para llegar y trabajar. Entonces, copiando

el repositorio git dentro de esta máquina, es posible correr el sistema operativo según lo documentado en la Sección A.5. El código además trae programas predefinidos que validan si la tarea realizada está trabajando correctamente[6].

En su página oficial es posible descargar la materia de forma ordenada además de las presentaciones en ppt para cada una de las clases semanales efectuadas. Las experiencias son realizadas fuera del horario de clases.

La última modificación conocida fue realizada en el código git el mes de Junio del año 2015[6].

Laboratorio Tema	0	1	2	3	4
Sistema Operativo Base	●				
Sistema de Booteo Base	●				
Kernel Base para Sistema Operativo	●				
Stack Base para Sistema Operativo	●				
Soporte de Multiprocesos		○●			
Procesos y Fork			○●		
Llamadas al Sistema			○●		
Memoria Virtual				○●	
File System					○●

Tabla 3.5: Tabla comparativa de temas abordados por el curso de Harvard, School of Engineering and Applied Sciences

3.1.5. Curso de University of Illinois at Chicago

El curso está enfocado directamente en la creación de un sistema operativo seguro, por lo mismo las experiencias prácticas realizadas están efectuadas en base al sistema operativo Ethos[31] que cumple con dicho propósito[30].

Las clases son de carácter online y sólo es necesario asistir los días de presentaciones de avances de las experiencias. Dichas clases se encuentran disponibles públicamente en formato de diapositivas.

Las experiencias prácticas 1 y 2 son las que se encuentran de forma pública siendo las otras tres experiencias privadas sólo para los alumnos de dicha universidad. Los tópicos representados en la Tabla 3.6 fueron encontrados gracias a que los documentos instructivos

para las experiencias se encuentran parcialmente disponibles (cuentan con el enunciado a desarrollar sin ningún tipo de guía para avanzar)[30].

La última actualización conocida de este curso fue realizada en Marzo del año 2012 el cual fue la fecha en la que se finalizó el curso. [30]

Laboratorio Tema	0	1	2	3	4	5
Sistema Operativo Base	●					
Sistema de Booteo Base	●					
Kernel Base para Sistema Operativo	●					
Stack Base para Sistema Operativo	●					
Llamadas al Sistema		○		○●		
Memoria Virtual		○	○●	○●		
Gestión de Páginas Físicas					○●	○●

Tabla 3.6: Tabla comparativa de temas abordados por el curso de University of Illinois at Chicago

3.1.6. Curso de University of California

El curso cuenta con clases lectivas especializadas en la construcción de un sistema operativo para ser utilizado como *webserver*. Se encuentra estructurado con horas de clases y horas prácticas de laboratorio, en las que se debe trabajar en ellas en conjunto con asistencia del profesor[16].

Los contenidos del curso se encuentran disponibles en su página web tanto la materia, las presentaciones en diapositivas y los documentos que guían en el trabajo de las experiencias, reflejado en la Tabla 3.7. El curso se encuentra bastante completo para el objetivo principal que buscan.

En cambio, el código a modificar no se encuentra disponible ya que fue retirado de los servidores de la universidad con lo que se hizo imposible realizar algún tipo de trabajo en él.

La última vez que se dictó el curso coincide con la fecha de su última actualización en los documentos, el cual es Mayo del año 2014[16].

Laboratorio Tema	0	0.5	1	2	3	4
Sistema Operativo Base	●					
Sistema de Booteo Base	●					
Kernel Base para Sistema Operativo	●					
Stack Base para Sistema Operativo	●					
Llamadas a Sistema		○●				
Gestión de Páginas Físicas		○●		○●		
La Interfaz VGA		○●			○●	
Soporte de Multiprocesos			○●			
Comunicación entre Procesos			○●	○●		
Memoria Virtual			○●			
Memoria de Kernel			○●			
Servicio Network			○●		○●	○●

Tabla 3.7: Tabla comparativa de temas abordados por el curso de University of California

3.1.7. Curso de University of Florida

El curso de Sistemas Operativos se complementa con experiencias prácticas basadas en la modificación del sistema operativo Xinu con el objetivo de mejorar y agregar distintos módulos a éste.

El curso no cuenta con un código especial, ya que se basa en el orden de aprendizaje del libro *Operating System Design: The Xinu Approach, Second Edition*[3] el cual trae su propio sistema, el cual fue probado y ejecutado según la documentación anexada en la Sección A.4.

La información del curso se limita a su existencia y no comparte sus documentos utilizados. Los tópicos indicados en la Tabla 3.8 son los indicados en la descripción del curso y no necesariamente son los que están incluidos en las experiencias prácticas[32].

La última actualización de la información del curso fue realizada en Abril del año 2015[32].

3.1.8. CROCOS 0.2

El sistema operativo CROCOS no es un curso pero está orientado para el aprendizaje, ya que su código se encuentra separado en etapas de construcción según los tópicos necesarios

Laboratorio Tema	Todos
Sistema Operativo Base	●
Sistema de Booteo Base	●
Kernel Base para Sistema Operativo	●
Stack Base para Sistema Operativo	●
Procesos y Fork	○●
Comunicación entre procesos	○●
Gestión de Páginas Físicas	○●
Memoria Virtual	○●
Memoria de Kernel	○●
La Terminal y Modo Usuario	○●
File System	○●

Tabla 3.8: Tabla comparativa de temas abordados por el curso de University of Florida

a abordar. El sistema no se encuentra completo y se nota claramente que fue abandonado por la comunidad debido a que su última actualización se realizó en Abril del año 2013 luego de haber presentado actualizaciones periódicas de no más de 2 días de diferencia en su repositorio[11].

Como no es un curso no cuenta con material de estudio ni con presentaciones disponibles. Los temas presentados en la Tabla 3.9 son los que actualmente abarca[11].

Fases Tema	1	2	3
Sistema Operativo Base	●		
Sistema de Booteo Base	●		
Kernel Base para Sistema Operativo	●		
Stack Base para Sistema Operativo	●		
Soporte de Multiprocesos		●	
Procesos y Fork		●	
File System			●

Tabla 3.9: Tabla comparativa de temas abordados por el SO CROCOS 0.2

3.1.9. JamesM's kernel development tutorials

A pesar de que no es un curso, se considera como posibilidad por ser un tutorial que construye un kernel funcional. Está basado en el trabajo de Bran explicado en la Sección 2.3.2.

Al realizar el tutorial por completo el resultado es un sistema operativo que contiene operaciones básicas de ejecución hasta llamadas a sistema, tal como se muestra en la Tabla 3.10[25].

El tutorial es desarrollado totalmente por el alumno y es explicado paso a paso, dando explicaciones de funcionalidad y estructura.

El desarrollo de este material fue finalizado en Abril del 2008, teniendo aún vigencia para los equipos actuales. De hecho, fue compilado y probado según la documentación adjunta en la Sección A.2[25].

Tema	Realizado por
Sistema Operativo Base	○
Sistema de Booteo Base	○
Kernel Base para Sistema Operativo	○
Stack Base para Sistema Operativo	○
La Interfaz VGA	○
Gestión de Páginas Físicas	○
Manejo de Interrupciones y Excepciones	○
Memoria Virtual	○
Memoria de Kernel	○
File System	○
Comunicación entre procesos	○
Llamadas a Sistema	○

Tabla 3.10: Tabla comparativa de temas abordados por el Tutorial de JamesM

3.1.10. Bran's Kernel Development

Al igual que el Tutorial realizado por James, fue considerado dentro de este estudio por ser un tutorial que crea un kernel funcional.

Éste tutorial cuenta con menos temas abordados que el tutorial de James lo se ve reflejado en la Tabla 3.11, pero cuenta con otros temas de reconocimiento de interfaz gráfica y manejos de interfaces de teclado[12].

Como es un tutorial, el desarrollo del código es realizado por completo. Podemos ver un ejemplo de ejecución de éste en la documentación generada en la Sección A.6.

Por último, el tutorial fue finalizado y publicado en Marzo del 2007[12].

Tema	Realizado por
Sistema Operativo Base	○
Sistema de Booteo Base	○
Kernel Base para Sistema Operativo	○
Stack Base para Sistema Operativo	○
La Interfaz VGA	○
Gestión de Páginas Físicas	○
Manejo de Interrupciones y Excepciones	○
La Interfaz Teclado	○

Tabla 3.11: Tabla comparativa de temas abordados por el Tutorial de Bran

3.2. Resumen, Tablas comparativas

Para un mejor orden en las tablas 3.13, 3.14, 3.15, 3.16 y 3.17, se decidió representar cada curso por una sigla la cual puede representar el nombre del curso implementado como también el sistema operativo que modifican durante el curso. Dichas siglas están ordenadas en la tabla 3.12.

Curso	Sigla
Curso de Massachusetts Institute of Technology	xv6
Curso de Carnegie Mellon University	15-410
Curso de Standford University	PintOS
Curso de Harvard, School of Engineering and Applied Sciences	CS 161
Curso de University of Illinois at Chicago	Ethos
Curso de University of California	CS194-24
Curso de University of Florida	Xinu
CROCOS 0.2	CrocOS
JamesM's kernel development tutorials	James
Bran's Kernel Development	Bran

Tabla 3.12: Representación de los cursos en siglas para las siguientes tablas.

3.2.1. Comparación con respecto a los contenidos ofrecidos

Cada uno de los cursos fue evaluado según su disponibilidad de contenido. En la Tabla 3.13 podemos ver una clara comparación entre los cursos que ofrecen más contenidos independiente de su disponibilidad de éstos.

	XV6	CS194-24	James	PintOS	Xinu	CS 161	EthOS	Bran	15-410	CrocOS
Sistema Operativo Base	●	●	○	●	●	●	●	○	●	●
Sistema de Booteo Base	●	●	○	●	●	●	●	○	●	●
Kernel Base para Sistema Operativo	●	●	○	●	●	●	●	○	●	●
Stack Base para Sistema Operativo	●	●	○	●	●	●	●	○	●	●
Gestión de Páginas Físicas	○●	○●	○		○●		○●	○		
Memoria Virtual	○●	○●	○	○●	○●	○●	○●			
Memoria de Kernel	○●	○●	○		○●		○●	○		
Manejo de Interrupciones y Excepciones	○●		○	○				○	○●	
Soporte de Multiprocesos	○●	○●		○●		○●				●
Procesos y Fork	○●			○●	○●	○●				●
Comunicación entre Procesos	○●	○●	○	○●	○●					
File System	○●		○	○●	○●	○●				●
Llamadas al Sistema		○●	○			○●	○●			
La Interfaz Teclado	○●			○				○	○●	
La Interfaz Mouse									○●	
La Interfaz VGA	○●	○●	○							
La Terminal y Modo Usuario	○●				○●					
Servicio Network	○●	○●								
Ideas libres del alumno	○									

Tabla 3.13: Tabla comparativa de temas abordados por todos los cursos.

3.2.2. Comparación con respecto a su disponibilidad y última actualización

Las tablas 3.14 y 3.15 muestran la disponibilidad de información que ofrecen los cursos hacia el público en general. Ésto considera un código existente y compilable, materia de la cual es posible leer por completo el curso, presentación en proyecciones con cada clase implementada y la existencia de documentos guías que sirvan de instructivos para realizar las experiencias prácticas con respecto a la modificación de un kernel real. Por último, se registra la fecha en la cual se realizó una última modificación en el curso con respecto a su código e información general.

	CS 161	XV6	PintOS	CS194-24	15-410
Código compilable	SI	SI	SI	NO	Parcial
Materia del curso incluido	SI	SI	SI	SI	SI
Presentaciones en proyecciones	SI	NO	NO	SI	SI
Documentos guías para experiencias	SI	SI	SI	SI	SI
Fecha última actualización	06-2015	11-2015	12-2009	05-2014	11-2015

Tabla 3.14: Tabla comparativa de disponibilidad de contenidos (1).

	James	Bran	Xinu	EthOS	CrocOS
Código compilable	SI	SI	SI	Parcial	SI, incompleto
Materia del curso incluido	SI	SI	NO	NO	NO
Presentaciones en proyecciones	NO	NO	NO	SI	NO
Documentos guías para experiencias	NO	NO	NO	Solo Enunciados	NO
Fecha última actualización	04-2008	03-2007	04-2015	03-2012	04-2013

Tabla 3.15: Tabla comparativa de disponibilidad de contenidos (2).

3.2.3. Comparación con respecto a los tiempos necesarios para realizar los cursos

Cada uno de los cursos que se encuentran ya implementados en otras universidades cuentan con clases estructuradas para las estructuras según sus intereses a explicar, la tabla 3.16 resume la cantidad de horas estimadas ocupadas para cada una de ellas. Cabe mencionar que las horas son estimadas y no necesariamente representan la realidad.

	XV6	15-410	PintOS	CS 161	Ethos	CS194-24	Xinu
Semanas de duración (Semanas)	16	16	10	16	-	17	15
Sesiones de clase por semana (Sesiones)	2	2	3	2	-	2	2
Horas/sesiones de taller por semana	-	-	1	-	-	-	-
Horas estimadas de dedicación adicional (semanas)	4	3	15	2	-	8	-
Horas promedio dedicadas por taller	9	7.5	25	10	-	20	-

Tabla 3.16: Horas estimadas para todos los cursos investigados implementados en otras universidades

3.2.4. Comparación con respecto al tamaño del kernel a modificar

Lo siguiente es una comparación con respecto al tamaño en KB del sistema operativo que contienen los cursos a los cuales se le es posible compilar su código. La idea de realizar esta medición es para analizar de alguna forma la complejidad que podría tener al momento de abordar a realizar dicho curso. La tabla 3.17 refleja los datos obtenidos para cada uno de ellos. Notamos como es que el sistema operativo Xinu tiene un tamaño bastante diferenciado del resto, y esto es porque Xinu es un sistema operativo ya utilizado en dispositivos, más recientemente en dispositivos portátiles como las Raspberry[39].

Curso	Tamaño [KB]
CS-161	560
Xv6	291
PintOS	244
Xinu	7334
James	366
Brand	110

Tabla 3.17: Tabla comparativa de tamaños de sistemas operativos de cursos con código compilable

3.3. Análisis de los datos

Llegado a este punto nos queda comenzar a trabajar con los datos obtenidos. Se necesita converger a uno de los cursos evaluados para poder cumplir el objetivo de adaptar un curso para el Departamento de Informática.

Se utilizan distintos discriminantes para poder escoger de mejor forma el curso que se tomará a futuro, y se escogieron las más relevantes, tales son:

1. Si posee un código compilable para trabajar en él.

2. Si su actividad de actualización es reciente.

3. Cuenta con libro guía en cual estudiar las experiencias.

4. Cuenta con experiencias ya realizadas.

3.3.1. Características de su código

El objetivo de esta variable es conocer si los cursos cuentan con un código el cual se pueda descargar, compilar y ejecutar para simular su sistema operativo, claramente con la idea de seguir recompilando luego de realizar las modificaciones pertinentes.

Con ello, y gracias a las Tablas 3.14 y 3.15 escogeremos sólo a los sistemas operativos que cuenten con ello.

Curso	Código Compila
Curso de Massachusetts Institute of Technology	SI
Curso de Carnegie Mellon University	NO
Curso de Standford University	SI
Curso de Harvard, School of Engineering and Applied Sciences	SI
Curso de University of Illinois at Chicago	NO
Curso de University of California	NO
Curso de University of Florida	SI
CROCOS 0.2	NO
JamesM's kernel development tutorials	SI
Bran's Kernel Development	SI

Tabla 3.18: Extracto de Tablas 3.14 y 3.15 con respecto a compilación.

Por lo que nos quedamos con solo seis de los diez cursos propuestos, varios de ellos no pueden ser utilizados por privacidad de su código u otros no contaban con los requisitos de ser compilables.

3.3.2. Última actividad conocida

Esta variable nos indica si el curso en cuestión fue modificado o cuenta con actividades con fechas de no más de un año hasta la actualidad. Con esto aprovechamos de utilizar características que sólo las últimas tecnologías nos otorgan, como mejores programas de debug, simuladores y compiladores.

Aquí notamos que los tutoriales indicados como alternativas de aprendizaje quedaron atrás a causa de que usan tecnologías del año 2007. Algo similar ocurre con el curso de Standford University ya que no presentaba actividad del año 2009.

Curso	Actualización en el último año
Curso de Massachusetts Institute of Technology	SI
Curso de Standford University	NO
Curso de Harvard, School of Engineering and Applied Sciences	SI
Curso de University of Florida	SI
JamesM's kernel development tutorials	NO
Bran's Kernel Development	NO

Tabla 3.19: Extracto de Tablas 3.14 y 3.15 con respecto a última actualización.

3.3.3. Existencia de libro guía

La idea de considerar esta variable como importante, es que el curso cuente con un libro en el cual sea posible utilizar como referencias para estudiar sobre el desarrollo de las experiencias a realizar durante el semestre, por lo que se considerará las que sí lo posean.

Curso	Cuenta con libro Guía
Curso de Massachusetts Institute of Technology	SI [29]
Curso de Harvard, School of Engineering and Applied Sciences	NO
Curso de University of Florida	SI [3]

Tabla 3.20: Cursos considerados que cuentan con libro guía.

Entre los tres cursos disponibles hasta el momento, el que no cuenta con un libro guía es el curso de Harvard a pesar de que sí es posible conseguir de forma libre sus presentaciones de diapositivas. Se consideró más importante debido a que, con los libros respectivos de los dos cursos restantes, se encuentra un orden en el cual es posible realizar una enseñanza estructurada para nuestra universidad.

3.3.4. Experiencias aplicadas en sus cursos

Esta variable fue considerada una de las importantes al comienzo del estudio. Consiste en conocer si el curso en cuestión posee experiencias prácticas ya redactadas y estructuradas en un orden en el cual cualquier interesado en el curso pueda desarrollar para su bien propio. Se considerará el curso que sí cuente con ellas.

En este caso, el curso de University of Florida estructuró el curso gracias al libro

Curso	Cuenta con experiencias
Curso de Massachusetts Institute of Technology	SI
Curso de University of Florida	NO

Tabla 3.21: Cursos considerados que cuentan con experiencias prácticas pre-construidas.

de Xinu[3] y las experiencias, las cuales no estaban disponibles, seguían el orden de los capítulos de éste. En cambio, para el curso de Massachusetts, cuenta con siete experiencias ordenadas de forma incremental.

3.3.5. Conclusiones luego del análisis

Finalmente, el curso que cumple con todas estas variables corresponde al implantado en Massachusetts Institute of Technology utilizando el sistema operativo Xv6. En la Tabla 3.22 vemos un resumen de lo considerado en el proceso de selección.

Curso	Código Compila	Actualización en el último año	Cuenta con libro Guía	Cuenta con experiencias
Curso de Massachusetts Institute of Technology	SI	SI	SI	SI
Curso de Carnegie Mellon University	NO	–	–	–
Curso de Standford University	SI	NO	–	–
Curso de Harvard, School of Engineering and Applied Sciences	SI	SI	NO	–
Curso de University of Illinois at Chicago	NO	–	–	–
Curso de University of California	NO	–	–	–
Curso de University of Florida	SI	SI	SI	NO
CROCOS 0.2	NO	–	–	–
JamesM's kernel development tutorials	SI	NO	–	–
Bran's Kernel Development	SI	NO	–	–

Tabla 3.22: Resumen, Discriminación de Cursos.

4 | Implementación del Curso

El curso que se utilizará como base según la conclusión obtenida por el capítulo 3 es el curso Curso de Massachusetts Institute of Technology el cual cuenta con una licencia de atribución CC BY 3.0 US[4] con lo que es posible la utilización de su contenido y su modificación mientras exista un reconocimiento de su origen.

Como ya se había comentado, este curso trata el tema de la enseñanza de Sistemas Operativos mediante la experimentación práctica de modificar uno, en este caso se utiliza Xv6 el cual se le realizan aportes a sus módulos para así mejorar su kernel y cumplir con más funciones. Dicho trabajo está separado en 7 laboratorios prácticos de los cuales 5 son obligatorios y los últimos opcionales.

Para el curso que se desea implementar, contará con dos horas lectivas semanales durante 16 semanas, las cuales se estructurarán solo utilizando 15 de ellas y será de tal forma que la primera hora sea de clase en donde se repasarán los elementos necesarios para realizar la experiencia correspondiente a esa semana y la segunda hora se utilizaría para trabajar en las experiencias con asistencia del profesor y/o ayudantes.

En el anexo B encontramos el desarrollo del curso tal como lo ofrece actualmente el MIT. La adaptación que se hará con respecto a lo visto ahí será que solo se considerarán cinco laboratorios distribuidos durante el semestre. Un punto clave a considerar es que el código con las respuestas a los desafíos planteados en las experiencias se encuentra disponible públicamente, por lo que se necesita de un método para solucionar ésto.

4.1. Curso modificado para el Departamento de Informática

El curso cuenta con un repositorio git con un esqueleto del sistema operativo Xv6 el cual está separado por ramas según el laboratorio a realizar. El código está construido de forma evolutiva con el objetivo de ir reutilizando lo realizado en las experiencias anteriores.

El código se encuentra en:

`https://github.com/csfuente/LabsXv6/tree/master`

Las instrucciones para el uso de este repositorio se encontrarán en cada uno de los documentos instructivos que serán entregados para cada una de las experiencias.

Las experiencias están preparadas para ser desarrolladas individualmente o en grupo de a dos personas. Cada grupo debe generar su propio repositorio git en donde debe realizar su entrega (idealmente utilizando los servidores git del Departamento de Informática).

Las experiencias se encuentran adjuntas en el anexo digital como también una plantilla base para el programa de la asignatura.

4.1.1. Objetivos

1. Crear y editar código que permita ser utilizado en el kernel de un sistema operativo.
2. Identificar los distintos módulos y su relación con las funcionalidades del sistema operativo.
3. Experimentar el proceso de construcción de un sistema operativo simple.

4.1.2. Contenidos

Al igual que las primeras cinco experiencias del curso del MIT, los contenidos aplicados en las experiencias del curso son:

1. Booteo del sistema operativo

2. Manejo de memoria
3. Ambiente de usuario
4. Sistema de multitareas
5. File system, spawn y terminal

4.1.3. Evaluaciones

Se trabajarán 5 experiencias en las cuales abarcarán los temas representados en la sección 4.1.2 y se aplicará un bonus de un 5 % para quienes deseen aumentar su nota final. Dicha bonificación consiste en agregar al sistema operativo algo nuevo el cual se demuestre una nueva funcionalidad, no es necesario que funcione perfectamente.

El curso contará con cinco evaluaciones las que se deben realizar posterior a la fecha de entrega de cada una de las experiencias. Cada una de las evaluaciones se centrarán directamente en la experiencia realizada según los tópicos resueltos en éstos.

Las ponderaciones están representadas en la tabla 4.1 las cuales cada una cuenta con una nota entre 0 a 100 y están planificadas equitativamente durante el semestre.

	Ponderación
Experiencia 1	8 %
Experiencia 2	13 %
Experiencia 3	13 %
Experiencia 4	13 %
Experiencia 5	13 %
Promedio Evaluaciones	40 %
Bonus	5 %

Tabla 4.1: Ponderaciones de curso propuesto.

4.1.4. Programación

Semana	Clase 1	Clase 2
1	- Clase introductoria - Clase lectiva para Experiencia 1 - Conocimiento de herramientas a utilizar - Correr el sistema operativo - Direcciones físicas, BIOS, Kernel, Stack.	- Desarrollo de Experiencia 1
2	- Desarrollo de Experiencia 1	- Desarrollo de Experiencia 1
3	- Clase lectiva para Experiencia 2 - Manejo de memoria física, memoria virtual, direcciones de memoria de kernel - Entrega de Experiencia 1	- Desarrollo de Experiencia 2
4	- Desarrollo de Experiencia 2	- Desarrollo de Experiencia 2
5	- Desarrollo de Experiencia 2	- Desarrollo de Experiencia 2
6	- Clase lectiva para Experiencia 3 Parte A - Modo usuario y manejo de excepciones - Entrega de Experiencia 2	- Desarrollo de Experiencia 3
7	- Desarrollo de Experiencia 3	- Desarrollo de Experiencia 3
8	- Clase lectiva para Experiencia 3 Parte B - Fallas de páginas, excepciones de punto de quiebre, llamadas a sistema.	- Desarrollo de Experiencia 3
9	- Desarrollo de Experiencia 3	- Desarrollo de Experiencia 3
10	- Clase lectiva para Experiencia 4 Parte A - Soporte de multiprocesos, multitasking - Entrega de Experiencia 3	- Desarrollo de Experiencia 4
11	- Clase lectiva para Experiencia 4 Parte B - Soporte para Copy-on-Write en Fork	- Desarrollo de Experiencia 4
12	- Clase lectiva para Experiencia 4 Parte C - Comunicación entre procesos	- Desarrollo de Experiencia 4
13	- Clase lectiva para Experiencia 5 - FileSystem y Terminal	- Desarrollo de Experiencia 5
14	- Desarrollo de Experiencia 5	- Desarrollo de Experiencia 5
15	- Entrega de Experiencia 5	- Evaluaciones Tardías

Tabla 4.2: Schedule para 16 semanas de clases.

5 | Conclusiones

Luego de la investigación previa se logró concretar la construcción de un curso para el Departamento de Informática. El curso proporcionado por Massachusetts Institute of Technology terminó siendo la base para trabajar en este nuevo curso, todo gracias a su libre disposición de compartir el curso gracias a los derechos que ellos le otorgaron[27] el cual permite reutilización de código mientras exista una referencia de su origen. Con ello, se pudo tomar la información de sus laboratorios y se pudo construir un modelo similar y completo en español basado en su sistema operativo Xv6 y su kernel JOS.

El código fue clonado y se le realizaron unas pequeñas modificaciones de tal forma que funcione similar al original, de esta forma por cada nueva experiencia se copien archivos de una rama nueva en git.

Se construyó un plan de trabajo para un curso de 16 semanas en donde es posible la elaboración de cinco experiencias más cinco pruebas que contienen la materia abarcada por cada una de las experiencias para así asegurar que los alumnos tienen los conocimientos posterior a su realización, además de contar con clases presenciales en las cuales se introduce la materia para cada uno de los entregables a realizar.

Las experiencias construidas para el curso cumplen con el objetivo de que sean evolutivas. Se demostró que, mientras que se desarrollan las experiencias 2 y 3 por ejemplo, son necesarias las funciones creadas en la experiencia 1.

Los estudiantes que cursen esta asignatura terminarían capaces de describir cuales son los bloques básicos que componen un sistema operativo, conocer y explicar a groso modo como es la administración de la memoria, saber como funciona un sistema de archivos simple y tener las nociones básicas de la optimización de un sistema operativo como tal, lo que transforma esta posible asignatura en algo relevante para los alumnos que estén

interesados en este campo.

Ahora tan solo basta formalizar este contenido y plantearlo en un curso real para que los alumnos puedan alcanzar los conocimientos deseados de la construcción de un sistema operativo.

Bibliografía

[1] Anderson, T., and Dahlin, M. *Operating Systems: Principles and Practice*. Recursive Books, 2012. 3.1.2

[2] Carter, P. A. *PC Assembly Language*, November 2003. B.1.2

[3] Comer, D. *Operating System Design: The Xinu Approach, Second Edition*. Chapman and Hall/CRC, 2015. 2.1.7, 3.1.7, **??**, 3.3.4

[4] creativecommons. Attribution 3.0 united states (cc by 3.0 us). http://creativecommons.org/licenses/by/3.0/us/, Dec. [Acceso Diciembre de 2015]. 4

[5] creativecommons. Creative commons atribución-nocomercial 2.0 chile (cc by-nc 2.0 cl). https://creativecommons.org/licenses/by-nc/2.0/cl/, July. [Acceso Julio de 2015]. 2.1.3

[6] creativecommons. Cs 161: Operating systems. http://www.eecs.harvard.edu/~margo/cs161/index.html, July. [Acceso Julio de 2015]. 2.1.4, 2.1.4, 2.1.4, 3.1.4, A.5

[7] Dave Eckhardt, DJ Begos, C. W. C. L. T. C. B. M. M. S. J. Z. 15-410, operating system design and implementation. https://www.cs.cmu.edu/~410/, July. [Acceso Julio de 2015]. 2.1.2, 2.1.2, 2.1.2, 3.1.2

[8] David Bafumba-Lokilo, E. P. d. M. The xinu page. http://www.xinu.cs.purdue.edu/, Oct. [Acceso Octubre de 2015]. 2.1.7

[9] de Informática, D. Contenidos de asignatura sistemas operativos en utfsm. http://wiki.inf.utfsm.cl/index.php?title=Sistemas_Operativos, July 2012. [Acceso Julio de 2015]. 1

[10] de Informática, D. Malla curricular de ingeniería civil en informática utfsm. [Online], 2013. http://www.inf.utfsm.cl/images/Mallas/malla-ici.png. 1

[11] Duranceau, G. The crocos kernel. http://crocos.sourceforge.net/, July. [Acceso Julio de 2015]. 2.2.1, 3.1.8

[12] Friesen, B. Bran's kernel development, a tutorial on writing kernels. http://www.osdever.net/bkerndev/index.php, July. [Acceso Julio de 2015]. 2.3.2, 3.1.10

[13] GNU.ORG. What is the gnu hurd? http://www.gnu.org/software/hurd/s, July 2015. [Acceso Julio de 2015]. 1

[14] INTEL. Intel 64 and ia-32 architectures software developer's manual. volume 3a: System programming guide, part 1. [Online]. https://pdos.csail.mit.edu/6.828/2014/readings/ia32/IA32-3A.pdf. B.3.1

[15] KERNIGHAN, B. *The C programming language*. Prentice Hall, Englewood Cliffs, N.J, 1988. B.1.2

[16] KUBIATOWICZ, J. Cs194-24: Advanced operating systems structures and implementation. http://www.cs.berkeley.edu/~kubitron/courses/cs194-24-S14/index_overview.html, July. [Acceso Julio de 2015]. 2.1.6, 2.1.6, 3.1.6

[17] LEWISCHENG MS. mit-jos. [Online]. https://github.com/lewischeng-ms/mit-jos. B, B.2.2, B.2.3

[18] LIONS, J. *Lions' Commentary on Unix*. Peer to Peer Communications/ Annabook, 1977. 2.1.1

[19] LOVE, R., ARE, S. H. W., LINUS, A. C., KERNELS, L. V. C. U., AND BEGIN, B. W. *Linux Kernel Development Second Edition*. Novell Press: Sams Publishing, 2005. 1

[20] MCLEAN, P. T. *Information Technology - AT Attachment with Packet Interface - 6 (ATA/ATAPI-6)*, December 2001. B.1.2

[21] MIT. Intel 80386 reference programmer's manual. [Online]. https://pdos.csail.mit.edu/6.828/2014/readings/i386/toc.htm. B.2.2, B.3.1

[22] MIT. 6.828: Operating system engineering. http://pdos.csail.mit.edu/6.828/2014/general.html, July 2014. [Acceso Julio de 2015]. 2.1.1, 3.1.1, A.1

[23] MIT. 6.828: Operating system engineering, lab 1: Booting a pc. https://pdos.csail.mit.edu/6.828/2014/labs/lab1/, Dec. 2014. [Acceso Diciembre de 2015]. B.1

[24] MIT. xv6, a simple, unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/xv6/, July 2014. [Acceso Julio de 2015]. 2.1.1

[25] MOLLOY, J. Jamesm's kernel development tutorials. http://www.jamesmolloy.co.uk/tutorial_html/, July. [Acceso Julio de 2015]. 2.3.1, 2.3.1, 3.1.9

[26] NAKULM95. Gitlab nakul (nick) malhotra / 451. [Online]. https://gitlab.cs.washington.edu/nakulm95/jos-15au/tree/lab1. B

[27] (OSI), O. S. I. The mit license (mit). http://opensource.org/licenses/mit-license.php, July. [Acceso Julio de 2015]. 2.1.1, 5

[28] RAMOS, M. *SISTEMAS OPERATIVOS MONOPUESTO*. Ciclos formativos. Paraninfo, 2010. 3

[29] Russ Cox, Frans Kaashoek, R. M. *xv6, a simple, Unix-like teaching operating system*, September 2014. 2.1.1, **??**

[30] Solworth, J. A. Cs 486: Secure operating system design and implementation. http://www.ethos-os.org/~solworth/cs486.html, July. [Acceso Julio de 2015]. 2.1.5, 2.1.5, 3.1.5

[31] Solworth, J. A. The ethos operating system. http://www.ethos-os.org/, July. [Acceso Julio de 2015]. 2.1.5, 3.1.5

[32] Sumi Helal, P. Cop 4600: Operating systems. http://www.cise.ufl.edu/~helal/classes/s15/, Oct. [Acceso Octubre de 2015]. 2.1.7, 3.1.7

[33] System, G. O. Gnu general public license version 3. http://www.gnu.org/licenses/gpl-3.0.en.html, July. [Acceso Julio de 2015]. 2.2.1

[34] Ubuntu, C. L. . F. Imagen iso centos 7. [Online], 2015. http://www.ubuntu.com/. A.1, A.2, A.3

[35] Underwood, B. B. Brennan's guide to inline assembly. http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html, Dec. 1996. [Acceso Diciembre de 2015]. B.1.1

[36] University, C. M. A.2 cmu license. http://www.gnu.org/software/hurd/gnumach-doc/CMU-License.html, July. [Acceso Julio de 2015]. 2.1.2

[37] University, L. S. J. Cs 140: Operating systems (spring 2014). http://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/index.php, July. [Acceso Julio de 2015]. 2.1.3, 2.1.3, 2.1.3

[38] University, L. S. J. Pintos documentation. http://web.stanford.edu/class/cs140/projects/pintos/pintos.html, July. [Acceso Julio de 2015]. 2.1.3, 3.1.3, A.3

[39] University, P. S. How to install xinu on vmware. [Online]. ftp://ftp.cs.purdue.edu/pub/comer/private/Xinu/How-to-install-Xinu-on-VMWare.pdf. 3.2.4, A.4

[40] Wolf, G. *Fundamentos de sistemas operativos*. 3

EX UMBRA
IN
SOLEM

A | Documentación

A.1. Instalación de Sistema Operativo XV6 para curso de Massachusetts Institute of Technology

Para realizar el procedimiento de prueba del repositorio de trabajo de XV6 del curso, se utilizó una máquina virtual con sistema operativo Ubuntu Desktop 15.10[34].

Lo primero es la instalación de las dependencias del sistema. para ello se debe abrir una terminal con permisos de administrador e ingresar los siguientes comandos:

```
sudo apt-get -y update
sudo apt-get -y upgrade
sudo apt-get -y update
sudo apt-get install -y git
```

Luego se procede a la instalación de QEMU, el cual virtualizará el sistema operativo con el que se trabajará.

```
sudo apt-get install -y qemu-kvm qemu virt-manager virt-viewer libvirt-bin
```

Por último sólo basta con clonar el repositorio git de XV6[22], compilar y ejecutar bajo QEMU.

```
git clone git://github.com/mit-pdos/xv6-public.git
cd xv6-public
make
make qemu
```

En este repositorio se encuentra el proyecto completo luego de realizar las experiencias que ofrece el curso.

A.2. Instalación de Sistema Operativo creado en JamesM's kernel development tutorials

Para probar el sistema operativo que se crea durante el tutorial otorgado por JamesM, es necesario instalar el emulador de éste, el cual en este caso es Bochs. Trabajamos bajo Ubuntu Desktop 15.10 x32[34]. Para ello procedemos a abrir una terminal con permisos de administrador y ejecutar los siguientes comandos:

```
sudo apt-get -y update
sudo apt-get -y upgrade
sudo apt-get -y update
sudo apt-get install -y git
sudo apt-get install -y bochs bochs-x bochs-sdl
```

Luego descargamos el archivo que contiene el código generado luego de realizar todo el tutorial, lo descomprimimos y lo ejecutamos bajo Bochs.

```
wget http://www.jamesmolloy.co.uk/downloads/user_mode.tar.gz
tar -xzvf user_mode.tar.gz
cd src
make clean
make
cd ..
mkdir /mnt2
./update_image.sh
./run_bochs.sh
```

Podemos apreciar los resultados del tutorial hasta el capítulo de "User Mode".

A.3. Instalación de Sistema Operativo PintOS para curso de Standford University

Toda la insformación del sistema operativo PintOS se encuentra en su página oficial[38] junto con su repositorio con el código para realizar las experiencias que ofrece el curso. Para ello necesitamos una máquina virtual x32, que en nuestro caso utilizaremos Ubuntu Desktop 15.10 x32[34]. En ella ejecutamos los comandos de actualización e instalación de dependencias. PintOS funciona tanto para ser utilizado con Bochs como por QEMU. Instalaremos ambas en esta ocasión.

```
sudo apt-get -y update
sudo apt-get -y upgrade
sudo apt-get -y update
sudo apt-get install -y git
sudo apt-get install -y bochs bochs-x bochs-sdl
sudo apt-get install -y qemu-kvm qemu virt-manager virt-viewer libvirt-bin
```

Luego descargamos el archivo comprimido con el código de PintOS.

```
wget http://www.stanford.edu/class/cs140/projects/pintos/pintos.tar.gz
tar -xzvf pintos.tar.gz
cd pintos/src
```

Aquí nos encontramos con varias carpetas las cuales separan por cada una de las experiencias a realizar. Tener en cuenta que este código se encuentra incompleto ya que está preparado para ser modificado.

Antes que todo, es necesario copiar las aplicaciones que sirven para realizar la compilación de los kernels. Para ello copiamos todos los archivos que se encuentran dentro de la carpeta *utils* en la carpeta */bin*.

```
scp -r utils/* /bin
```

Así contamos con el script llamado *pintos* que nos permitirá ejecutar las máquinas simuladas.

Por último, podemos realizar las pruebas necesarias para cada experiencia ya que se encuentran separadas en carpetas. Por ejemplo, si deseamos probar el código para la experiencia 1 de Threads será necesario realizar lo siguiente:

```
cd threads
make
pintos run alarm-multiple
continue
```

Vemos como un script de prueba se ejecuta en la experiencia a realizar.

A.4. Instalación de sistema operativo Xinu para curso de University of Florida

Xinu es posible ser ejecutado bajo VMWare gracias a que el mismo sistema ofrece máquinas virtuales pre-armadas. Con esta información procedemos a descargar dichas máquinas virtuales de la dirección:

```
ftp://ftp.cs.purdue.edu/pub/comer/private/Xinu/xinu-vm.tar.gz
```

Luego, se descomprimen los archivos y se continúan los pasos tal como se muestra en el manual adjunto al archivo de descarga[39].

Primero se importan ambas máquinas virtuales a VMWare, *xinu-vclient* y *xinu-vserver*, luego se procede a configurar las conexiones de red.

En xinu-vserver:

1. Entrar a las configuraciones de la máquina (Click derecho>Settings)

2. Seleccionar la primera tarjeta de red, luego colocar la opción *Custom: Specific virtual network* y seleccionar *VMnet9*

3. Seleccionar la segunda tarjeta de red, para ella escogemos *NAT: Used to share the host's IP address*.

4. Seleccionar *Ok* para guardar los cambios.

En xinu-vclient:

1. Entrar a las configuraciones de la máquina (Click derecho>Settings)

2. Seleccionar la primera tarjeta de red, luego colocar la opción *Custom: Specific virtual network* y seleccionar *VMnet9* (escogiendo la misma que la del servidor)

3. Seleccionar *Ok* para guardar los cambios.

Con esto ya tenemos conexión por red, ahora la configuramos para tener comunicación vía puerto Serial.

En xinu-vserver:

1. Entrar a las configuraciones de la máquina (Click derecho>Settings).

2. Click en *Add*.

3. Seleccionar *Serial Port* y luego *Next*.

4. Seleccionar *Output to named pipe* y luego *Next*.

5. Colocar un nombre para la comunicación, éste debe comenzar con \\.*pipe*\, seleccionar *This end is the server*, *The other end is a virtual machine* y *Connect at power on*.

6. Click en *Finish* para finalizar.

En xinu-vclient:

1. Entrar a las configuraciones de la máquina (Click derecho>Settings).

2. Click en *Add*.

3. Seleccionar *Serial Port* y luego *Next*.

4. Seleccionar *Output to named pipe* y luego *Next*.

5. Colocar un nombre para la comunicación, éste debe ser el mismo ingresado anteriormente, seleccionar *This end is the client*, *The other end is a virtual machine* y *Connect at power on*.

6. Click en *Finish*.

7. En las configuraciones del nuevo puerto Serial se debe activar la opción de *Yield CPU on poll*.

Finalmente nos queda correr las máquinas virtuales, lo haremos de la siguiente manera:

1. Encender *xinu-vserver* e ingresar con los datos:

```
User: xinu
Pass: xinurocks
```

2. Nos encontraremos con un archivo llamado *xinu-x86-vm.tar.gz* en la raíz, lo descomprimimos con:

```
tar -xzvf xinu-x86-vm.tar.gz
```

3. Ingresamos a la carpeta y nos posicionamos dentro de *compile*.

```
cd xinu-x86-vm/compile
```

4. Compilamos y lo dejamos como ejecutable.

```
make clean
make
cp xinu /srv/tftp
```

5. Ejecutamos minicom

```
sudo minicom
```

6. Finalmente ejecutamos la máquina virtual *xinu-vclient*, con ello obtenemos corriendo sin problemas Xinu en la terminal de minicom.

A.5. Instalación de Sistema Operativo CS 161 para el curso CS 161: Operating Systems de Harvard University

El curso otorga una máquina virtual base para realizar todo el trabajo que se necesita por delante. La máquina cuenta con Ubuntu 14.04.1 LTS y viene con las herramientas ya puestas en su lugar, esto incluye editores de texto, aplicaciones de compilación, aplicaciones de virtualización, control de versiones, sistemas de debug y más. Funciona bajo VMWare y es posible descargar acá:

```
http://www.eecs.harvard.edu/~margo/cs161/appliance161-2015.ova
```

Luego de tener la máquina virtual descargada, es necesario realizar la importación en VMWare para luego proceder a encender.

En ella nos queda descargar el proyecto para comenzar a desarrollar las experiencias dadas por el curso, por lo que, en la máquina virtual, ingresamos a una terminal e ingresamos la siguiente línea de comando:

```
git clone git://code.seas.harvard.edu/cs161/os161.git
```

Con esto ya podemos trabajar en las actividades según se indican en la página web del curso[6]. Finalmente, para probar que la tarea fue correctamente realizada, es necesario realizar los siguientes pasos:

1. Posicionarse en la base del repositorio git clonado y ejecutar los siguientes comandos para realizar las compilaciones necesarias (Con ASSTN la experiencia que se está realizando):

```
mkdir ~/tmp/root
./configure --ostree=$HOME/tmp/root
bmake
bmake install
```

```
cd kern/config
./config ASSTN
cd ../compile/ASSTN
bmake depend
bmake
bmake install
```

2. Luego de tener todo compilado, nos posicionamos en la raíz del kernel que dejamos en ~*tmp**root*, copiamos el archivo de configuración y ejecutamos el sistema.

```
cd ~/tmp/root
cp ~/cs161/sys161/sys161-2.0.2/sys161.conf.sample sys161.conf
sys161 kernel "tt1"
```

3. Ejecutar los comandos de prueba según la experiencia a necesitar.

A.6. Instalación de Sistema Operativo creado en Bran kernel tutorial

El proceso de ejecución de este sistema operativo es bastante similar al aplicado para ejecutar el de tutorial de Jamesm en la Sección A.2 con unas pequeñas diferencias que haremos notar. Cabe destacar que se utiliza la misma máquina virtual utilizada en la Sección A.2 por lo que se encuentran los archivos de éste.

1. Lo primero que debemos hacer es descargar el código y descomprimir los archivos.

```
wget http://www.osdever.net/bkerndev/bkerndev.zip
unzip bkerndev.zip
```

2. Nos posicionamos en el tutorial de James y copiamos los archivos *bochsrc.txt* y *run_bochs.sh* dentro de la carpeta *bkerndev/Source* recien descomprimida.

3. Nos posicionamos en la carpeta *bkerndev/Source* y ejecutamos:

```
sh build.bat
```

Ignorar los errores.

4. Clonar y renombrar el archivo *dev_kernel_grub.img* a *floppy.img*.

```
cp dev_kernel_grub.img floppy.img
```

5. Finalmente ejecutamos el script *run_bochs.sh*

```
sh run_bochs.sh
```

Con esto vemos el kernel simplista creado durante este tutorial.

EX UMBRA
IN
SOLEM

B | Desarrollo de Curso ofrecido por MIT

El objetivo principal es el desarrollo de las primeras 5 experiencias, las cuales en el orden dado conllevan a un trabajo incremental. Para poder ejecutar el emulador QEMU es necesario ya haber pasado por las explicaciones dadas en la documentación del Anexo A.1.

Los códigos fueron basados por trabajos ya realizados por alumnos que ya cursaron el curso y fueron verificados para el desarrollo de este trabajo[17][26].

B.1. Lab 1: Booting a PC

El primer laboratorio se separa en tres partes, la primera consta de familiarizarse con el lenguaje assembly, el emulador QEMU x86 y el procedimiento de prendido y booteo de un PC. La segunda parte se encarga de analizar el código de carga del kernel 6.828 el cual se encuentra en la carpeta *boot* en los archivos del laboratorio. Finalmente, la tercera parte analiza el template inicial del kernel 6.828 el cual lo llamaremos JOS, el cual reside en el directorio *kernel*[23].

El laboratorio en cuestión se encuentra en un repositorio git público el cual es posible ser clonado con el comando:

```
git clone https://pdos.csail.mit.edu/6.828/2014/jos.git lab
```

Los distintos laboratorios se encuentran en ramas (branch) del repositorio git, tan sólo basta con revisar cuales ramas contiene para luego saltar a la otra. Tener en cuenta que al

momento de pasar de un laboratorio a otro, es necesario mezclar el trabajo realizado al nuevo laboratorio antes de proceder con su elaboración.

```
git checkout -b lab2 origin/lab2
git merge lab1
```

En cualquier momento de la realización de la experiencia de laboratorio, es posible ejecutar un comando que evalúa si el trabajo realizado es correcto; dicho comando se debe ejecutar en la raíz y corresponde a:

```
make grade
```

El cual nos representará lo que está correcto y lo que no, junto con una nota entre 0 y 100.

B.1.1. Part 1: PC Bootstrap

El principal objetivo de esta primera parte es familiarizarnos con las herramientas que utilizaremos durante el curso. Para acercarnos a Assembly, se recomienda sólo leer al respecto[35], sin causar impacto en el kernel en construcción, ya que en ningún momento del curso se programará en dicho lenguaje, pero es importante conocer qué es lo que está haciendo.

Luego se presenta QEMU como la aplicación que emula por completo un PC, con la ventaja de que es posible analizar el procedimiento de éste con programas de debuggueo. Para utilizar QEMU en el proyecto es necesario posicionarse en la raíz y ejecutar:

```
make
make qemu
```

Lo que nos abrirá la aplicación con el sistema base que trae el laboratorio. Claramente no trae muchas cosas ya que uno es el que debe ir agregándole características.

Tener en cuenta que al momento de ejecutar el comando *make*, se genera un archivo en el directorio *obj/kernel* llamado *kernel.img*, el cual si lo ponemos en un dispositivo de memoria, es posible ejecutarlo en un computador real, pero no es recomendable porque

puede causar conflictos con el sistema de booteo que posea actualmente dicho computador, provocando que éste sea reemplazado por el que estamos probando, pudiendo causar problemas tan graves como la pérdida de toda la información.

Finalmente se realiza la introducción del sistema de debuggueo llamado *gdb*, éste viene preconfigurado para que sea posible su utilización enlazada con QEMU para poder ir revisando en vivo lo que sucede en nuestro kernel. Para ello es necesario realizar los siguientes pasos:

1. Lo primero es compilar el código que llevamos actualmente, tan sólo es necesario ejecutar el comando

   ```
   make
   ```

2. Luego debemos ejecutar lo compilado, con la diferencia de que ahora utilizaremos el comando

   ```
   make qemu-gdb
   ```

   Veremos una pantalla de QEMU en negro pausada.

3. Abrimos una nueva terminal, nos posicionamos en la carpeta raíz del usuario y creamos un archivo llamado *.gdbinit* con el siguiente contenido:

   ```
   add-auto-load-safe-path /*usuario*/lab/.gdbinit
   ```

   Debemos indicar la dirección en donde se encuentra el archivo *.gdbinit* ubucado en la raíz de la carpeta del laboratorio.

4. Nos posicionamos en la raíz del laboratorio y ejecutamos el comando

   ```
   gdb
   ```

Con esto es posible evaluar los lugares en donde los punteros están direccionados en la memoria para la ejecución inicial de booteo del sistema operativo. Como ejercicio se pide ejecutar comandos en dicha nueva terminal para analizar lo sucedido según lo explicado en las mismas indicaciones con respecto a las direcciones físicas del computador.

B.1.2. Part 2: The Boot Loader

Es una introducción a la estructura de Boot que se utiliza en el arranque de un sistema operativo. Para la experiencia, el sistema de booteo se encuentra programado en lenguaje Assembly y C en los archivos *boot/boot.S* y *boot/main.c* respectivamente, con ello, el primer ejercicio es tratar de conocer que es lo que está sucediendo en estos dos archivos, el material cuenta con lecturas especializadas para esta etapa[2][20].

Luego de comprender estos archivos podemos analizar el archivo ya compilado que se creó luego de la primera prueba realizada; el archivo llamado *obj/boot/boot.asm* contiene las instrucciones de memoria indicadas en los dos archivos. El código se encuentra comentado para facilitar su identificación de sus líneas. Una de las cosas muy claras que podemos apreciar es la configuración de los modos privados y reales además de los saltos de memoria que se utilizan al arranque del sistema.

De la misma manera podemos ver como quedó compilado el kernel JOS analizando el archivo *obj/kern/kernel.asm*, esto nos sirve para el proceso de debug. Por ejemplo, podemos indicar en que dirección de memoria el kernel pause su ejecución hasta que se le indique lo contrario, de esta forma analizar lo que se ejecuta por partes. Con esto aparece un nuevo ejercicio en el cual se solicita realizar pausas al sistema en la dirección 0x7c00 y con ello indicar en que parte del código de *boot/boot.S* corresponde dicha parte. Además de familiarizarse más con la terminal de gdb.

Por otro lado tenemos el código en lenguaje C del archivo *boot/main.c* el cual lo podemos comprender con conocimientos básicos de punteros. El curso recomienda el libro *The C Programming Language*[15] desde el capítulo 5.1 hasta 5.5. Con ello, se pide analizar para que son los primeros punteros encontrados en las líneas 1 a 6.

Posteriormente se indica la estructura de un archivo binario de formato ELF, como énfasis sólo se da a notar, para centrarnos en el curso, que se considerará como un archivo que contiene el código necesario para cargar información inicial. Para los archivos en lenguaje C esta información es incluida gracias a la línea de código que incluye el header *inc/elf.h*. Si analizamos el archivo, vemos como se instancializan las variables globales para el sistema. Para examinar la lista entera de nombres de variables de la cabecera, tamaños y

direcciones de las direcciones en el kernel, es posible ejecutar cualquiera de estas líneas de comando:

```
i386-jos-elf-objdump -h obj/kern/kernel
objdump -h obj/kern/kernel
```

En donde podemos observar:

```
obj/kern/kernel:     formato del fichero elf32-i386

Secciones:
Ind Nombre          Tamaño    VMA       LMA       Desp fich Alin
  0 .text           00001861  f0700000  00100000  00001000  2**4
                    CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata         00000730  f0101880  00101880  00002880  2**5
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab           000038b9  f0101fb0  00101fb0  00002fb0  2**2
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr        000018c9  f0105869  00105869  00006869  2**0
                    CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data           0000a300  f0108000  00108000  00009000  2**12
                    CONTENTS, ALLOC, LOAD, DATA
  5 .bss            00000644  f0112300  00112300  00013300  2**5
                    ALLOC
  6 .comment        0000002d  00000000  00000000  00013300  2**0
                    CONTENTS, READONLY
```

Podemos obtener más información si el proceso *objdump* le cambiamos la indicación por *-x*, con la cual nos muestra con más detalle las funciones que utiliza y en que posición de la memoria están guardadas. También tenemos *-f* el cual sólo nos muestra el tipo de arquitectura y la dirección de memoria que corresponde a su inicio.

Finalmente esta parte queda con un ejercicio de gdb, el cual consiste en analizar 8 palabras de memoria cuando inicializa la bios y cuando inicializa el kernel y comparar sus resultados.

B.1.3. Part 3: The Kernel

Parte del laboratorio en donde se escribe código en lenguaje C, se basa en la modificación del sistema de arranque del kernel, iniciando por su posicionamiento en la memoria. Lo primero es mapear los primeros 4 MB de la memoria física, la que es suficiente para ejecutar el arranque. Para ello se utiliza un directorio y tablas estáticas ubicadas en el archivo *kern/entrypgdir.c*. Por ahora solo basta con entender que es lo que hace, debemos entender que el archivo *kern/entry.S* traspasa esta tabla a memoria virtual para luego ser transformada en direcciones de memoria físicas. Con ello, el laboratorio realiza un ejercicio el cual consiste en ejecutar gdb en el sistema operativo y parar cada vez que se ejecute la línea *movl %eax, %cr0* y analizar en que posición de la memoria física se encuentra ejecutando.

Luego pasamos a la edición del código. Como primera tarea se pide modificar la funcionalidad de printf para que le sea posible imprimir por pantalla números octales; para ello se requiere analizar los archivos *kern/printf.c*, *lib/printfmt.c* y *kern/console.c* para realizar los cambios correspondientes. El código a cambiar se encuentra documentado en *lib/printfmt.c* y se cuenta con una aplicación que verifica si el procedimiento realizado se encuentra correctamente. El código a agregar es:

```
case 'o':
    num = getuint(&ap, lflag);
    base = 8;
    goto number;
```

El ejercicio final del laboratorio trata sobre la Pila (Stack). Vemos la forma en como el código escrito en lenguaje C contempla el manejo completo de una Pila x86. El primer ejercicio es encontrar entre todo el código el lugar en donde la pila se inicializa, esto es

el punto exacto de la memoria en que es convocada, como el kernel reserva su espacio y como finaliza este espacio.

Se da bastante énfasis en trabajar con las direcciones otorgadas por *eip* y *ebp* las cuales son el puntero base y el puntero de la pila. Se pide programar en lenguaje C una forma de leer la Pila y mostrar sus direcciones por pantalla de la forma:

```
Stack backtrace:
  ebp f0109e58  eip f0100a62  args 00000001 f0109e80 f0109e98 f01...
  ebp f0109ed8  eip f01000d6  args 00000000 00000000 f0100058 f01...
  ...
```

Para ello se edita el archivo *kern/monitor.c* en la función *mon_backtrace()* con lo siguiente:

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf)
{
  uint32_t ebp = read_ebp(), *ebpp, eip, i;
  struct Eipdebuginfo dbg;
  while (ebp > 0) {
    ebpp = (uint32_t *)ebp;
    eip = ebpp[1];
    cprintf("ebp %x eip %x args", ebp, eip);
    for (i = 2; i < 6; i++) {
      cprintf(" %08x", ebpp[i]);
    }
    debuginfo_eip(eip, &dbg);
    cprintf("\n\t%s:%d: ", dbg.eip_file, dbg.eip_line);
    for (i = 0; i < dbg.eip_fn_namelen; i++)
      cputchar(dbg.eip_fn_name[i]);
    cprintf("+%d\n", eip - dbg.eip_fn_addr);
    ebp = *ebpp;
  }
```

```
    return 0;
}
```

Luego, para los ejercicios siguientes, es necesario registrar esta función para que sea posible ejecutar desde JOS. Para ello editamos el archivo *kern/kdebug.c* y activamos la función *stab_binsearch()* agregándola en la línea 204 el siguiente código:

```
stab_binsearch(stabs, &lline, &rline, N_SLINE, addr);
if (lline > rline)
  return -1;
info->eip_line = stabs[lline].n_desc;
```

y por último agregamos el comando al monitor editando nuevamente el archivo *kern/monitor.c* pero ahora agregamos un elemento a la estructura inicial *commands[]* con la siguiente línea:

```
{ "backtrace", "Display backtrace", mon_backtrace },
```

Con esto, desde nuestro sistema operativo podemos ver las aplicaciones que se encuentran actualmente en la Pila.

Para finalizar, es importante realizar el comando *make grade* para verificar que todo se encuentra en orden. En caso de problemas entonces hay que revisar si el código se está ejecutando correctamente.

B.2. Lab 2: Memory Management

El objetivo principal de este laboratorio es la construcción completa de la gestión de memoria. Esta se separa en dos principales componentes: La memoria física y la memoria virtual. La memoria física es asignada por el kernel el cual la puede asignar o liberar según corresponda, para ello se utilizan unidades de 4096 bytes llamadas *páginas*. La memoria Virtual por otro lado mapea las direcciones usadas por el kernel y el sofftware del usuario en la memoria física.

Para trabajar en este laboratorio, es necesario realizar unos cambios en el repositorio git.

```
git add kern/kdebug.c kern/monitor.c lib/printfmt.c
git commit -m "Término lab 1"
git checkout -b lab2 origin/lab2
git merge lab1
```

B.2.1. Part 1: Physical Page Management

Es necesario conocer cuales son las partes libres y ocupadas físicas de la memoria RAM. En JOS se utiliza *MMU* para tener el registro de mapeo de las páginas.

Se pide modificar el archivo *kern/pmap.c* para escribir el sistema *MMU*. En ella se deben modificar las funciones *boot_alloc()*, *mem_init()*, *page_init()*, *page_alloc()* y *page_free()*. Luego de tenerlas implementadas, es posible probarlas con las funciones *check_page_free_list()* y *check_page_alloc()*. No hay mucha explicación de como se debe desarrollar todo, lo que es parte del objetivo del ejercicio. Las respuestas son:

```
static void * boot_alloc(uint32_t n)
{
  static char *nextfree;
  char *result;

  if (!nextfree) {
    extern char end[];
    nextfree = ROUNDUP((char *) end, PGSIZE);
  }
  if (n > 0) {
    uint32_t alloc_size = ROUNDUP(n, PGSIZE);
    result = nextfree;
    nextfree += alloc_size;
    if ((uintptr_t)nextfree >= 0xf0400000)
      panic("boot_alloc: out of memory");
  } else {
```

```
    result = nextfree;
  }
  return result;
}

void mem_init(void)
{
  uint32_t cr0;
  size_t n;

  i386_detect_memory();

  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);
  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
  page_init();
  check_page_free_list(1);
  check_page_alloc();
  check_page();
  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
  boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
                                          PADDR(bootstack), PTE_W);
  boot_map_region(kern_pgdir, KERNBASE, -KERNBASE/* ~KERNBASE + 1
                                                  */, 0, PTE_W);
  check_kern_pgdir();
  lcr3(PADDR(kern_pgdir));
  check_page_free_list(0);

  cr0 = rcr0();
  cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_MP;
```

```
  cr0 &= ~(CR0_TS|CR0_EM);
  lcr0(cr0);

  check_page_installed_pgdir();
}

void page_init(void)
{
  size_t i;
  page_free_list = NULL;
  for (i = 0; i < npages; i++) {
    if (i == 0 || !inMapAvail(&pages[i]) || isInIOHole(&pages[i])) {
      pages[i].pp_ref = 1;
      pages[i].pp_link = NULL;
    } else {
      pages[i].pp_ref = 0;
      pages[i].pp_link = page_free_list;
      page_free_list = &pages[i];
    }
  }
}

struct Page * page_alloc(int alloc_flags)
{
  struct Page *result = page_free_list;

  if (!result)
    return NULL;
  page_free_list = result->pp_link;
```

```
  if (alloc_flags & ALLOC_ZERO) {
    char *kva = page2kva(result);
    memset(kva, '\0', PGSIZE);
  }

  return result;
}

void page_free(struct Page *pp)
{
  assert(pp);
  assert(pp->pp_ref == 0);

  pp->pp_link = page_free_list;
  page_free_list = pp;
}
```

B.2.2. Part 2: Virtual Memory

Para esta parte es necesario realizar ciertas lecturas en el Manual de Intel 80386[21], más específico los capítulos 5 y 6 para familiarizarse con el modo protegido de la arquitectura de manejo de memoria (segmentación y traducción de páginas).

Luego, como ejercicio se pide utilizar GDB para inspeccionar el emulador QEMU y sus direcciones virtuales. Para ello se pueden utilizar comandos especiales para que GDB nos retorne dichas direcciones.

1. En QEMU, es posible ingresar la secuencia de botones *Ctrl-a c* para acceder al monitor de éste.

2. En el monitor de QEMU se puede utilizar el comando *xp* y en GDB el comando *x* para inspeccionar la memoria y así identificar cual es memoria física y cual virtual. (recomendable verificar en ambos lugares para verificar que es la misma información)

3. En el monitor de QEMU puede que se encuentre el comando *info pg* el cual muestra una tabla con las páginas actuales, incluyendo rangos de memoria mapeados, permisos y banderas. También se encuentra el comando *info mem* para saber que rangos de la memoria virtual están mapeados y con que permisos. Que se encuentren estos comandos depende de la versión de QEMU.

Después nos hacen una pregunta:

C type	Addres type
T*	Virtual
uintptr_t	Virtual
physaddr_t	Physical

Tabla B.1: Tabla para ejercicio Laboratorio 2 Parte 2.

Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr_t or physaddr_t?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

R: la variable *x* corresponde a un *uintptr_t* debido a que se utiliza la referencia de un puntero para que sea asignado a la variable. En caso de que *x* sea *physaddr_t* entonces éste puede tomar la dirección de memoria correspondiente pero no le es posible realizar modificaciones en caso de que ésta se encuentre en la memoria virtual. El kernel diferencia las direcciones de memoria sumando o restando respectivamente 0xf0000000. Para memoria física a memoria virtual debe sumar esa cantidad, para realizar el proceso inverso entonces se resta.

Como último ejercico de este tema se pide implementar en el archivo *kern/pmap.c* las funciones *pgdir_walk(), boot_map_region(), page_lookup(), page_remove()* y *page_insert()*. Dicho código corresponde al siguiente [17]:

```
pte_t * pgdir_walk(pde_t *pgdir, const void *va, int create)
```

```
{
  pte_t *vaddr;
  pte_t *pgtb;
  pde_t pde;
  struct Page *temp;
  pde=pgdir[PDX(va)];
  if(pde&PTE_P)
  {
    pgtb=(pte_t *)PTE_ADDR(pde);
    vaddr=KADDR((physaddr_t)(pgtb+PTX(va)));
    return vaddr;
  }
  if(!create)
  {
    return NULL;
  }
  else
  {
    if(page_alloc(&temp))
    {
      return NULL;
    }
  }
  pgtb=(pte_t *)page2pa(temp);
  pde=(physaddr_t)pgtb|PTE_W|PTE_P;
  pgdir[PDX(va)]=pde;
  memset(page2kva(temp), 0, PGSIZE);
  temp->pp_ref =1;
  vaddr=KADDR((physaddr_t)(pgtb+PTX(va)));
  return vaddr;
```

```
}

static void boot_map_region(pde_t *pgdir, uintptr_t va,
  size_t size, physaddr_t pa, int perm)
{
  assert(pgdir);
  assert(ROUNDDOWN(va, PGSIZE) == va);
  assert(ROUNDDOWN(size, PGSIZE) == size);
  assert(ROUNDDOWN(pa, PGSIZE) == pa);
  while (size >= PGSIZE) {
    pte_t *pte = pgdir_walk(pgdir, (void *)va, 1);
    *pte = pa | perm | PTE_P;
    va += PGSIZE;
    pa += PGSIZE;
    size -= PGSIZE;
  }
}

struct Page * page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
{
  pte_t *pte;
  pte=pgdir_walk(pgdir, va, 0);
  if((!pte) || !pte_store || (!*pte))
  {
    return NULL;
  }
  else
  {
    *pte_store=pte;
    return pa2page(PTE_ADDR(**pte_store));
  }
```

```
}

void page_remove(pde_t *pgdir, void *va)
{
  assert(pgdir);
  pte_t *pte;
  struct Page *page = page_lookup(pgdir, va, &pte);
  if (!pte || !(*pte & PTE_P))
    return;
  page_decref(page);
  *pte = 0;
  tlb_invalidate(pgdir, va);
}

int page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
{
  assert(pgdir);
  assert(pp);
  pte_t *pte = pgdir_walk(pgdir, va, 1);
  if (!pte)
    return -E_NO_MEM;
  physaddr_t pa = page2pa(pp);
  if (*pte & PTE_P) {
    if (PTE_ADDR(*pte) == pa)
      goto SUCCESS;
    page_remove(pgdir, va);
  }
  pp->pp_ref++;
  tlb_invalidate(pgdir, va);
SUCCESS:
  *pte = pa | perm | PTE_P;
```

```
  return 0;
}
```

Las funciones pueden ser probadas ejecutando *./grade-lab2* o utilizando la información adquirida anterior a estos códigos.

B.2.3. Part 3: Kernel Address Space

En esta parte se centra específicamente en la memoria que es utilizada por el kernel, la que es definida por la variable *ULIM* en el archivo *inc/memlayout.h* la cual actualmente reserva 256 MB de la memoria virtual.

El usuario sólo puede usar la memoria disponible fuera del rango utilizado por el kernel, es decir éste sólo tiene acceso a lectura a dicha memoria. Se es necesario iniciar el espacio especial del kernel para que todo funcione correctamente. Para ello es necesario nuevamente editar el archivo *kern/pmap.c* y completar la función *mem_init()*. Dicha función debe quedar de la siguiente manera [17]:

```
void mem_init(void)
{
  uint32_t cr0;
  size_t n;
  i386_detect_memory();
  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);
  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
  pages = (struct Page *)boot_alloc(npages * sizeof(struct Page));
  page_init();
  check_page_free_list(1);
  check_page_alloc();
  check_page();
  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
  boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
```

```
    PADDR(bootstack), PTE_W);
  boot_map_region(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
  mem_init_mp();
  check_kern_pgdir();
  lcr3(PADDR(kern_pgdir));
  check_page_free_list(0);
  cr0 = rcr0();
  cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_MP;
  cr0 &= ~(CR0_TS|CR0_EM);
  lcr0(cr0);
  check_page_installed_pgdir();
}
```

Finalmente se pide responder a las preguntas planteadas, las cuales resumen todo el proceso realizado durante esta experiencia.

B.3. Lab 3: User Environments

El objetivo de la experiencia es crear el ambiente para un sistema protegido de modo usuario que corra sin problemas. Lo ideal es que se active un único modo usuario, se carguen programas en él y se ejecuten, con tal de que éstos funcionen solo en su espacio y que se manejen con cuidado cada llamada al sistema necesario para éstos.

Para la experiencia 3 es necesario agregar los archivos correspondientes. En este trabajo los comandos para ello son los siguientes:

```
git checkout -b lab3 origin/lab3
git merge lab2
```

Se adjuntan más de 20 archivos nuevos al proyecto.

B.3.1. Part A: User Environments and Exception Handling

Lo primero es aprender sobre las nuevas estructuras que se presentan para trabajar. Para ello se de a conocer sobre la cantidad de ambientes que se pueden construir, la cual está definida por el número *NENV* que se encuentra en el archivo *inc/env.h* y su enlace con las variables *envs, curenv* y *env_free_list*.

Se da a conocer también la estructura *Env* con todos sus detalles, la cual se debe ocupar para los ejercicios siguientes.

El primer ejercicio es modificar nuevamente la función *mem_init()* para colocar y mapear los arreglos de *env*. Con lo que la función nos quedará como:

```
void mem_init(void)
{
  uint32_t cr0;
  size_t n;
  i386_detect_memory();
  kern_pgdir = (pde_t *) boot_alloc(PGSIZE);
  memset(kern_pgdir, 0, PGSIZE);
  kern_pgdir[PDX(UVPT)] = PADDR(kern_pgdir) | PTE_U | PTE_P;
  pages = (struct Page *)boot_alloc(npages * sizeof(struct Page));
  envs = (struct Env *)boot_alloc(NENV * sizeof(struct Env));
  page_init();
  check_page_free_list(1);
  check_page_alloc();
  check_page();
  boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
  boot_map_region(kern_pgdir, UENVS, PTSIZE, PADDR(envs), PTE_U);
  boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE,
    PADDR(bootstack), PTE_W);
  boot_map_region(kern_pgdir, KERNBASE, -KERNBASE, 0, PTE_W);
  mem_init_mp();
```

```
  check_kern_pgdir();
  lcr3(PADDR(kern_pgdir));
  check_page_free_list(0);
  cr0 = rcr0();
  cr0 |= CR0_PE|CR0_PG|CR0_AM|CR0_WP|CR0_NE|CR0_MP;
  cr0 &= ~(CR0_TS|CR0_EM);
  lcr0(cr0);
  check_page_installed_pgdir();
}
```

Luego de que son mapeados en la memoria, es necesario trabajar en las funciones de los ambientes del modo usuario. Para ello y como ejercicio 2 se pide modificar en archivo *kern/env.c* para completar las funciones *env_init(), env_setup_vm(), region_alloc(), load_icode(), env_create()* y *env_run()* en ese orden, lo que quedaría como:

```
void env_init(void)
{
  int i;
  for (i = 0; i != NENV - 1; ++i) {
    envs[i].env_id = 0;
    envs[i].env_link = &envs[i + 1];
  }
  envs[NENV - 1].env_link = NULL;
  env_free_list = &envs[0];
  env_init_percpu();
}

static int env_setup_vm(struct Env *e)
{
  int i;
  struct Page *p = NULL;
  if (!(p = page_alloc(ALLOC_ZERO)))
```

```
    return -E_NO_MEM;
  e->env_pgdir = page2kva(p);
  for (i = 0; i != PDX(UTOP); ++i)
    e->env_pgdir[i] = 0;
  for (; i != NPDENTRIES; ++i)
    e->env_pgdir[i] = kern_pgdir[i];
  p->pp_ref++;
  e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
  return 0;
}

static void region_alloc(struct Env *e, void *va, size_t len)
{
  uintptr_t addr = ROUNDDOWN((uintptr_t)va, PGSIZE);
  uintptr_t end = ROUNDUP((uintptr_t)va + len, PGSIZE);
  while (addr < end) {
    struct Page *page = page_alloc(0);
    if (page == NULL)
      panic("region_alloc: not enough memory for page_alloc");
    page_insert(e->env_pgdir, page, (void *)addr, PTE_U | PTE_W |
                                                        PTE_P);
    addr += PGSIZE;
  }
}

static void load_icode(struct Env *e, uint8_t *binary, size_t size)
{
  struct Elf *elfhdr = (struct Elf *)binary;
  if (elfhdr->e_magic != ELF_MAGIC)
    panic("load_icode: bad elf");
  struct Proghdr *ph = (struct Proghdr *) (binary + elfhdr->e_phoff);
```

```
  struct Proghdr *eph = ph + elfhdr->e_phnum;
  lcr3(PADDR(e->env_pgdir));
  for (; ph < eph; ph++) {
    if (ph->p_type == ELF_PROG_LOAD) {
      if (ph->p_filesz > ph->p_memsz)
        panic("load_icode: size in file should <= size in memory");
      region_alloc(e, (void *)ph->p_va, ph->p_memsz);
      memmove((void *)ph->p_va, binary + ph->p_offset, ph->p_filesz);
      memset((void *)(ph->p_va + ph->p_filesz), 0,
                                    ph->p_memsz - ph->p_filesz);
    }
  }
  e->env_tf.tf_eip = elfhdr->e_entry;
  region_alloc(e, (void *)(USTACKTOP - PGSIZE), PGSIZE);
}

void env_create(uint8_t *binary, enum EnvType type)
{
  struct Env *e;
  int code = env_alloc(&e, 0);
  if (code < 0)
    panic("Error: Can't create a new environment\n");
  load_icode(e, binary);
  e->env_link = NULL;
  e->env_type = type;
}

void env_run(struct Env *e)
{
  if (curenv != e) {
    if (curenv && curenv->env_status == ENV_RUNNING) {
```

```
      curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
  }
  env_pop_tf(&e->env_tf);
}
```

A este punto ya se puede ejecutar QEMU para verificar las funciones ya ingresadas en el proyecto. El laboratorio incluye un programa llamado *hello* que se puede ejecutar como prueba. Lo que nos presenta el problema de que no existen llamadas a sistemas que controlen la ejecución de este binario, lo que puede que se ejecute con muchos errores. Es importante a este punto verificar que todo está corriendo donde corresponde, es decir, que el almacenamiento de memoria realizado en el laboratorio 2 esté funcionando bien, sino, entonces este es el punto para volver a corregir el trabajo.

Para el manejo de interrupciones y excepciones, se solicita realizar la lectura del capítulo 9 del manual del programador [21] o el capítulo 5 del manual de arquitectura de software para el desarrollador Intel[14].

La experiencia cuenta con cierta introducción al tema de la utilización de una de las formas en la cual el sistema operativo enfrenta las interrupciones, tales como la tabla de descripción de interrupciones y el segmento de procesos pendientes. Se presenta un ejemplo gráfico el cual se debe implementar de forma posterior modificando los archivos *kern/trapentry.S* y *kern/trap.c* de tal forma de definirlas.

Para el archivo *kern/trapentry.S* se debe agregar al final del archivo las siguientes líneas:

```
TRAPHANDLER_NOEC(div_zero, T_DIVIDE) /* 0 */
TRAPHANDLER_NOEC(debug_exception, T_DEBUG) /* 1 */
TRAPHANDLER(non_mask_inter, T_NMI) /* 2 */
TRAPHANDLER_NOEC(break_point, T_BRKPT) /* 3 */
```

```
TRAPHANDLER_NOEC(overflow, T_OFLOW) /* 4 */
TRAPHANDLER_NOEC(bounds_check, T_BOUND) /* 5 */
TRAPHANDLER_NOEC(illegal_opcode, T_ILLOP) /* 6 */
TRAPHANDLER_NOEC(device_not_avail, T_DEVICE) /* 7 */
TRAPHANDLER(double_fault, T_DBLFLT) /* 8 */
TRAPHANDLER_NOEC(invalid_tss, T_TSS) /* 10 */
TRAPHANDLER(seg_not_pres, T_SEGNP) /* 11 */
TRAPHANDLER(stack_exception, T_STACK) /* 12 */
TRAPHANDLER(gen_protec_fault, T_GPFLT) /* 13 */
TRAPHANDLER(page_fault, T_PGFLT) /* 14 */
TRAPHANDLER_NOEC(float_point_err, T_FPERR) /* 16 */
TRAPHANDLER_NOEC(align_check, T_ALIGN) /* 17 */
TRAPHANDLER_NOEC(machine_check, T_MCHK) /* 18 */
TRAPHANDLER_NOEC(simd_float_error, T_SIMDERR) /* 19 */
TRAPHANDLER_NOEC(sys_call, T_SYSCALL) /* 48 */

 _alltraps:
  pushl %ds
  pushl %es
  pushal

  movw $GD_KD, %ax
  movw %ax, %ds
  movw %ax, %es

  pushl %esp
  call trap
```

Y para el archivo *kern/trap.c* se debe modificar la función *trap_init()* para inicializar.

```
void trap_init(void)
```

```
{
  extern struct Segdesc gdt[];
  SETGATE(idt[T_DIVIDE], 0, GD_KT, div_zero, 0);
  SETGATE(idt[T_DEBUG], 0, GD_KT, debug_exception, 0);
  SETGATE(idt[T_NMI], 0, GD_KT, non_mask_inter, 0);
  SETGATE(idt[T_BRKPT], 0, GD_KT, break_point, 3);
  SETGATE(idt[T_OFLOW], 0, GD_KT, overflow, 0);
  SETGATE(idt[T_BOUND], 0, GD_KT, bounds_check, 0);
  SETGATE(idt[T_ILLOP], 0, GD_KT, illegal_opcode, 0);
  SETGATE(idt[T_DEVICE], 0, GD_KT, device_not_avail, 0);
  SETGATE(idt[T_DBLFLT], 0, GD_KT, double_fault, 0);
  SETGATE(idt[T_TSS], 0, GD_KT, invalid_tss, 0);
  SETGATE(idt[T_SEGNP], 0, GD_KT, seg_not_pres, 0);
  SETGATE(idt[T_STACK], 0, GD_KT, stack_exception, 0);
  SETGATE(idt[T_GPFLT], 0, GD_KT, gen_protec_fault, 0);
  SETGATE(idt[T_PGFLT], 0, GD_KT, page_fault, 0);
  SETGATE(idt[T_FPERR], 0, GD_KT, float_point_err, 0);
  SETGATE(idt[T_ALIGN], 0, GD_KT, align_check, 0);
  SETGATE(idt[T_MCHK], 0, GD_KT, machine_check, 0);
  SETGATE(idt[T_SIMDERR], 0, GD_KT, simd_float_error, 0);
  trap_init_percpu();
}
```

B.3.2. Part B: Page Faults, Breakpoints Exceptions, and System Calls

Se solicita la modificación de la función *trap_dispatch()* para despachar la página de excepción a la función *page_fault_handler()*. El objetivo es tener una excepción en caso de que la lista falle, ésta se encuentra en la posición 14 de la lista de vectores (*T_PGFLT*). Además se pide enlazar, en la misma función, las interrupciones de tipo llamadas a sistema (*T_SYSCALL*) y modificar la función *trap_init()* y con ello modificar además la función *syscall()* en el archivo *kern/syscall.c*. El código a modificar se encuentra en el archivo

kern/trap.c y quedaría como:

```
static void trap_dispatch(struct Trapframe *tf)
{
  switch (tf->tf_trapno) {
    case T_DEBUG:
      break;
    case T_BRKPT:
      monitor(tf);
      break;
    case T_OFLOW:
      break;
    case T_SYSCALL:
      tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
      tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx, tf->tf_regs.reg_ebx,
      tf->tf_regs.reg_edi, tf->tf_regs.reg_esi);
      break;
    case T_PGFLT:
      page_fault_handler(tf);
      break;
    default:
      if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
      else {
        env_destroy(curenv);
        return;
      }
  }
}

void page_fault_handler(struct Trapframe *tf)
```

```
{
  uint32_t fault_va;
  fault_va = rcr2();
  if ((tf->tf_cs & 3) != 3)
    panic("Page fault happened in kernel mode\n");
  cprintf("[%08x] user fault va %08x ip %08x\n",
    curenv->env_id, fault_va, tf->tf_eip);
  print_trapframe(tf);
  env_destroy(curenv);
}

void trap_init(void)
{
  extern struct Segdesc gdt[];
  SETGATE(idt[T_DIVIDE], 0, GD_KT, div_zero, 0);
  SETGATE(idt[T_DEBUG], 0, GD_KT, debug_exception, 0);
  SETGATE(idt[T_NMI], 0, GD_KT, non_mask_inter, 0);
  SETGATE(idt[T_BRKPT], 0, GD_KT, break_point, 3);
  SETGATE(idt[T_OFLOW], 0, GD_KT, overflow, 0);
  SETGATE(idt[T_BOUND], 0, GD_KT, bounds_check, 0);
  SETGATE(idt[T_ILLOP], 0, GD_KT, illegal_opcode, 0);
  SETGATE(idt[T_DEVICE], 0, GD_KT, device_not_avail, 0);
  SETGATE(idt[T_DBLFLT], 0, GD_KT, double_fault, 0);
  SETGATE(idt[T_TSS], 0, GD_KT, invalid_tss, 0);
  SETGATE(idt[T_SEGNP], 0, GD_KT, seg_not_pres, 0);
  SETGATE(idt[T_STACK], 0, GD_KT, stack_exception, 0);
  SETGATE(idt[T_GPFLT], 0, GD_KT, gen_protec_fault, 0);
  SETGATE(idt[T_PGFLT], 0, GD_KT, page_fault, 0);
  SETGATE(idt[T_FPERR], 0, GD_KT, float_point_err, 0);
  SETGATE(idt[T_ALIGN], 0, GD_KT, align_check, 0);
  SETGATE(idt[T_MCHK], 0, GD_KT, machine_check, 0);
```

```
  SETGATE(idt[T_SIMDERR], 0, GD_KT, simd_float_error, 0);
  trap_init_percpu();
}

int32_t syscall(uint32_t syscallno, uint32_t a1, uint32_t a2,
  uint32_t a3, uint32_t a4, uint32_t a5)
{
  switch (syscallno) {
    case SYS_cputs:
      sys_cputs((char *)a1, a2);
      return 0;
    case SYS_cgetc:
      return (int32_t)sys_cgetc();
    case SYS_getenvid:
      return (int32_t)sys_getenvid();
    case SYS_env_destroy:
      return sys_env_destroy(a1);
    default:
      return -E_INVAL;
  }
}
```

A esta altura ya nos encontramos en una buena posición para poder iniciar a nuestra conveniencia el modo usuario. Dicha instanciación se realiza en la parte superior del archivo *lib/entry.S* la que enlaza a la función *libmain()* que se encuentra en el archivo *lib/libmain.c*. Es necesario modificar esta función para que inicie el puntero global *thisenv* para que enlace la estructira *Env* en la lista *envs[]*.

```
void libmain(int argc, char **argv)
{
  thisenv = 0;
  if (argc > 0)
```

```
    binaryname = argv[0];
  envid_t thisEnvId = sys_getenvid();
  thisenv = &envs[ENVX(thisEnvId)];
  umain(argc, argv);
  exit();
}
```

Luego de ingresar este código, el sistema está configurado para que inicie la aplicación *user/hello* la que nos saludará con un ”Hello World” y después un ”i am environment 00001000”.

Para los fallos de páginas y la protección de la memoria, es importante tener una forma en la cual los programas no utilicen la memoria de otros ya que puede ocasionar problemas de procedimientos en ellos. En caso de que no haya memoria disponible, el sistema operativo debe tratar de solucionar de alguna forma o lanzar una interrupción al respecto.

Para el modo usuario es necesario realizar este tipo de revisión en la memoria para que el ambiente de trabajo funcione con la memoria correspondiente y no menos. Para ello es necesario editar el archivo *kern/pmap.c* y completar la función *user_mem_check()*.

```
int user_mem_check(struct Env *env, const void *va, size_t len,
  int perm)
{
  int permBits = perm | PTE_P;
  user_mem_check_addr = (uintptr_t)va;
  for (uintptr_t i = ROUNDDOWN((uint32_t)va, PGSIZE); i <
    ROUNDUP((uint32_t)va + len, PGSIZE); i += PGSIZE) {
    if (i >= ULIM) {
      if (user_mem_check_addr < i)
        user_mem_check_addr = i;
      return -E_FAULT;
    }
```

```
    pte_t *entry = pgdir_walk(env->env_pgdir, (void *)i, 0);
    if (!entry) {
      if (user_mem_check_addr < i)
        user_mem_check_addr = i;
      return -E_FAULT;
    }
    if ((*entry & permBits) != permBits) {
      if (user_mem_check_addr < i)
        user_mem_check_addr = i;
      return -E_FAULT;
    }
  }
  return 0;
}
```

Luego es necesario modificar el arhicvo *kern/syscall.c* para revisar si los argumentos de las llamadas a sistemas son correctas. Finalmente editar el archivo *kern/kdebug.c* para que en el *usd, stabs* y *stabstr* llame a la función *user_mem_check()*.

Finalmente basta con revisar que todo esté corriendo como corresponde ejecutando *./grade-lab3*. Si es que hay alguna falta, es necesario revisar todo el procedimiento nuevamente.

B.4. Lab 4: Preemptive Multitasking

Al finalizar este laboratorio el sistema operativo contará con un sistema simple de multitasking, lo que abre la posibilidad de contar con simultáneos ambientes de modo usuario. en la parte A se trabajará sobre el sistema multiprocesos, implementando la itineración de tipo round-robin. En la parte B se trabajará en la implementación de la función *fork()* para realizar copias de procesos. En la parte C se trabajará en la comunicación entre procesos (IPC) con tal de que los procesos puedan trabajar en conjunto y sincronizados entre los demás.

Para comenzar es necesario copiar los nuevos archivos que se incluyen para esta elaboración.

```
git checkout -b lab4 origin/lab4
git merge lab3
```

El laboratorio está separado por tres partes de las cuales es recomendable realizar una por semana.

B.4.1. Part A: Multiprocessor Support and Cooperative Multitasking

Como se había comentado anteriormente, en esta parte de la experiencia se trabaja en correr un sistema que soporte multiprocesos y la implementación de nuevas llamadas al sistema que permita la creación de nuevos procesos. También se implementa la itineración de tipo round-robin, permitiendo al kernel cambiar de un proceso a otro cuando el proceso actual abandona voluntariamente la CPU.

Para el soporte de multiprocesos se implementará el sistema SMP (symmetric multiprocessing) que consiste en una partición equivalente de los recursos del hardware. El código en este punto es separado en dos partes, los procesos de inicio del sistema (BSP) y los procesos de aplicaciones (APs), en donde BSP activa las APs luego de que el sistema operativo arranque.

Lo primero que hay que realizar es la modificación del archivo *kern/pmap.c* en la función *mmio_map_region*; se debe realizar con lecturas previas al tema y buscando referencias en la función *lapic_init* en el archivo *kern/lapic.c*. La función queda:

```
void * mmio_map_region(physaddr_t pa, size_t size)
{
  static uintptr_t base = MMIOBASE;
  uintptr_t retBase = base;
  uintptr_t endAddr = ROUNDUP(base + size, PGSIZE);
  if (endAddr >= MMIOLIM)
    panic("Too much size\n");
```

```
  uint32_t mapped_size = ROUNDUP(size, PGSIZE);
  boot_map_region(kern_pgdir, base, mapped_size, pa, PTE_PCD |
    PTE_PWT | PTE_W);
  base += mapped_size;
  return (void *)retBase;
}
```

Antes de que comienze las Aps, el BSP debe recolectar información del sistema, tal como el número total de CPU, sus IDs del APIC y las direcciones MMIO de la unidad LAPIC. La función *mp_init()* ubicada en el archivo *kern/mpconfig.c* da esta información leyendo la tabla que se ubica en la región de la memoria reservada para la BIOS.

Luego se explica el uso de la función *boot_aps()* la cual, en resumen, recopila la información antes nombrada e inicia las rutinas necesarias para que el sistema quede como multiprocesos. Con lo que como ejercicio se propone leer el comportamiento de la función *boot_aps()* en detalle en conjunto de *mp_main()* en el archivo *kern/init.c* y el código assembly *kern/mpentry.S* para luego modificar la función *page_init()* en *kern/pmap.c* para evadir agregar la página *MPENTRY_PADDR* a la lista libre. Entonces nuestra función quedaría cambiada como:

```
void page_init(void)
{
  size_t i;
  page_free_list = NULL;
  for (i = 0; i < npages; i++) {
    if (i == 0 || !inMapAvail(&pages[i]) || isInIOHole(&pages[i]) ||
      page2pa(&pages[i]) == MPENTRY_PADDR) {
      pages[i].pp_ref = 1;
      pages[i].pp_link = NULL;
    } else {
      pages[i].pp_ref = 0;
      pages[i].pp_link = page_free_list;
```

```
      page_free_list = &pages[i];
    }
  }
}
```

Es importante distinguir entre los procesos que son per-CPU (privados entre ellos) y los procesos globales que todo el sistema comparte. Se solicita conocer el archivo *kern/cpu.h* en donde se definen varios de los procesos per.CPU, incluyendo la estructura *CpuInfo*. Es necesario mapear en la pila per-CPU empezando en *KSTACKTOP*, para ello hay que modificar la función *mem_init_mp()* ubicada en *kern/pmap.c*.

```
static void mem_init_mp(void)
{
  for (int i = 0; i < NCPU; i++) {
    uintptr_t kstacktop_i = KSTACKTOP - (i * (KSTKSIZE +
      KSTKGAP));
    boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE,
      KSTKSIZE, PADDR(percpu_kstacks[i]), PTE_W | PTE_P);
  }
}
```

También es necesario modificar la función *trap_init_percpu()* ubicada en *kern/trap.c* ya que se encuentra actualmente configurada para un solo proceso.

```
void trap_init_percpu(void)
{
  int num = cpunum();
  thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - num * (KSTKSIZE + KSTKGAP);
  thiscpu->cpu_ts.ts_ss0 = GD_KD;
  uint16_t index = (GD_TSS0 >> 3) + num;
  gdt[index] = SEG16(STS_T32A, (uint32_t) (&thiscpu->cpu_ts),
    sizeof(struct Taskstate) - 1, 0);
```

```
  gdt[index].sd_s = 0;
  ltr(index << 3);
  lidt(&idt_pd);
}
```

El sistema Operativo debe contar con un sistema que proteja de conflictos todo lo que tenga que ver con procesos que corran a nivel del kernel, por lo que es importante contar con algo que bloquee el acceso a los procesos que estén en modo usuario para que sólo uno se ejecute a la vez si es que necesita entrar a los niveles más bajos. Las definiciones para este sistema se encuentran declaradas en *kern/spinlock.h* con la variable *kernel_lock* y las funciones *lock_kernel()* y *unlock_kernel()*.

Es necesario implementar estas dos últimas funciones en *i386_init(), mp_main(), trap()* y *env_run()* según corresponda.

```
void i386_init(uint32_t magic, uint32_t addr)
{
  extern char edata[], end[];
  memset(edata, 0, end - edata);
  cons_init();
  assert(magic == MULTIBOOT_BOOTLOADER_MAGIC);
  cprintf("451 decimal is %o octal!\n", 451);
  cpuid_print();
  e820_init(addr);
  mem_init();
  env_init();
  trap_init();
  acpi_init();
  mp_init();
  lapic_init();
  pic_init();
  ioapic_init();
```

```
  ioapic_enable(IRQ_KBD, bootcpu->cpu_apicid);
  ioapic_enable(IRQ_SERIAL, bootcpu->cpu_apicid);
  lock_kernel();
  boot_aps();
#if defined(TEST)
  ENV_CREATE(TEST, ENV_TYPE_USER);
#else
  ENV_CREATE(user_yield, ENV_TYPE_USER);
#endif
  sched_yield();
}


void mp_main(void)
{
  lcr3(PADDR(kern_pgdir));
  cprintf("  AP #%d [apicid %02x] starting\n", cpunum(),
    thiscpu->cpu_apicid);
  lapic_init();
  env_init_percpu();
  trap_init_percpu();
  xchg(&thiscpu->cpu_status, CPU_STARTED);
  lock_kernel();
  sched_yield();
}

void trap(struct Trapframe *tf)
{
  asm volatile("cld" ::: "cc");
  extern char *panicstr;
  if (panicstr)
```

```
    asm volatile("hlt");
  if (xchg(&thiscpu->cpu_status, CPU_STARTED) == CPU_HALTED)
    lock_kernel();
  assert(!(read_eflags() & FL_IF));
  if ((tf->tf_cs & 3) == 3) {
    lock_kernel();
    assert(curenv);
    if (curenv->env_status == ENV_DYING) {
      env_free(curenv);
      curenv = NULL;
      sched_yield();
    }
    curenv->env_tf = *tf;
    tf = &curenv->env_tf;
  }
  last_tf = tf;
  trap_dispatch(tf);
  if (curenv && curenv->env_status == ENV_RUNNING)
    env_run(curenv);
  else
    sched_yield();
}

void env_run(struct Env *e)
{
  if (curenv != e) {
    if (curenv && curenv->env_status == ENV_RUNNING) {
      curenv->env_status = ENV_RUNNABLE;
    }
    curenv = e;
    e->env_status = ENV_RUNNING;
```

```
    e->env_runs++;
    lcr3(PADDR(e->env_pgdir));
  }
  unlock_kernel();
  env_pop_tf(&e->env_tf);
}
```

Ahora toca implementar el método round-robin para la itineración de procesos en el kernel. Éste funciona en base a la función *sched_yield()* en *kern/sched.c* la cual se encarga de decidir cual es el siguiente proceso a considerar correr. Es importante modificar dicha función para la implementación de este sistema y además cambiar *syscall()* para que utilice *sys_yield()* la cual es la función que se utiliza a nivel usuario para llamar a *sched_yield()* con seguridad.

```
void sched_yield(void)
{
  struct Env *idle;
  idle = thiscpu->cpu_env;
  int startEnv = !idle ? 0 : ENVX(idle->env_id);
  for (int i =  startEnv; i < NENV + startEnv; i++) {
    if (envs[i % NENV].env_status == ENV_RUNNABLE) {
      env_run(&envs[i % NENV]);
    }
  }
  if (idle && idle->env_status == ENV_RUNNING) {
    env_run(idle);
  }
  sched_halt();
}
```

Es necesario actualizar la función *mp_main* de *kern/init.c* para que sea invocado *sched_yield()*. Desde este punto ya es posible ejecutar QEMU con la variable *CPUS=1*.

Actualmente el sistema se encuentra limitado a correr procesos que el kernel inicialmente lista para ejecutar, por lo que ahora es necesario implementar llamadas a sistemas necesarias para que el usuario pueda crear y comenzar un proceso.

Tal como lo hace Unix, existe una forma de clonar procesos con una función llamada *fork()* con la única diferencia que éste retorna el PID del proceso hijo cuando es el proceso padre y 0 cuando es el proceso hijo. Con esto cada proceso tiene su propio espacio de memoria para su trabajo. La idea es implementar algo similar implementando las funciones *sys_exofork, sys_env_set_status, sys_page_alloc, sys_page_map* y *sys_page_unmap* en el archivo *kern/syscall.c* con ayuda de otras funciones nombradas en el enunciado. El resultado es el siguiente:

```
static envid_t sys_exofork(void)
{
  struct Env *e;
  int code = env_alloc(&e, thiscpu->cpu_env->env_id);
  if (code < 0)
    return code;
  e->env_status = ENV_NOT_RUNNABLE;
  e->env_tf = thiscpu->cpu_env->env_tf;
  e->env_tf.tf_regs.reg_eax = 0;
  return e->env_id;
}

static int sys_env_set_status(envid_t envid, int status)
{
  if (status != ENV_RUNNABLE && status != ENV_RUNNING)
    return -E_INVAL;
  struct Env *e;
  int code = envid2env(envid, &e, 1);
  if (!e)
    return -E_BAD_ENV;
```

```
  e->env_status = status;
  return 0;
}

static int sys_page_alloc(envid_t envid, void *va, int perm)
{
  uintptr_t virtAddr = (uintptr_t) va;
  if (virtAddr >= UTOP || (virtAddr % PGSIZE) ||
      (perm & ~PTE_SYSCALL) || (perm & (PTE_U | PTE_P))
      != (PTE_U | PTE_P))
    return -E_INVAL;
  struct Env *e;
  int code = envid2env(envid, &e, 1);
  if (!e)
    return -E_BAD_ENV;
  struct PageInfo *pp = page_alloc(ALLOC_ZERO);
  if (!pp)
    return -E_NO_MEM;
  code = page_insert(e->env_pgdir, pp, va, perm);
  if (code < 0) {
    page_decref(pp);
    return -E_NO_MEM;
  }
  return code;
}

static int sys_page_unmap(envid_t envid, void *va)
{
  uintptr_t virtAddr = (uintptr_t)va;
  if (virtAddr >= UTOP || (virtAddr % PGSIZE))
    return -E_INVAL;
```

```
  struct Env *e;
  int code = envid2env(envid, &e, 1);
  if (!e)
    return -E_BAD_ENV;
  page_remove(e->env_pgdir, va);
  return 0;
}

static int sys_page_map(envid_t srcenvid, void *srcva,
       envid_t dstenvid, void *dstva, int perm)
{
  uintptr_t srcAddr = (uintptr_t) srcva, dstAddr = (uintptr_t) dstva;
  if (srcAddr >= UTOP || (srcAddr % PGSIZE) || dstAddr >= UTOP
    || (dstAddr % PGSIZE) || (perm & ~PTE_SYSCALL) ||
        (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P))
    return -E_INVAL;
  struct Env *e_src, *e_dst;
  int code = envid2env(srcenvid, &e_src, 1);
  if (!e_src)
    return -E_BAD_ENV;
  int dst_code = envid2env(dstenvid, &e_dst, 1);
  if (!e_dst)
    return -E_BAD_ENV;
  pte_t *entry_addr;
  struct PageInfo *pp = page_lookup(e_src->env_pgdir, srcva,
    &entry_addr);
  if (!pp)
    return -E_INVAL;
  if ((perm & PTE_W) && !(*entry_addr & PTE_W))
    return -E_INVAL;
  code = page_insert(e_dst->env_pgdir, pp, dstva, perm);
```

```
  return code;
}
```

B.4.2. Part B: Copy-on-Write Fork

Fork() es implementado en este sistema de tal forma que copia toda la información del las páginas de memoria del padre y las pone en el lugar en el cual se reservó para el hijo, y luego se llama a la función *exec()* para que éste se independice del padre lo que se puede considerar un gasto de recurso ya que se pueden evaluar métodos tales como el Copy-on-Write el que copia la memoria sólo cuando el hijo necesite una modificación en éste, para lo demás utilizan un sistema de memoria compartida. En esta parte del laboratorio se busca que dicha optimización sea implementada.

Lo primero es implementar un sistema para que el usuario pueda registrar la entrada de un proceso con la llamada a sistema *sys_env_set_pgfault_upcall* que se encuentra en el archivo *kern/syscall.c*.

```
static int sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
  struct Env *e;
  int code = envid2env(envid, &e, 1);
  if (!e)
    return -E_BAD_ENV;
  e->env_pgfault_upcall = func;
  return 0;
}
```

Ahora hay que implementar un sistema de pila especial para los procesos ejecutados en el modo usuario, pila llamada user exception stack la cual mejora bastante el manejo de memoria y ayuda a reparar casos como la reestructuración de la memoria. Para esto es necesario modificar la función *page_fault_ahndler()* del archivo *kern/trap.c* para que soporte dicho requerimiento. El resultado es:

```
void page_fault_handler(struct Trapframe *tf)
{
  uint32_t fault_va;
  fault_va = rcr2();
  if ((tf->tf_cs & 3) == 0)
    panic("Page fault happened in kernel mode\n");
  if (curenv->env_pgfault_upcall) {
    struct UTrapframe *ttf;
    uintptr_t esp, ebp;
    if ((tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP -
      PGSIZE)) {
      esp = (uintptr_t)(tf->tf_esp - 4 - sizeof(struct UTrapframe));
      ebp = (uint32_t)(tf->tf_esp - 4);
    } else {
      esp = (uintptr_t)(UXSTACKTOP - sizeof(struct UTrapframe));
      ebp = (uintptr_t)(UXSTACKTOP - 1);
    }
    user_mem_assert(curenv, (void *)esp, UXSTACKTOP - esp,
      PTE_W | PTE_U);
    ttf = (struct UTrapframe *)esp;
    ttf->utf_fault_va = fault_va;
    ttf->utf_err = tf->tf_err;
    ttf->utf_regs = tf->tf_regs;
    ttf->utf_eflags = tf->tf_eflags;
    ttf->utf_esp = tf->tf_esp;
    ttf->utf_eip = tf->tf_eip;
    tf->tf_esp = esp;
    tf->tf_regs.reg_ebp = ebp;
    tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
    env_run(curenv);
```

```
  }
  cprintf("[%08x] user fault va %08x ip %08x\n",
    curenv->env_id, fault_va, tf->tf_eip);
  print_trapframe(tf);
  env_destroy(curenv);
}
```

Ahora es necesario implementar la rutina en assembly que permita la ejecución del manejo de páginas caídas en C y resuma la ejecución en la última instrucción buena, por lo que se debe modificar *_pgfault_upcall* en el archivo *lib/pfentry.S*

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
  pushl %esp
  movl _pgfault_handler, %eax
  call *%eax
  addl $4, %esp
  subl $4, 0x30(%esp)
  addl $8, %esp
  movl 0x28(%esp), %ebp
  movl 0x20(%esp), %eax
  movl %eax, (%ebp)
  popal
  addl $4, %esp
  popfl
  popl %esp
  ret
```

Luego, se necesita implementar la librería C para el mecanismo de manejo de páginas, por lo que se pide finalizar la función *set_pgfault_handler()* en el archivo *lib/pgfault.c*.

```
void set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
  int r;
  if (_pgfault_handler == 0) {
    if (sys_page_alloc(0, (void *)(UXSTACKTOP - PGSIZE), PTE_P |
      PTE_U | PTE_W) < 0)
      panic("gg cant alloc exeption stack\n");
    sys_env_set_pgfault_upcall(0, _pgfault_upcall);
  }
  _pgfault_handler = handler;
}
```

Luego de todo esto recién es posible implementar el sistema Copy-on-Write para la función *fork()*. Se pide modificar la función *fork()* del archivo *lib/fork.c* en conjunto con las funciones *duppage()* y *pgfault()*.

```
envid_t fork(void)
{
  envid_t envid;
  uint8_t *addr;
  int r;
  set_pgfault_handler(&pgfault);
  envid = sys_exofork();
  if (envid < 0)
    panic("sys_exofork: %e", envid);
  if (envid == 0) {
    thisenv = &envs[ENVX(sys_getenvid())];
    set_pgfault_handler(&pgfault);
    return 0;
  }
  for (addr = (uint8_t *)UTEXT; addr < (uint8_t *)USTACKTOP;
```

```
    addr += PGSIZE) {
    pde_t *dir = (pde_t *)&uvpd[PDX(addr)];
    pte_t *entry = (pte_t *)&uvpt[PGNUM(addr)];
    if ((*dir & PTE_P) && (*entry & PTE_P))
      r = duppage(envid, PGNUM(addr));
  }
  r = sys_page_alloc(envid, (void *)(addr + PGSIZE), PTE_W |
    PTE_P | PTE_U);
  if (r < 0)
    panic("exception stack %d\n", r);
  if (sys_env_set_pgfault_upcall(envid,
    thisenv->env_pgfault_upcall) < 0)
    panic("cant set upcall\n");
  if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
    panic("sys_env_set_status: %e", r);
  return envid;
}

static int duppage(envid_t envid, unsigned pn)
{
  int r;
  void *address = (void *)(pn * PGSIZE);
  pte_t *entry = (pte_t *)&uvpt[pn];
  bool copyWrite = ((*entry & PTE_W) == PTE_W || (*entry & PTE_COW)
    == PTE_COW);
  int perm = PTE_P | PTE_U;
  if (copyWrite) {
    if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), envid,
      ROUNDDOWN(address, PGSIZE), perm | PTE_COW) < 0)
      panic("cant update child mapping\n");
    if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), 0,
```

```
      ROUNDDOWN(address, PGSIZE), perm | PTE_COW) < 0)
      panic("can't update parent mapping\n");
  } else if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), envid,
    ROUNDDOWN(address, PGSIZE), perm) < 0) {
    panic("cant map parent to child\n");
  }
  return 0;
}


static void pgfault(struct UTrapframe *utf)
{
  void *addr = (void *) utf->utf_fault_va;
  uint32_t err = utf->utf_err;
  int r;
  pte_t *entry;
  if ((err & FEC_WR) != FEC_WR)
    panic("broked\n");
  entry = (pte_t *)&uvpt[PGNUM(addr)];
  if ((*entry & PTE_COW) != PTE_COW)
    panic("incorrect perm\n");
  if (sys_page_alloc(0, (void *)PFTEMP, PTE_W |
    PTE_P | PTE_U) < 0)
    panic("cant allocate a page at temp\n");
  memmove(PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
  if (sys_page_map(0, PFTEMP, 0, ROUNDDOWN(addr, PGSIZE), PTE_W |
    PTE_P | PTE_U) < 0)
    panic("gggg\n");
  if (sys_page_unmap(0, PFTEMP) < 0)
    panic("cant remove page at temp location\n");
}
```

Con esto finaliza la parte B de la experiencia.

B.4.3. Part C: Preemptive Multitasking and Inter-Process communication (IPC)

En la parte final de esta experiencia se desea implementar un sistema que comunique los procesos en ejecución mediante envío de mensajes.

Primero es necesario implementar un sistema de interrupción que tenga directa relación con el Clock, de tal forma que, por ejemplo, procesos que queden en loop infinito puedan ser finalizados después de un transcurso de tiempo. Para esto es necesario modificar la función *env_alloc* en el archivo *kern/env.c*

```
int env_alloc(struct Env **newenv_store, envid_t parent_id)
{
  int32_t generation;
  int r;
  struct Env *e;
  if (!(e = env_free_list))
    return -E_NO_FREE_ENV;
  if ((r = env_setup_vm(e)) < 0)
    return r;
  generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
  if (generation <= 0)  // Don't create a negative env_id.
    generation = 1 << ENVGENSHIFT;
  e->env_id = generation | (e - envs);
  e->env_parent_id = parent_id;
  e->env_type = ENV_TYPE_USER;
  e->env_status = ENV_RUNNABLE;
  e->env_runs = 0;
  memset(&e->env_tf, 0, sizeof(e->env_tf));
  e->env_tf.tf_ds = GD_UD | 3;
```

```
  e->env_tf.tf_es = GD_UD | 3;
  e->env_tf.tf_ss = GD_UD | 3;
  e->env_tf.tf_esp = USTACKTOP;
  e->env_tf.tf_cs = GD_UT | 3;
  e->env_tf.tf_eflags = FL_IF;
  e->env_pgfault_upcall = 0;
  e->env_ipc_recving = 0;
  env_free_list = e->env_link;
  *newenv_store = e;
  cprintf("[%08x] new env %08x\n", curenv ? curenv->env_id : 0,
    e->env_id);
  return 0;
}
```

Ahora, para manejar estas interrupciones es necesario modificar la función con la cual ya trabajamos antes llamada *trap_dispatch()* para que llame a *sched_yield()*para buscar y correr un ambiente diferente cuando una interrupción de tipo clock toma lugar.

```
static void trap_dispatch(struct Trapframe *tf)
{
  if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
    cprintf("Spurious interrupt on irq 7\n");
    print_trapframe(tf);
    return;
  }
  switch (tf->tf_trapno) {
    case T_BRKPT:
      monitor(tf);
      break;
    case T_SYSCALL:
      tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
```

```
        tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx,
        tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi,
        tf->tf_regs.reg_esi);
      break;
    case T_PGFLT:
      page_fault_handler(tf);
      break;
    case IRQ_OFFSET + IRQ_TIMER:
      lapic_eoi();
      time_tick();
      sched_yield();
      break;
    default:
      print_trapframe(tf);
      if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
      else {
        env_destroy(curenv);
        return;
      }
  }
}
```

Antes de proceder con lo importante de esta experiencia, es bueno saber si todo lo ya programado se encuentra ejecutando en orden. Por lo que es esencial que lo que se ha modificado esté pasando todos los test posibles y que no tenga inconsecuencias.

Se comienza a trabajar con la implementación de la comunicación entre procesos (IPC). Primero es necesario construir dos llamadas a sistemas y dos paquetes de librerías, éstas son *sys_ipc_recv()*, *sys_ipc_try_send()*, *ipc_recv* y *ipc_send* respectivamente de los archivos *kern/syscall.c* y *lib/ipc.c*.

```
static int sys_ipc_recv(void *dstva)
{
  uintptr_t dst_va = (uintptr_t)dstva;
  if (dst_va < UTOP && (dst_va % PGSIZE))
    return -E_INVAL;
  curenv->env_ipc_recving = 1;
  curenv->env_ipc_dstva = dst_va < UTOP ?
      dstva : (void *)UTOP;
  curenv->env_status = ENV_NOT_RUNNABLE;
  curenv->env_tf.tf_regs.reg_eax = 0;
  sys_yield();
  return 0;
}

static int sys_ipc_try_send(envid_t envid, uint32_t value,
  void *srcva, unsigned perm)
{
  uintptr_t va_src = (uintptr_t) srcva;
  struct Env *e;
  int code = envid2env(envid, &e, 0);
  if (!e)
    return -E_BAD_ENV;
  if (!e->env_ipc_recving)
    return -E_IPC_NOT_RECV;
  if (va_src < UTOP) {
    if ((va_src % PGSIZE) || (perm & ~PTE_SYSCALL) ||
          (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P))
      return -E_INVAL;
    pte_t *entry_addr;
    struct PageInfo *pp = page_lookup(
          thiscpu->cpu_env->env_pgdir, srcva, &entry_addr);
```

```
    if (!entry_addr)
      return -E_INVAL;
    if ((perm & PTE_W) && !(*entry_addr & PTE_W))
      return -E_INVAL;
    if ((uintptr_t)e->env_ipc_dstva < UTOP &&
          page_insert(e->env_pgdir, pp, e->env_ipc_dstva, perm))
      return -E_NO_MEM;
  }
  e->env_ipc_perm = va_src < UTOP ? perm : 0;
  e->env_ipc_recving = 0;
  e->env_ipc_from = curenv->env_id;
  e->env_ipc_value = value;
  e->env_status = ENV_RUNNABLE;
  return 0;
}

int32_t ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
  if (!pg)
    pg = (void *)(UTOP + 1);
  int val = sys_ipc_recv(pg);
  if (val < 0) {
    if (from_env_store)
      *from_env_store = 0;
    if (perm_store)
      *perm_store = 0;
    return val;
  }
  if (from_env_store)
    *from_env_store = thisenv->env_ipc_from;
  if (perm_store)
```

```
    *perm_store = thisenv->env_ipc_perm;
  return thisenv->env_ipc_value;
}

void ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
  if (!pg)
    pg = (void *)(UTOP + 1);
  int sys_val;
  while ((sys_val = sys_ipc_try_send(to_env, val, pg, perm))
      == -E_IPC_NOT_RECV)
    sys_yield();
  if (sys_val < 0)
    panic("ipc_send got a non-err recv error");
}
```

Es posible probar estas funciones ejecutando QEMU y utilizando los binarios *user/-pingpong* y *user/primes*. Con estas cuatro funciones desarrolladas ya nos encontramos con la finalización de la experiencia 4 por lo que se puede revisar que todo esté en orden ejecutando *./grade-lab4*.

B.5. Lab 5: File system, Spawn and Shell

En esta experiencia se trabaja en la librería *spawn* la que se encarga de cargar y correr ejecutables que se encuentran en disco, y se preparará una terminal que corra dentro de la consola. Para esto es necesario implementar un sistema de file system.

Al igual que los anteriores, se cargarán nuevos archivos predefinidos que sirven de guía para el desarrollo del trabajo.

```
git checkout -b lab5 origin/lab5
git merge lab4
```

Para proseguir es necesario comentar la línea *ENV_CREATE(fs_fs)* del archivo *kern/init.c* y la llamada a la función *close_all()* en el archivo *lib/exit.c* de forma temporal hasta que sea necesario ya que puede causar errores de ejecución al implementar las funciones de *fs/fs.c*.

B.5.1. The File System

La idea es implementar un file System con las funciones básicas, tales como crear, leer, escribir y borrar. nNo contendrá ningún sistema de permisos ni un manejo de archivos para más de un usuario a la vez, tampoco tendrá enlaces simbólicos, registro de tiempos, entre otras cosas.

Lo primero es dar los privilegios de tipo I/O en la función *env_create()* en el archivo *kern/env.c*. Ojo que sólo debe darse estos permisos a esta función y no a ninguna otra.

```
void env_create(uint8_t *binary, enum EnvType type)
{
  struct Env *e;
  int code = env_alloc(&e, 0);
  if (code < 0)
    panic("Error: Can't create a new environment\n");
  load_icode(e, binary);
  e->env_link = NULL;
  e->env_status = ENV_RUNNABLE;
  e->env_type = type;
  if (type == ENV_TYPE_FS) {
    pte_t *entry = pgdir_walk(e->env_pgdir, (void *)ahci_va, 0);
    *entry |= PTE_U | PTE_W | PTE_P;
  }
}
```

Lo siguiente es implementar un buffer cache para ayudar con la memoria virtual de los procesos. Se trabajará con un disco duro de tamaño 3GB o menos por lo que el mapeo

se tiene que armar desde 0x10000000 a 0xD0000000. Se pide implementar las funciones *bc_pgfault()* y *flush_block()* en *fs/bc.c*.

```
static void bc_pgfault(struct UTrapframe *utf)
{
  void *addr = (void *) utf->utf_fault_va;
  uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
  int r;
  if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
    panic("page fault in FS: eip %08x, va %08x, err %04x",
          utf->utf_eip, addr, utf->utf_err);
  if (super && blockno >= super->s_nblocks)
    panic("reading non-existent block %08x\n", blockno);
  addr = ROUNDDOWN(addr, PGSIZE);
  if ((r = sys_page_alloc(0, addr, PTE_U | PTE_W | PTE_P)) < 0)
    panic("gucked %e\n", r);
  if ((r = ahci_read((blockno * BLKSECTS), addr, (PGSIZE /
      SECTSIZE))) < 0)
    panic("cant read\n");
  if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
      PTE_SYSCALL)) < 0)
    panic("in bc_pgfault, sys_page_map: %e", r);
  if (bitmap && block_is_free(blockno))
    panic("reading free block %08x\n", blockno);
}

void flush_block(void *addr)
{
  uint32_t blockno = ((uint32_t)addr - DISKMAP) / BLKSIZE;
  if (addr < (void*)DISKMAP || addr >= (void*)(DISKMAP + DISKSIZE))
    panic("flush_block of bad va %08x", addr);
```

```
  addr = ROUNDDOWN(addr, PGSIZE);
  if (va_is_mapped(addr) && va_is_dirty(addr)) {
    int r;
    if ((r = ahci_write((blockno * BLKSECTS), addr, (PGSIZE /
          SECTSIZE))) < 0)
      panic("cant write\n");
    if ((r = sys_page_map(0, addr, 0, addr, uvpt[PGNUM(addr)] &
          PTE_SYSCALL)) < 0)
      panic("in bc_pgfault, sys_page_map: %e", r);
  }
}
```

Ahora hay que configurar el bloque de bitmap. Se solicita implementar la función *alloc_block()* usando *free_block* como modelo en el archivo *fs/fs.c*

```
int alloc_block(void)
{
  int blockno = 0;
  while (blockno != super->s_nblocks && !block_is_free(blockno))
    blockno++;
  if (blockno != super->s_nblocks) {
    bitmap[blockno / 32] &= ~(BIT(blockno % 32));
    flush_block(diskaddr(blockno));
    return blockno;
  }
  return -E_NO_DISK;
}
```

Hay que trabajar ahora en las funciones relacionadas con el manejo de los bloques, es decir, la modificación de los bloques que son apuntados por la lista de punteros que lo enlazan. Hay que realizar las funciones *file_block_walk()* y *file_get_block()*.

```
static int file_block_walk(struct File *f, uint32_t filebno,
  uint32_t **ppdiskbno, bool alloc)
{
  if (filebno < 0 || filebno >= (NDIRECT + NINDIRECT))
    return -E_INVAL;
  if (filebno < NDIRECT) {
    if (ppdiskbno) {
      *ppdiskbno = &f->f_direct[filebno];
    }
  } else {
    if (!f->f_indirect) {
      if (!alloc)
        return -E_NOT_FOUND;
      f->f_indirect = alloc_block();
      if (f->f_indirect < 0)
        return -E_NO_DISK;
      memset(diskaddr(f->f_indirect), 0, BLKSIZE);
    }
    uint32_t *ind = (uint32_t *)diskaddr(f->f_indirect);
    if (ppdiskbno)
      *ppdiskbno = &ind[filebno - NDIRECT];
  }
  return 0;
}

int file_get_block(struct File *f, uint32_t filebno, char **blk)
{
  if (filebno < 0 || filebno >= NDIRECT + NINDIRECT)
    return -E_INVAL;
  uint32_t *blockNo;
  int ret = file_block_walk(f, filebno, &blockNo, 1);
```

```
  if (ret < 0)
    return ret;
  if (!(*blockNo)) {
    *blockNo = alloc_block();
    if (*blockNo < 0)
      return -E_NO_DISK;
  }
  if (blk)
    *blk = (char *)diskaddr(*blockNo);
  return 0;
}
```

A esta altura ya se encuentra implementada la funcionalidad, ahora hay que hacerla accesible a otros procesos que quieran utilizar file system. El camino de lectura y escritura de archivos será una petición entre procesos con lo implementado en el laboratorio 4. Para ello es necesario editar la función *serve_read()* ubicada en *fs/serv.c*.

```
int serve_read(envid_t envid, union Fsipc *ipc)
{
  struct Fsreq_read *req = &ipc->read;
  struct Fsret_read *ret = &ipc->readRet;
  int r;
  struct OpenFile *o;
  if (debug)
    cprintf("serve_read %08x %08x %08x\n", envid,
      req->req_fileid, req->req_n);
  if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;
  r = file_read(o->o_file, ret->ret_buf, req->req_n,
    o->o_fd->fd_offset);
  if (r < 0)
```

```
    return r;
  int read = r;
  if ((r = seek(fd2num(o->o_fd), o->o_fd->fd_offset + r)) < 0) {
    return r;
  }
  return read;
}
```

Siguiendo el ritmo, y con la idea de la función implementada recién, implementar *serve_write()* en *fs/serv.c* y *devfile_write()* en *lib/file.c*.

```
int serve_write(envid_t envid, struct Fsreq_write *req)
{
  int r;
  struct OpenFile *o;
  if (debug)
    cprintf("serve_write %08x %08x %08x\n", envid,
          req->req_fileid, req->req_n);
  if ((r = openfile_lookup(envid, req->req_fileid, &o)) < 0)
    return r;
  r = file_write(o->o_file, req->req_buf, req->req_n,
      o->o_fd->fd_offset);
  if (r < 0)
    return r;
  int bWritten = r;
  if ((r = seek(fd2num(o->o_fd), o->o_fd->fd_offset + r)) < 0)
    return r;
  return bWritten;
}

static ssize_t devfile_write(struct Fd *fd, const void *buf,
  size_t n)
```

```
{
  int r;
  fsipcbuf.write.req_fileid = fd->fd_file.id;
  fsipcbuf.write.req_n = n;
  memmove(fsipcbuf.write.req_buf, buf, n);
  if ((r = fsipc(FSREQ_WRITE, NULL)) < 0)
    return r;
  assert(r <= n);
  return r;
}
```

B.5.2. Spawning Processes

La idea es crear un nuevo ambiente, cargar archivos del file system en él y comenzar a correr los procesos como hijos. *Spawn* funciona de forma similar a *fork()* y *exec()* pero con la diferencia que los ejecuta al iniciar.

Se pide implementar una nueva llamada al sistema *sys_env_set_trapframe()* para inicializar el estado de un nuevo ambiente, en el archivo *kern/syscall.c* y actualizar la función *syscall()*.

```
static int sys_env_set_trapframe(envid_t envid, struct Trapframe *tf)
{
  struct Env *e;
  int code = envid2env(envid, &e, 1);
  if (!e)
    return code;
  e->env_tf = *tf;
  return 0;
}

int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3,
```

```
  uint32_t a4, uint32_t a5)
{
  switch (syscallno) {
    case SYS_cputs:
      sys_cputs((char *)a1, a2);
      return 0;
    case SYS_cgetc:
      return (int32_t)sys_cgetc();
    case SYS_getenvid:
      return (int32_t)sys_getenvid();
    case SYS_env_destroy:
      return sys_env_destroy(a1);
    case SYS_yield:
      sched_yield();
      return 0;
    case SYS_page_alloc:
      return sys_page_alloc(a1, (void *)a2, a3);
    case SYS_page_map:
      return sys_page_map(a1, (void *)a2, a3, (void *)a4, a5);
    case SYS_page_unmap:
      return sys_page_unmap(a1, (void *)a2);
    case SYS_exofork:
      return sys_exofork();
    case SYS_env_set_status:
      return sys_env_set_status(a1, a2);
    case SYS_env_set_pgfault_upcall:
      return sys_env_set_pgfault_upcall(a1, (void *)a2);
    case SYS_time_msec:
      return sys_time_msec();
    case SYS_ipc_try_send:
```

```
      return sys_ipc_try_send(a1, a2, (void *)a3, a4);
    case SYS_ipc_recv:
      return sys_ipc_recv((void *)a1);
    case SYS_env_set_trapframe:
      return sys_env_set_trapframe(a1, (struct Trapframe *)a2);
    default:
      return -E_INVAL;
  }
}
```

Nos queda compartir los estados de las librerías de fork y spawn, tales como la información de los procesos y los archivos. Hay que cambiar fork de tal forma que ocupe las librerías operativas del sistema integradas en este proceso y así utilice una interacción entre las librerías; fork deja de trabajar de forma copy-on-write para algo más óptimo.

Se solicita cambiar *duppage* en el archivo *lib/fork.c* para que funcione de la forma recién comentada. Después implementar *copy_shared_pages* en *lib/spawn.c*.

```
static int duppage(envid_t envid, unsigned pn)
{
  void *address = (void *)(pn * PGSIZE);
  pte_t *entry = (pte_t *)&uvpt[pn];
  bool copyWrite = ((*entry & PTE_W) == PTE_W ||
      (*entry & PTE_COW) == PTE_COW);
  bool isShared = ((*entry & PTE_SYSCALL) & PTE_SHARE)
      == PTE_SHARE;
  int perm = PTE_P | PTE_U;
  if (copyWrite && !isShared) {
    if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), envid,
          ROUNDDOWN(address, PGSIZE), perm | PTE_COW) < 0)
      panic("cant update child mapping\n");
    if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), 0,
```

```
          ROUNDDOWN(address, PGSIZE), perm | PTE_COW) < 0)
      panic("can't update parent mapping\n");
  } else {
    if (sys_page_map(0, ROUNDDOWN(address, PGSIZE), envid,
          ROUNDDOWN(address, PGSIZE), isShared ?
          (*entry & PTE_SYSCALL) : perm) < 0)
      panic("cant map parent to child\n");
  }
  return 0;
}
```

```
static int copy_shared_pages(envid_t child)
{
  uint8_t *addr;
  int r;
  for (addr = (uint8_t *)UTEXT; addr < (uint8_t *)USTACKTOP;
      addr += PGSIZE) {
    pde_t *dir = (pde_t *)&uvpd[PDX(addr)];
    pte_t *entry = (pte_t *)&uvpt[PGNUM(addr)];
    if ((*dir & PTE_P) && ((*entry & PTE_P) &&
          (*entry & PTE_SHARE) == PTE_SHARE))
      if ((r = sys_page_map(0, addr, child, addr,
            (*entry & PTE_SYSCALL))) < 0)
        panic("can't copy page mapping in spawn\n");
  }
  return 0;
}
```

B.5.3. The keyboard interface

Es necesario reconocer el teclado si es que queremos tener nuestra propia consola. Es necesario modificar la función *trap_dispatch* para usar *kbd_intr*, localizada en el archivo *kern/trap.c* para manejar la interrupción *IRQ_OFFSET+IRQ_KBD* y *serial_intr* para manejar la interrupción *IRQ_OFFSET+IRQ_SERIAL*.

```
static void trap_dispatch(struct Trapframe *tf)
{
  if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
    cprintf("Spurious interrupt on irq 7\n");
    print_trapframe(tf);
    return;
  }
  switch (tf->tf_trapno) {
    case T_BRKPT:
      monitor(tf);
      break;
    case T_SYSCALL:
      tf->tf_regs.reg_eax = syscall(tf->tf_regs.reg_eax,
            tf->tf_regs.reg_edx, tf->tf_regs.reg_ecx,
            tf->tf_regs.reg_ebx, tf->tf_regs.reg_edi,
            tf->tf_regs.reg_esi);
      break;
    case T_PGFLT:
      page_fault_handler(tf);
      break;
    case IRQ_OFFSET + IRQ_TIMER:
      lapic_eoi();
      time_tick();
      sched_yield();
```

```
      break;
    case IRQ_OFFSET + IRQ_KBD:
      lapic_eoi();
      kbd_intr();
      break;
    case IRQ_OFFSET + IRQ_SERIAL:
      lapic_eoi();
      serial_intr();
      break;
    default:
      print_trapframe(tf);
      if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
      else {
        env_destroy(curenv);
        return;
      }
  }
}
```

B.5.4. The Shell

A esta altura sólo nos queda verificar que todo el trabajo realizado en ésta y las experiencias anteriores dio frutos. El código de una consola ya viene implementado, lo podemos ejecutar con el comando *make run-icode* y como prueba podemos ejecutar los comandos:

```
echo hello world | cat
cat lorem |cat
cat lorem |num
cat lorem |num |num |num |num |num
```

```
lsfd
```

Si todo funciona, entonces se finaliza el trabajo.