UNIVERSIDAD SAN CARLOS DE GUATEMALA

CENTRO UNIVERSITARIO DE OCCIDENTE

DIVISIÓN CIENCIAS DE LA INGENIERÍA

CARRERA DE INGENIERÍA CIENCIAS Y SISTEMAS

LABORATORIO DE SISTEMAS OPERATIVOS 1

“SÉPTIMO SEMESTRE”

ING: FRANCISCO ROJAS

ESTUDIANTE: Mario Moisés Ramírez Tobar - 201830007

Trabajo: “Arboleda por procesos y semáforos”

FECHA: 28 de marzo de 2,021

INTRODUCCIÓN

En el siguiente trabajo se documentará el trabajo realizado para el proyecto de Sistemas operativos donde lo que se tratará de realizar es un manejo de procesos, e intercomunicación de procesos, y así lograr demostrar gráficamente este manejo de buena manera de los procesos.

Se realizará una investigación sobre lo que son los semáforos, los procesos, la memoria compartida y los pipes en c + +, que es el lenguaje de programación que se tomará como base para el proyecto mismo. Este se trata sobre árboles donde el “tallo” se refiere a los procesos principales, estos generan “ramas” donde son hijos, y estas ramas a su vez generan “hojas”, que son hijos de las ramas. Pero no solo es la generación de procesos, si no como dijimos el manejo de los colores de los procesos, en este caso representados por tallos, ramas y hojas.

Estos colores son cambiados por colores específicos o aleatorios, pero cada proceso debe llevar este manejo propiamente dicho, independientemente de todos los demás procesos.

OBJETIVOS

Objetivos Generales:

- Organizar, crear, eliminar procesos, y que cada uno de estos procesos realice actividades diferentes que se visualice gráficamente.

Objetivos Específicos:

- Eliminar, crear, y organizar procesos.
- Visualizar este manejo de procesos gráficamente.
- Compartir información entre procesos.

REQUERIMIENTOS TÉCNICOS

Requerimientos mínimos hardware:

- 500 Megabytes de RAM

Requerimientos mínimos software:

- OS de alguna distribución de linux (en mi caso kubuntu).
- QT
- C++, librerías necesarias para su uso correcto
    - stdlib.h
    - unistd.h
    - stdio.h
    - vector
    - sys/wait.h
    - signal.h
    - sys/types.h
    - shared_mutex
    - semaphore.h
- QT Creator (Community)

HERRAMIENTAS PARA EL DESARROLLO

Kubuntu 20.04:

El equipo de Kubuntu se complace en anunciar que se ha lanzado Kubuntu 20.04 LTS, con el hermoso KDE Plasma 5.18 LTS: simple por defecto, poderoso cuando es necesario.

Con el nombre en código "Focal Fossa", Kubuntu 20.04 continúa nuestra tradición de brindarle Computación amigable al integrar las últimas y mejores tecnologías de código abierto en una distribución de Linux de alta calidad y fácil de usar.

El equipo ha trabajado arduamente durante este ciclo, introduciendo nuevas funciones y solucionando errores.

Bajo el capó, ha habido actualizaciones para muchos paquetes centrales, incluido un nuevo kernel basado en 5.4, KDE Frameworks 5.68, Plasma 5.18 LTS y KDE Applications 19.12.3

C++:

C++ es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades de programación genérica, que se sumaron a los paradigmas de programación estructurada y programación orientada a objetos. Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. Existen también algunos intérpretes, tales como ROOT.

El nombre "C++" fue propuesto por Rick Mascitti en el año 1983, cuando el lenguaje fue utilizado por primera vez fuera de un laboratorio científico. Antes se había usado el nombre "C con clases". En C++, la expresión "C++" significa "incremento de C" y se refiere a que C++ es una extensión de C.

QtCreator:

Qt Creator es un entorno de desarrollo integrado (IDE) multiplataforma creado para la máxima experiencia de desarrollador. Qt Creator se ejecuta en sistemas operativos de escritorio Windows, Linux y macOS, y permite a los desarrolladores crear aplicaciones en plataformas de escritorio, móviles e integradas.

Descarga Qt.

El editor de código avanzado de Qt Creator le permite codificar en C ++, QML, JavaScript, Python y otros lenguajes. Cuenta con finalización de código, resaltado de sintaxis, refactorización y tiene documentación incorporada a su alcance.

INSTALACION Y CONFIGURACION DEL SOFTWARE

Instalación C++ Ubuntu:

```
# sudo apt update -
# sudo apt upgrade -
# sudo apt-get install build-essential -
```

Instalación Qt 5.9.1 en Ubuntu:

```
# sudo apt update -
# sudo apt upgrade -
# wget
http://download.qt.io/official_releases/qt/5.9/5.9.1/qt-opensource-linux-x
64-5.9.1.run -
# chmod +x qt-opensource-linux-x64-5.9.1.run -
# ./qt-opensource-linux-x64-5.9.1.run -
```

MARCO TEÓRICO

fork():

La llamada al sistema fork se utiliza para crear un nuevo proceso, que se denomina proceso hijo, que se ejecuta al mismo tiempo que el proceso que realiza la llamada fork () (proceso padre). Después de que se crea un nuevo proceso hijo, ambos procesos ejecutarán la siguiente instrucción después de la llamada al sistema fork (). Un proceso hijo usa la misma PC (contador de programa), los mismos registros de CPU, los mismos archivos abiertos que usa en el proceso padre.

No toma parámetros y devuelve un valor entero. A continuación se muestran diferentes valores devueltos por fork ().

Valor negativo: la creación de un proceso hijo no tuvo éxito.

Cero: devuelto al proceso hijo recién creado.

Valor positivo: devuelto al padre o la persona que llama. El valor contiene el ID de proceso del proceso hijo recién creado.

El proceso padre y el proceso hijo están ejecutando el mismo programa, pero eso no significa que sean idénticos. El sistema operativo asigna diferentes datos y estados para estos dos procesos, y el flujo de control de estos procesos puede ser diferente. Vea el siguiente ejemplo:

La llamada al sistema de la bifurcación crea un nuevo proceso. El nuevo proceso creado por fork () es una copia del proceso actual excepto por el valor devuelto. La llamada al sistema exec () reemplaza el proceso actual con un nuevo programa.

fork () crea un nuevo proceso duplicando el proceso de llamada.

El nuevo proceso se denomina proceso hijo. La llamada proceso se conoce como el proceso padre.

El proceso hijo y el proceso padre se ejecutan en una memoria separada

espacios. En el momento de fork () ambos espacios de memoria tienen el mismo contenido. Escrituras de memoria, asignaciones de archivos (mmap (2)) y desasignaciones (munmap (2)) realizado por uno de los procesos no afecta el otro.

El proceso hijo es un duplicado exacto del proceso padre excepto por los siguientes puntos:

- El niño tiene su propio ID de proceso único, y este PID no coincide con el ID de cualquier grupo de procesos existente (setpgid (2)) o sesión.
- El ID de proceso del padre del niño es el mismo que el del padre identificación de proceso.
- El niño no hereda los bloqueos de memoria de sus padres. (mlock (2), mlockall (2)).
- Process resource utilizations (getrusage(2)) and CPU time counters (times(2)) are reset to zero in the child.
- El conjunto de señales pendientes del niño está inicialmente vacío.
- El hijo no hereda los ajustes de semáforo de su padre.
- El niño no hereda los bloqueos de registros asociados al proceso de su padre (fcntl (2)). (Por otro lado, lo hace heredar fcntl (2) abrir archivo descripción bloqueos y rebaño (2) bloqueos de su padre.).
- El niño no hereda los temporizadores de su padre (setitimer (2), alarma (2), timer_create (2)).
- El hijo no hereda E / S asincrónicas pendientes operaciones de su padre (aio_read (3), aio_write (3)), ni ¿Hereda algún contexto de E / S asíncrono de su padre?.

Comunicación entre dos procesos usando señales en C

Una forma de comunicación entre los procesos padre e hijo se realiza mediante kill () y signal (), llamada al sistema fork ().

- fork () crea el proceso hijo a partir del padre. El pid se puede verificar para decidir si es el hijo (si pid == 0) o el padre (pid = id del proceso hijo).

- El padre puede enviar mensajes al hijo usando pid y kill().
- El hijo capta estas señales con signal() y llama a las funciones apropiadas.

Señales

Las señales son interrupciones de software entregadas a un proceso por el sistema operativo. El sistema operativo también puede emitir señales según el sistema o las condiciones de error. Hay un comportamiento predeterminado para algunos (es decir, un proceso se termina cuando recibe una señal SIGINT de interrupción presionando las teclas ctrl-C), pero este tutorial muestra cómo manejar la señal definiendo funciones de devolución de llamada para administrar la señal. Siempre que sea posible, esto permite cerrar archivos y realizar operaciones y reaccionar de la manera definida por el programador.

Tenga en cuenta que no se pueden manejar todas las señales.

ALGORITMO DE DEKKER

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable de turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

Existen cinco versiones del algoritmo Dekker, teniendo ciertos fallos los primeros cuatro. La versión 5 es la que trabaja más eficientemente, siendo una combinación de la 1 y la 4.

Primer algoritmo

Garantiza la exclusión mutua, pero su desventaja es que acopla los procesos fuertemente, esto significa que los procesos lentos atrasan a los procesos rápidos.

Segundo algoritmo

Problema interbloqueo. No existe la alternancia, aunque ambos procesos caen a un mismo estado y nunca salen de ahí.

Tercer algoritmo

La región crítica no garantiza la exclusión mutua. Este algoritmo no evita que dos procesos puedan acceder al mismo tiempo a la región crítica.

Cuarto algoritmo

Postergación indefinida. Aunque los procesos no están en interbloqueo, un proceso o varios se quedan esperando a que suceda un evento que tal vez nunca suceda.

INTERRUPCIONES Y SEÑALES

En esta sección analizaremos las formas en que dos procesos pueden comunicarse. Cuando un proceso termina de manera anormal, generalmente intenta enviar una señal que indica qué salió mal. Los programas C (y UNIX) pueden atraparlos para realizar diagnósticos. También la comunicación especificada por el usuario puede tener lugar de esta manera.

Las señales son interrupciones generadas por software que se envían a un proceso cuando ocurre un evento. Las señales se pueden generar de forma síncrona por un error en una aplicación, como SIGFPE y SIGSEGV, pero la mayoría de las señales son asíncronas. Las señales se pueden enviar a un proceso cuando el sistema detecta un evento de

software, como un usuario que ingresa una interrupción o parada o una solicitud de eliminación de otro proceso. Las señales también pueden provenir directamente del kernel del sistema operativo cuando se encuentra un evento de hardware, como un error de bus o una instrucción ilegal. El sistema define un conjunto de señales que se pueden enviar a un proceso. La entrega de señales es análoga a las interrupciones de hardware en el sentido de que se puede bloquear una señal para que no se entregue en el futuro. La mayoría de las señales provocan la terminación del proceso de recepción si el proceso no realiza ninguna acción en respuesta a la señal. Algunas señales detienen el proceso de recepción y otras señales pueden ignorarse. Cada señal tiene una acción predeterminada que es una de las siguientes:

- La señal se descarta después de ser recibida.
- El proceso finaliza después de que se recibe la señal.
- Se escribe un archivo principal, luego se termina el proceso
- Detenga el proceso después de recibir la señal

int kill (int pid, int signal) - una llamada al sistema que envía una señal a un proceso, pid. Si pid es mayor que cero, la señal se envía al proceso cuyo ID de proceso es igual a pid. Si pid es 0, la señal se envía a todos los procesos, excepto a los procesos del sistema.

Memoria compartida en C para Linux

La memoria compartida, junto con los semáforos y las colas de mensajes, son los recursos compartidos que pone unix a disposición de los programas para que puedan intercambiarse información.

En C para unix es posible hacer que dos procesos (dos programas) distintos sean capaces de compartir una zona de memoria común y, de esta manera, compartir o comunicar datos.

La forma de conseguirlo en un programa es la siguiente:

- En primer lugar necesitamos **conseguir una clave**, de tipo key_t, que sea común para todos los programas que quieran compartir la memoria. Para ello existe la función **key_t ftok (char *, int)**. A dicha función se le pasa un fichero que exista y sea accesible y un entero. Con ellos construye una clave que nos devuelve. Si todos los programas utilizan el mismo fichero y el mismo entero, obtendrán la misma clave. Es habitual como primer parámetro pasar algún fichero del sistema que sepamos seguro de su existencia, como por ejemplo "/bin/ls", que es el "ls" del unix.

  Para el entero, bastaría con poner un #define en algún fichero.h de forma que todos los programas que vayan a utilizar la memoria compartida incluyan dicho fichero y utilicen como entero el del **#define**
- Una vez obtenida la clave, se **crea la zona de memoria**. Para ello está la función **int shmget (key_t, int, int)**. Con dicha función creamos la memoria y nos devuelve un identificador para dicha zona.
  Si la zona de memoria correspondiente a la Clave key_t ya estuviera creada, simplemente nos daría el identificdor de la memoria (siempre y cuando los parámetros no indiquen lo contrario).
  El **primer parámetro** es la clave **key_t** obtenida anteriormente y que debería ser la misma para todos los programas.

  El **segundo parámetro** es el tamaño en bytes que deseamos para la memoria.

  El **tercer parámetro** son unos flags. Aunque hay más posibilidades, lo imprescindible es:

  - **9 bits menos significativos**, son permisos de lectura/escritura/ejecución para propietario/grupo/otros, al igual que los ficheros. Para obtener una

memoria con todos los permisos para todo el mundo, debemos poner como parte de los flags el número 0777. Es importante el cero delante, para que el número se interprete en octal y queden los bits en su sitio (En C, cualquier número que empiece por cero, se considera octal). El de ejecución no tiene sentido y se ignora.

- **IPC_CREAT**. Junto con los bits anteriores, este bit indica si se debe crear la memoria en caso de que no exista.

  Si está puesto, la memoria se creará si no lo está ya y se devolverá el identificador.

  Si no está puesto, se intentará obtener el identificador y se obtendrá un error si no está ya creada.

En resumen, los flags deberían ser algo así como **0777 | IPC_CREAT**

- El último paso poder usar la memoria consiste en obtener un puntero que apunte la zona de memoria, para poder escribir o leer en ella. Declaramos en nuestro código un puntero al tipo que sepamos que va a haber en la zona de memoria (una estructura, un array, tipos simples, etc) y utilizamos la función **char * shmat (int, char *, int)**.
  - El **primer parámetro** es el identificador de la memoria obtenido en el paso anterior.
  - Los **otros dos** bastará rellenarlos con ceros.
  - El puntero devuelto es de tipo char *. Debemos hacerle un "cast" al tipo que queramos, por ejemplo, (mi_estructura *)shmat (...);

    Esta función lo que en realidad hace, además de darnos el puntero, es asociar la memoria compartida a la zona de datos de nuestro programa, por lo que es necesario llamarla sólo una vez en cada proceso. Si queremos más punteros en la zona de memoria, bastará con igualarlos al que ya tenemos.

- Ya estamos en condiciones de utilizar la memoria. Cualquier cosa que escribamos en el contenido de nuestro puntero, se escribirá en la zona de memoria compartida y será accesible para los demás procesos.
- Una vez terminada de usar la memoria, debemos liberarla. Para ello utilizamos las funciones **int shmdt (char *)** e **int shmctl (int, int, struct shmid_ds *).**

  La primera función desasocia la memoria compartida de la zona de datos de nuestro programa. Basta pasarle el puntero que tenemos a la zona de memoria compartida y llamarla una vez por proceso.

  La segunda función destruye realmente la zona de memoria compartida. Hay que pasarle el identificador de memoria obtenido con shmget(), un flag que indique que queremos destruirla IPC_RMID, y un tercer parámetro al que bastará con pasarle un NULL A esta función sólo debe llamarla uno de los procesos.

Exclusión mutua

En un sistema multiprogramado con un único procesador, los procesos se intercalan en el tiempo (i.e. Round Robin) para dar apariencia de ejecución simultánea. Aunque no se consigue un procesado en paralelo real, y aunque se produce un sobrecargado en la cpu por el hecho de tener que cambiar de tarea constantemente, las ventajas de todo esto son muy elevadas. Ejemplo: avión-torre, chats, etc.

Uno de los grandes problemas que nos podemos encontrar es que el hecho de compartir recursos está lleno de riesgos. Por ejemplo,si dos procesos hacen uso al mismo tiempo de una variable global y ambos llevan a cabo tanto operaciones de lectura como de escritura sobre dicha variable, el orden en que se ejecuten estas lecturas y escrituras es crítico, puesto que se verá afectado el valor de la variable.

Concepto de condiciones de carrera:

- Situaciones en las que dos o más procesos leen o escriben en un área de memoria compartida y el resultado final depende de los instantes de ejecución de cada uno.
- Esto se soluciona impidiendo que más de un proceso acceda simultáneamente a las
- variables compartidas. Se soluciona garantizando la exclusión mutua.

Concepto de exclusión mutua.

- Consiste en que un solo proceso excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

Concepto de sección crítica.

- Es la parte del programa con un comienzo y un final claramente marcados que generalmente contiene la actualización de una o más variables compartidas.
- Para que una solución al problema de la exclusión mutua sea válida, se tienen que cumplir una serie de condiciones:

    - Hay que garantizar la exclusión mutua entre los diferentes procesos a la hora de acceder al recurso compartido. No puede haber en ningún momento dos procesos dentro de sus respectivas secciones críticas.
    - No se deben hacer suposiciones en cuanto a la velocidad relativa de los procesos en conflicto.
    - Ningún proceso que esté fuera de su sección crítica debe interrumpir a otro para el acceso a la sección crítica.
    - Cuando más de un proceso desee entrar en su sección crítica, se le debe conceder la entrada en un tiempo finito, es decir, que nunca se le tendrá esperando en un bucle que no tenga final.

Para solucionar el problema de la exclusión mutua vamos a tener tres tipos de soluciones:

- Soluciones software.
- Soluciones hardware.
- Soluciones aportadas por el Sistema Operativo.

Pipes:

Conceptualmente, una tubería es una conexión entre dos procesos, de modo que la salida estándar de un proceso se convierte en la entrada estándar del otro proceso. En el sistema operativo UNIX, las tuberías son útiles para la comunicación entre procesos relacionados (comunicación entre procesos).

- La tubería es solo comunicación unidireccional, es decir, podemos usar una tubería de modo que un proceso escriba en la tubería y el otro proceso lea desde la tubería. Abre una tubería, que es un área de la memoria principal que se trata como un "archivo virtual".
- El proceso de creación puede utilizar la tubería, así como todos sus procesos secundarios, para leer y escribir. Un proceso puede escribirse en este "archivo virtual" o canalización y otro proceso relacionado puede leer desde él.
- Si un proceso intenta leer antes de que se escriba algo en la tubería, el proceso se suspende hasta que se escriba algo.
- La llamada al sistema de tuberías encuentra las dos primeras posiciones disponibles en la tabla de archivos abiertos del proceso y las asigna para los extremos de lectura y escritura de la tubería.

Cuando usamos fork en cualquier proceso, los descriptores de archivo permanecen abiertos en el proceso hijo y también en el proceso padre. Si llamamos a fork después de crear una tubería, entonces el padre y el niño pueden comunicarse a través de la tubería.

CONCLUSIONES

- Es complicado encontrar información de cómo realizar un semáforo de memoria compartida, y cómo organizar los tiempos de cada proceso, para que estos no se bloqueen, y esto hace dificultosa la implementación de la misma.
- Se vuelve muy importante el realizar forks y como saber el cómo hacer que solo el padre cree forks, y los hijos no, ya que se hace con un ciclo fork.
- Los pipes bloquean lo que es los procesos, por lo cual es más difícil el usarlos, al igual que no es lo mejor.
- Los procesos al no compartir memoria, debemos utilizar una memoria compartida, ya sea utilizada por las mismas clases de C, o memoria por parte de la misma computadora.

RECOMENDACIONES

- Investigar bastante sobre lo que es la memoria y semáforos de una manera extendida, ya que es lo que puede causar más conflicto al implementar el proyecto, esto causando muchos conflictos.
- Los forks se bloquean en ciertos casos, por lo cual se aconseja el no usar pipes, ya que estos crean estos bloqueos.
- Los signals solo funcionan con métodos estáticos y con un entero de parámetro, esto haciendo complicado el uso y mensajes entre procesos.

- Obtener información sobre Qt, ya que es una librería gráfica muy extensa y completa, que ayuda a la realización de ventanas dinámicos e interesantes con C++, esto se facilita más en el QT creator, que funciona como IDE del mismo.

BIBLIOGRAFÍA

- http://www.chuidiang.org/clinux/ipcs/mem_comp.php
- https://users.cs.cf.ac.uk/Dave.Marshall/C/node24.html
- http://www.yolinux.com/TUTORIALS/C++Signals.html
- https://www.qt.io/product/development-tools
- https://www.geeksforgeeks.org/fork-system-call/
- https://ubunlog.com/qt-mas-qtcreator-ubuntu/
- https://hetpro-store.com/TUTORIALES/compilar-cpp-g-linux-en-terminal-leccion-1/
- https://kubuntu.org/news/kubuntu-20-04-lts-has-been-released/
- https://es.wikipedia.org/wiki/C%2B%2B
- https://www.geeksforgeeks.org/pipe-system-call/
- http://www.chuidiang.org/clinux/senhales/pruebakill.c.txt
- https://pubs.opengroup.org/onlinepubs/7908799/xsh/signal.h.html
- https://stackoverflow.com/questions/42311299/c-pipe-and-fork
- https://www.geeksforgeeks.org/exit0-vs-exit1-in-c-c-with-examples/
- http://www.lsi.us.es/cursos/seminario-1.html
- https://revistas.intec.edu.do/index.php/cite/article/download/1568/2171?inline=1
- https://stackoverflow.com/questions/45293951/how-can-i-resume-a-paused-child-process-from-its-parent
- https://www3.physnet.uni-hamburg.de/physnet/Tru64-Unix/HTML/APS33DTE/DOCU_004.HTM
- https://doc.qt.io/qt-5/search-results.html?q=jsoncpp
- https://doc.qt.io/qt-5/json.html
- https://doc.qt.io/qtcreator/creator-tool-chains.html
- https://doc.qt.io/archives/qt-5.9/qtcore-json-savegame-example.html
- https://www.aprendeaprogramar.com/cursos/verApartado.php?id=16008
- https://trucosinformaticos.wordpress.com/2010/12/05/como-usar-static-en-c-y-c/
- https://users.cs.cf.ac.uk/Dave.Marshall/C/node24.html
- https://w3.ual.es/~jjfdez/SOA/pract6.html
- http://sopa.dis.ulpgc.es/ii-dso/leclinux/interrupciones/senales/lec4_senales.pdf
- https://stackoverflow.com/questions/45293951/how-can-i-resume-a-paused-child-process-from-its-parent
- https://stackoverflow.com/questions/31305717/member-function-with-static-linkage
- https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Punteros
- https://poesiabinaria.net/2014/02/variables-compartidas-entre-procesos-hijos-en-c-fork/
- https://stackoverflow.com/questions/19172541/procs-fork-and-mutexes
- https://blog.martincruz.me/2012/09/obtener-numeros-aleatorios-en-c-rand.html

- https://www.cplusplus.com/reference/vector/vector/erase/