Informe - TP2

Grupo Nº3

Nombre	Legajo
Alejo Flores Lucey	62622
Andrés Carro Wetzel	61655
Nehuén Gabriel Llanos	62511

Algoritmo de Scheduling

El algoritmo de Scheduling implementado en este trabajo práctico está formado una estructura y dos listas.

La estructura presente en Scheduling representa un proceso y permite guardar información pertinente utilizada por el algoritmo de Scheduling. Esta se denomina ***Node*** y representa un nodo de las dos listas presentes. A continuación se pueden ver los elementos que forman parte de ella:

- ***next*****:** es un puntero al siguiente elemento de la lista.
- ***process*****:** es otra estructura que representa el ***Process Control Block*** de un proceso. Está formado por los siguientes elementos:
  - ***pid*****:** corresponde al pid identificador del proceso.
  - ***priority*****:** corresponde a la prioridad que posee el proceso. Se explicará más adelante las prioridades presentes en este algoritmo.
  - ***new priority*****:** representa la nueva prioridad cuando es modificada en Userland. Cuando se cambia la prioridad de un proceso se guarda este nuevo valor en ***new_priority*** dentro de ***PCB*** para que la siguiente vez que se le de tiempo de CPU, se actualice su valor.
  - ***status*****:** representa el status del proceso. Entre los que podemos encontrar: READY y BLOCKED.
  - ***quantums_left*****:** representa la cantidad de quantums faltantes hasta que se cambie de contexto.
  - ***rsp*****:** representa el lugar a dónde apuntaba el stack pointer cuando se cambió de contexto en este proceso.
  - ***stack_base*****:** representa el stack base dónde se guarda el contexto del proceso.

- ***blocked_queue*:** es una lista de todos los procesos que bloqueo el proceso. Ejemplo: cuando se realiza un `waitpid()`.
- ***file_descriptors[MAX_FDS]*:** representa el vector de los file descriptors.
- ***last_fd*:** representa la posición siguiente al último file descriptor guardado en el vector de file descriptors.
- ***argc*:** representa la cantidad de argumentos que recibió este proceso.
- ***argv*:** vector que guarda los argumentos que recibe este proceso.

Cada proceso cuenta con una prioridad (***priority***), que puede tener un valor entre 1 (prioridad más alta) y 9 (prioridad más baja). Cada prioridad se encuentra asociada unívocamente a una cantidad de quantums y un quantum es el tiempo hasta que se realiza un timer tick. Es importante aclarar que cuando un proceso es creado se le designa un prioridad default que tiene el valor de 4.

El algoritmo de Scheduling cuenta con dos listas. La primera de ellas se la denomina ***active*** y la segunda ***expired***. El primer elemento de la lista ***active*** es aquel que se encuentra corriendo y en esta misma lista tengo todos aquellos procesos que no se corrieron hasta el momento. Cuando el primer proceso de ***active*** terminó de correrse, en otras palabras, su quamtum es igual a cero, es agregado a ***expired*** teniendo en cuenta su prioridad. Cuando en ***active*** no tengo más procesos entonces se realiza un swap entre las dos listas. En caso de que se bloquee un proceso, solamente se cambiará el ***status*** a ***BLOCKED***. Como en ***active*** tengo los procesos a correrse, si en algún momento se encuentra con uno bloqueado, solamente lo saltea ubicándolo en la lista ***expired***. En caso de que todos los procesos que se encuentran en ***active*** están bloqueados, entonces directamente se realiza un swap de las dos listas para seguir buscando. Y en caso de que de todos los procesos presentes en ambas listas estén bloqueados se procede a darle tiempo de CPU a un proceso ***dummy*** hasta que alguno de los procesos sea desbloqueado. Es importante remarcar que cuando se crea un proceso este siempre es agregado a ***expired***, a menos que sea el primer proceso a agregar. A continuación se explicará como se inicializa el scheduler.

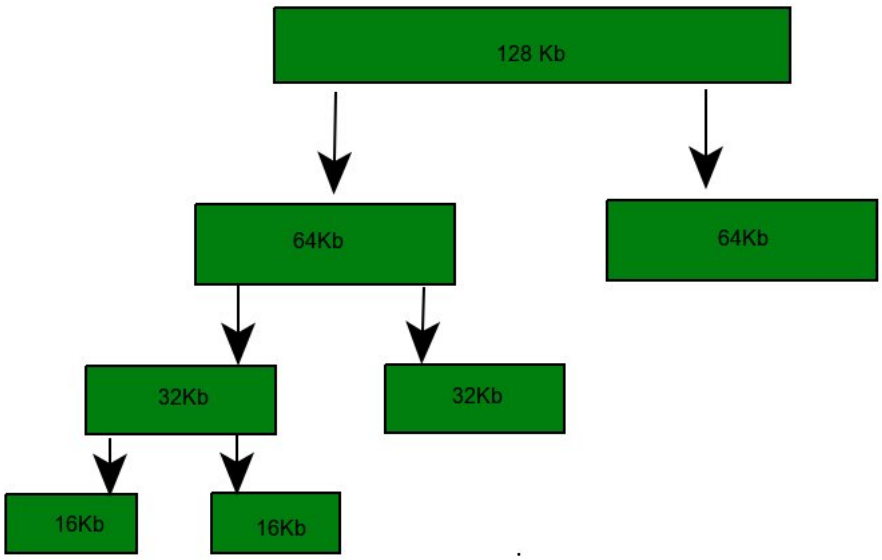
Cuando se da vida al Kernel, se deshabilitan las interrupciones y se crean dos procesos. El primero de ellos es el proceso ***dummy*** con el pid ***-1***. Este es agregado a la lista ***active*** y se setea su estado en ***BLOCKED***, dado que solamente se correrá cuando el resto de los procesos presentes en las listas se encuentran en estado ***BLOCKED***. Posteriormente, se crea el proceso ***Userland*** con el pid ***0***, se setea su estado en ***READY*** y se lo agrega a la lista ***expired***. Como se dijo previamente, pero se cree relevante remarcarlo, cada vez que se agrega un proceso, este es ubicado en la lista ***expired*** teniendo en cuenta su prioridad.

Cuando se habilitan las interrupciones, el Scheduler busca en ***active*** algún proceso con ***status=READY***, pero como el único que se encuentra presente es ***dummy*** que tiene ***status=BLOCKED*** entonces se realiza un swap entre las dos listas para ver si hay algún proceso con ***status=READY***. En este caso, encontramos a ***Userland*** y se realizan las configuraciones correspondientes para darle tiempo de CPU. Luego, ***Userland*** crea el proceso ***bash*** y el sistema operativo empieza a correrse.

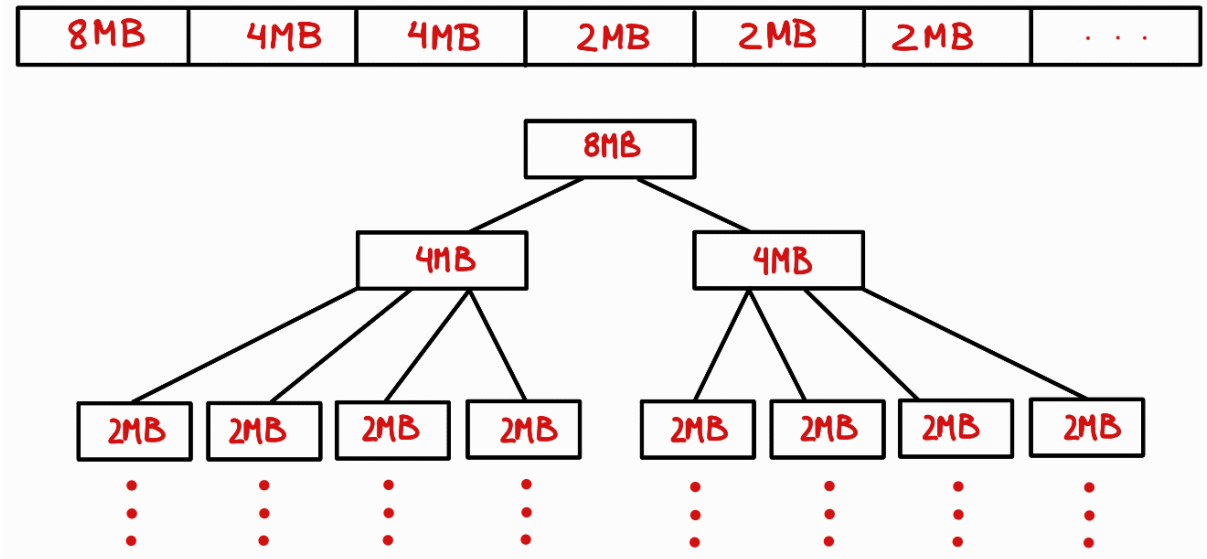
Memoria

Buddy Memory

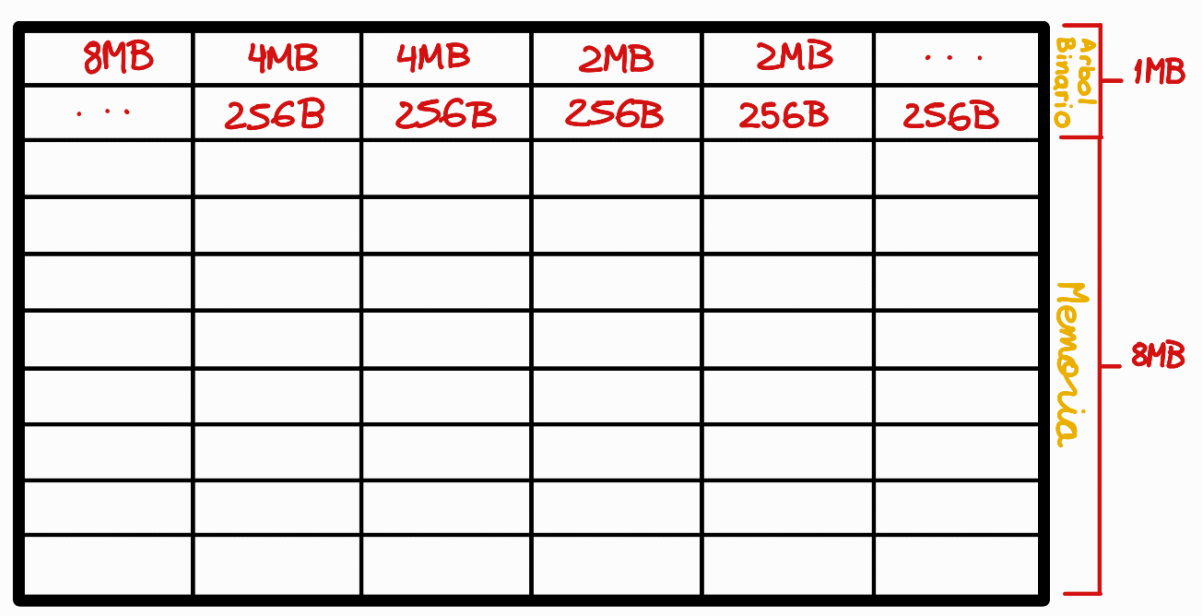
La idea de la memoria Buddy es buscar la memoria disponible más pequeña en la que el proceso que pidió ser alocado quepa. Para poder realizar esto se tiene un árbol binario, en donde cada nodo represeta un ***LEVEL*** distinto y este se encuentra asociado a $2^{LEVEL}B$ de memoria. Se puede ver en la siguiente imagen un Buddy con una memoria máxima de 128KB, es decir, tiene un máximo **LEVEL** de 17. Posteriormente vemos que se va subdividiendo en memorias de tamaño $2^{LEVEL-1}$ sucesivamente.



El Buddy implementado cuenta con 9MB de memoria. El primer MB está destinado a guardar un vector de una estructura denominadaba Buddy. Esta última representa un nodo del árbol binario y posee dos elementos, un ***status*** que me dice si el nodo o los subnodos fueron ocupados o no y un ***occupied_children*** que mantiene la cuenta de cuantos nodos hijos (incluyendose a si mismo) están ocupados. Entonces, el vector mencionado previamente guarda un árbol binario que tiene como raíz (posición 0) al ***level*** 23 y como últimos nodos a los que tienen de ***level*** 8. En total se tienen 65535 nodos. A continuación se muestra gráficamente como quedaría el vector:



Y la memoria en su totalidad quedaría formada de la siguiente manera:



Memory Allocation

Con el objetivo de alocar memoria, se recibe un solo parámetro que es el ***size*** y se retorna un ***void** ** que me indica el puntero a donde inicia el bloque de memoria.

En primer lugar, se chequea si el ***size*** tiene valores correctos, esto es, que el size sea distinto de cero, que sea menor que 8MB que la máxima memoria que puedo dar y también que existan bloques disponibles para la memoria solicitada. En caso de que ninguno de estos requisitos se cumpla entonces se retorna ***NULL***.

En segundo lugar, mediante la función `get_free_index_for_level()` obtenemos el índice del vector correspondiente al primer nodo de ese ***level*** que tiene su status ***FREE***, es decir, se encuentra disponible para ser utilizado. Luego de eso, se chequea que el índice obtenido sea menor que cero, porque si sucede esto, quiere decir que no hay lugar para alocar esa cantidad de memoria.

En tercer lugar, se realizan las actualizaciones de los nodos. Con `update_parents_status()` realizamos las actualizaciones de los padres para que actualicen la cantidad de nodos hijos ocupados. Después con `set_status()` actualizamos el estado del nodo elegido y el de todos los nodos hijos a ***TAKEN***.

En cuarto lugar, actualizamos al información de la memoria y finalmente retornamos la dirección de memoria obtenida mediante `obtain_address()` teniendo en cuenta su ***level*** y su índice.

Memory Deallocation

Con el fin de liberar memoria, se recibe como argumento un solo parámetro que es el puntero a la memoria que quiero liberar.

En primer lugrar, se verifica que ese puntero sea correcto, es decir, se encuentre dentro de los límites impuestos al realizar la inicialización de la memoria.

En segundo lugar, se obtiene el máximo ***level*** correspondiente a ese puntero mediante `get_max_posible_level()` y también el índice del vector, es decir, el nodo del árbol binario correspondiente a ese puntero por medio de `search_node()`. Esta información permite realizar las actualizaciones en ***status*** y ***occupied_children***. Antes de proseguir se debe chequear que el índice obtenido en `search_node()` haya sido menor que cero porque si es así quiere decir que no se encontró ningún nodo y por ende no se liberará nada.

En tercer lugar, se realizan las actualizaciones pertinentes en los padres y en los hijos del nodo obtenido. Con `update_parents_status()` se modifica la cantidad de nodos hijos ocupados de los padres y con `set_status()` se cambian los status del nodo obtenido y de sus hijos a ***FREE***.

En cuarto lugar, se actualizan las variables que guardan la información de la memoria para que se vea reflejada cuando se realice un llamado a `mem_info()`.

Heap Memory

La implementación de la Heap Memory consiste en formar una lista de bloques. Cada bloque posee un heading con la información relevante, representado por la estructura **MemoryNodeCDT**, y luego toda la data de ese bloque de memoria. El

MemoryNodeCDT posee los siguientes elementos:

- ***size*:** tamaño ocupado del bloque de memoria.
- ***leftover*:** es la cantidad de memoria libre con respecto al próximo bloque de memoria.
- ***next*:** es un puntero al siguiente bloque de memoria.
- ***previous*:** es un puntero al bloque de memoria anterior.

La información mencionada previamente permite disminuir la fragmentación de memoria cuando se realizan las operaciones de alocación y liberación de memoria.

Vemos un ejemplo en la siguiente imagen:

En esta memoria tenemos 3 bloques:

- Amarillo (0x100):
  - size: 2
  - leftover: 0
  - next: Verde (0x103)
  - previous: NULL
- Verde (0x103):
  - size: 3
  - leftover: 1
  - next: Violeta (0x108)
  - previous: Amarillo (0x100)
- Violeta (0x108):
  - size: 8
  - leftover: 23
  - next: NULL
  - previous: Verde (0x103)

Memory Allocation

Esta operación recibe mediante argumentos la cantidad de memoria que se quiere alocar y retorna un puntero a la dirección de memoria en donde se aloco el objeto en caso de éxito y si hay errores entonces retorna NULL.

En primer lugar, se chequea que exista memoria disponible, si el ***size*** que se pasa como argumento sea mayor a cero y si se está haciendo un malloc sin haber iniciado la memoria. En todos estos casos se retorna ***NULL*** porque son operaciones inválidas.

En segundo lugar, se recorre la lista de nodos en busca de alguno que tenga un leftover que sea mayor que la suma entre el tamaño que ocupa un ***MemoryNodeCDT*** y el ***size*** que se paso como argumento. En caso de que se llegue al final de la lista y no se encontró ningun nodo que cumpla con los requerimientos se retorna ***NULL***, caso contrario se prosigue con la última parte de esta operación.

Por último, se chequea si el ***size*** del ***MemoryNodeCDT*** encontrado es igual a cero. Si es así, entonces se utiliza este mismo nodo y se procede a setear las variables de ***MemoryNodeCDT***. En caso de que el size no sea cero entonces se debe crear un nuevo nodo, que iniciará en la dirección siguiente a la ocupada por el nodo encontrado. Se setean las variables correspondientes y también se ingresa el nodo en la lista cambiando los punteros previous y next. En ambos casos, se realizan los cambios correspondientes en la información de la memoria para que esta se muestre correctamente. Finalmente, se devuelve una dirección de memoria dependiendo si es que se agregó o no un nodo.

Memory Deallocation

Esta operación recibe mediante argumentos a un puntero y no retorna nada.

En primer lugar, se chequea el valor del puntero, si es ***NULL*** entonces se retorna porque es un valor inválido. Si es válido, entonces se obtiene la dirección donde se encontraría el nodo que contiene a la data.

En segundo y último lugar, se debe verificar si el nodo previo es ***NULL*** o no. Si es ***NULL*** se procede a modificar las variables pertinentes del nodo, dado que es el primero y no se lo puede borrar. Si no es ***NULL*** entonces se modifica el ***leftover*** del previo para que se le sume todo lo ocupado por el nodo a borrar y se cambia el nodo al que apunta el ***next*** del previo para eliminar en su totalidad la existencia del nodo a borrar. En ambos casos, se realizan los cambios correspondientes en la información de la memoria para que esta se muestre correctamente.

Limitaciones

Esta implementación cuenta con varias limitaciones que afectan a diferentes componentes del sistema operativo. En los siguientes apartados se explicarán cada una de ellas en profundidad.

En principio, la memoria Buddy tiene un tamaño máximo fijo de 9MB, es decir, que no se puede modificar el tamaño de la memoria al iniciarla, sino que su valor es fijo e inamovible. Esto se debe a que el diseño de la misma solamente permite tener una memoria máxima de 8MB y se tiene otro Byte ocupado por el vector que representa el árbol binario que, en el peor caso, tiene 65.535 nodos.

En segundo lugar, el algoritmo de scheduling posee dos limitaciones: la primera es que no es un algoritmo desalojable, es decir, si se cambia la prioridad de un proceso, este no se ejecutará instantaneamente y la segunda es que no existe una lista en particular para los procesos bloqueados.
Para la primera situación, se decidió no hacerlo desalojable dado que existía un problema con el guardado del contexto debido a otra ***system call*** más allá de la del ***Timer Tick***. Por otro lado, el manejo de los ***quantums_left*** del proceso que se encontraba corriendo actualmente era un problema. Por esto, se decidió cambiar la prioridad una vez que el proceso termine una ráfaga de CPU.
Para la segunda situación, se decidió no crear una lista específicamente para los procesos bloqueados y mantenerlos en las listas presentes dado que resultaba muy costoso el manejo de los nodos en las operaciones de bloqueo y desbloqueo.

En tercer lugar, se puede apreciar que el pipe posee un máximo en la cantidad de caracterés que se pueden escribir, es decir, que no es infinito y es necesario que exista un lector para que el escritor pueda seguir escribiendo.

En cuarto lugar, se puede especificar una limitación presente en la implementación de semáforos. Si se utilizará un procesador ***multicore*** deberíamos usar un ***mutex*** para modificar el valor de los semáforos, dado que estamos accediendo a variables globales. Sin embargo, como se tiene un procesador ***singlecore*** no se tiene ese problema y se puede obviar ese ***mutex***.

En quinto, se puede detallar que el programa ***phylo*** posee la limitación de que solo se pueden agregar hasta un máximo de diez filósofos por decisión del equipo y también para darles correspondientes nombres a cada uno de ellos. De la misma manera que los semáforos, si el procesador fuera ***multicore***, se debería adicionar un ***mutex*** para acceder a las variables globales.

Por último, se debe mencionar que compilando el sistema operativo en el Docker provisto, como lo indica el manual de usuario, en Windows existe la posibilidad de que el mismo lanze una excepción al momento de ejecutarlo, o funcione de manera errática. No está clara la causa del problema, pero se supone que tiene que ver con diferencias en la virtualización encabezada por el contenedor de Docker. ***Es necesario destacar que compilando el sistema operativo en el contenedor de Docker provisto desde macOS, la falla no se presenta y todo funciona correctamente.***

Problemas encontrados

Durante la realización de este trabajo práctico se presentaron varios problemas que se detallarán en los siguientes fragmentos.

La mayoría de los problemas se presentaron en la implementación del algoritmo de scheduling. El principal surgió cuando se implementó la funcionalidad de poder matar un proceso meidnate una syscall que llamaba a `terminate_process()`, pero en ningún momento se subía otro proceso para que corra, entonces la terminal se moría. Cuando se encontró el error, se agregó la variable ***autokill*** que permite diferenciar cuando se termina un proceso por un llamado a la syscall kill y en ese caso se realiza un llamado forzado a `_int20h()` para poder correr el siguiente proceso que tenga status ***READY***. Otra de las dificultades que se afrontó está relacionada con el código presente en la función `context_switch()` puesto que se tuvieron que tenerse en cuenta muchos casos, ver las similitudes de los mismos y diseñar el código teniendo en cuenta todo esto.

El diseño del PCB (Process Control Block) también generó dificultades dado que en un principio se definieron ciertos elementos que debían formar parte de este, pero a medida que se fue complejizando la implementación se tuvieron que agregar algunos elementos para poder cumplir con los requerimientos. Algunos ejemplos de esto son: el atributo ***new_priority*** que me permite modificar la prioridad la siguiente vez que se ponga a correr el proceso y la tabla de descriptores.

En la implementación de pipes surgieron problemas cuando se está por cerrar el último ***file descriptor*** dado que no se colocaba un ***EOF*** (End Of File) y por lo tanto los lectores no sabían cuando para de leer el contenido del pipe.

Se considera pertinente mencionar el proceso de testeo del sistema operativo, considerando la limitación inexplicable acerca del entorno de compilación. Siempre compilando con el Docker provisto en un sistema macOS, se ha testeado el correcto funcionamiento de la imagen del sistema operativo en los siguientes entornos:

- QEMU desde macOS Monterey 12.6.1
- QEMU desde Windows 11 22H2
- QEMU desde el Docker provisto por la cátedra en un sistema Windows
- QEMU desde el Docker provisto por la cátedra en un sistema macOS

También se ha probado de correr una imagen de turtleOS compilada con el Docker provisto en un sistema Windows en los entornos anteriormente mencionados y se puede verificar la falla explicada en el apartado *Limitaciones*.

Si hubiera algún problema en la compilación, se puede acceder a una imagen de turtleOS precompilada desde la sección ***Releases*** de ***GitHub***.

Modificaciones realizadas en los test

Las modificaciones realizadas en los test no están relacionadas con la falta de funcionalidad solicitadas en la consigna. Sin embargo, se cree que es importante especificarlos para simplificar el entendimiento del código a la hora de visualizarlo.

test_mm

En primer lugar, se reciben argumentos, el primero de ellos es la cantidad de argumentos y la segunda es un vector de dos elementos: el nombre y la cantidad de memoria a solicitar. Esto genera que los chequeos de parámetros recibidos también sean modificados.

En segundo lugar, dentro del primer ***while*** se agregó un chequeo de que si ya no hay más memoria disponible, es decir, ***malloc*** devuelve ***NULL***, entonces aviso mediante un mensaje que no hay más memoria disponible.

Los demás cambios son de estilo de código y un mensaje al final para indicar que el testeo fue exitoso.

test_prio

En este testeo se realizaron dos modificaciones. La primera de ellas está relacionada con que se reciben dos argumentos, el primero es la cantidad de argumentos y el segundo es un vector de un elemento (nombre del programa). La segunda modificación está relacionada con los ***define*** que se realizan en el principio del código adaptándolo a nuestra implementación. Se cambiaron los valores de las prioridades ***LOWEST***, ***MEDIUM*** y ***HIGHEST*** y también se eliminó el ***MINOR_WAIT*** dado que no se utilizaba en ninguna región del código.

test_processes

En este testeo se realizaron dos modificaciones. La primera de ellas está relacionada con que se reciben dos argumentos, el primero es la cantidad de argumentos y el segundo es un vector de un elemento (nombre del programa). La segunda modificación está relacionada con la modificación del ***enum State*** que se encuentra en el principio del código. Las demás modifcaciones son cambios en los nombres de las funciones que se llaman para realizar el testeo.

test_synchro

Las moficaciones realizadas en este testeo están relacionadas con la adaptación del código a la implementación realizada y el agregado de mensajes con el fin de notificar que el testeo fue exitoso.

Citas de fragmentos de código utilizados

Algunas de las funcionalidades implementadas en este trabajo práctico se basaron en códigos presentes en repositorios públicos. Cabe remarcar que todos aquellos fragmentos de código que no son de la autoría del grupo tienen un comentario en la parte superior indicando este comportamiento. A continuación, se detallarán las citas de los códigos consultados.

Memory

Buddy Memory

La implementación del Buddy terminó siendo considerablemente diferente a las que fueron consultadas, pero resultaron de gran utilidad para idear el diseño realizado. A continuación se pueden ver links a los códigos consultados.

Buddy Memory Allocation Program | Set 1 (Allocation) - GeeksforGeeks

Prerequisite - Buddy System Question: Write a program to implement the buddy system of memory allocation in Operating Systems. Explanation - The buddy system is implemented as follows- A list of free nodes, of all the different possible powers of 2, is maintained at all times (So if total memory size is 1 MB, we'd have 20

https://www.geeksforgeeks.org/buddy-memory-allocation-program-set-1-allocation/

GitHub - evanw/buddy-malloc: An implementation of buddy memory allocation

The file buddy-malloc.c implements a buddy memory allocator, which is an allocator that allocates memory within a fixed linear address range. It spans the address range with a binary tree that tracks free space. Both "malloc" and "free" are O(log N) time where N is the maximum possible number of allocations.

https://github.com/evanw/buddy-malloc

evanw/**buddy-malloc**

An implementation of buddy memory allocation

1 Contributor · 1 Issue · 77 Stars · 21 Forks

Heap Memory

La implementación del Heap está basada en la implementación de FreeRTOS dada por la cátedra. A continuación se tiene un link a dónde se encuentra la bibliografía consultada:

FreeRTOS - Memory management options for the FreeRTOS small footprint, professional grade, real time kernel (scheduler)

The RTOS kernel needs RAM each time a task, queue, mutex, software timer, semaphore or event group is created. The RAM can be automatically dynamically allocated from the RTOS heap within the RTOS API object creation functions, or it can be provided by the application writer.

https://freertos.org/a00111.html

FreeRTOS-Kernel/portable/MemMang at main · FreeRTOS/FreeRTOS-Kernel

FreeRTOS kernel files only, submoduled into https://github.com/FreeRTOS/FreeRTOS and various other repos. - FreeRTOS-Kernel/portable/MemMang at main · FreeRTOS/FreeRTOS-Kernel

https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/main/portable/MemMang

Pipes

La implementación de Pipes está basada en el código presente en el siguiente link. Cabe destacar que fue una inspiración y que la cátedra la recomendó.

xv6-public/pipe.c at master · mit-pdos/xv6-public

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

https://github.com/mit-pdos/xv6-public/blob/master/pipe.c

mit-pdos/**xv6-public**

xv6 OS

18 Contributors · 0 Issues · 6k Stars · 3k Forks

Standard Library functions

Muchas de las funciones de ***strings*** fueron obtenidas de la web y en el código implementado por el grupo se aclaran todos los fragmentos copiados.

Análisis de código estático

Se corrieron dos analizadores de código estático: PVS-Studio y CPPCheck. A continuación se detallan las advertencias que fueron lanzadas por dichos analizadores que se consideran erróneas o falsos positivos.

El codigo ***V566*** que aparece cuando se realiza el análisis de PVS-Studio es un falso positivo. El mismo aparece en los siguientes archivos: `console_driver.c`, `kernel.c` y `bash.c`. Esto sucede dado que se utiliza una constante numérica como dirección de memoria y posteriormente se la *castea* a puntero para utilizarla con ese fin.

El CPPCheck lanza errores de que funciones están declaradas pero no están utilizadas. Esto es un falso positivo porque se llaman desde Assembler. Además, lanza advertencias de funciones de la libería estándar, entre las que podemos encontrar ***gets*** y ***fprintf***. Estos son falsos positivos, dado que estas funciones no son de la libería estandar, sino que son de la autoría del grupo.