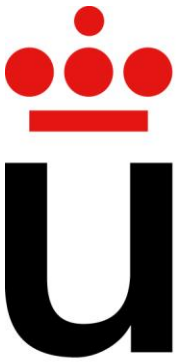

Universidad
Rey Juan Carlos

Sistemas Operativos

[PRÁCTICA 2 - THREADS]

RAUL FAUSTE Y ALVARO EGIDO

TABLA DE CONTENIDO

Autores

Raúl Fauste:

-Titulación: Doble Grado en Ingeniería Informática y Matemáticas

-Curso: 3º

-Asignatura: Sistemas Operativos

Álvaro Egido:

-Titulación: Doble Grado en Ingeniería Informática y Matemáticas

-Curso: 3º

-Asignatura: Sistemas Operativos

Descripción del Código

Diseño del Código

ESQUEMA GENERAL DEL CÓDIGO:

1-FICHEROS DE LECTURA/ESCRITURA

Disponemos de dos archivos de texto, uno de entrada de donde se recibirán los datos para poder realizar los cálculos pertinentes, y otro donde se guardará la salida del programa (habitantes eligiendo y vacunándose en un centro, fábricas creando vacunas y entregándolas, y las estadísticas finales. Por lo tanto, lo primero que se hace es comprobar que existe alguno de los dos archivos o nos lo están pasando por argumentos, abrimos el de entrada con permisos de lectura y el de salida con permisos de escritura, comprobamos que no haya error al abrirlos y leemos los datos correspondientes. Para el archivo de salida se basa en escribir en el fichero cualquier dato que se escriba por pantalla anteriormente. Y, posteriormente, mostramos estos datos de entrada por pantalla (a la vez escribiéndolo en el fichero, como hemos dicho que funciona anteriormente).

2- CREACIÓN y REPARTO de VACUNAS. LAS FARMACIAS

Cada farmacia está diseñada con un struct y un hilo. El struct, llamado farmacia, contiene dos atributos que nos indican el número de vacunas que deben fabricar cada farmacia (objetivo), y de qué farmacia se trata (num). Después, se crean los hilos, uno por farmacia, donde van a trabajar en concurrencia en la función crearVacunas y se van a diferenciar unas de otras porque le pasamos por argumentos dicho struct. Cabe indicar que, si el número de vacunas no es divisible entre tres, para no perder vacunas el resto de la división se la asignamos por convenio a la primera farmacia (en el atributo objetivo).

Nos ubicamos en la función crearVacunas, donde a través de un while y un acumulador de vacunas creadas indicamos que hasta que la farmacia en la que estamos trabajando no fabrica todas las vacunas marcadas en el objetivo, no termine de fabricar. Primero, se indica el número de vacunas que va a crear, a través del rand entre las vacunas mínimas y máximas indicadas en el archivo de entrada y después el tiempo que va a tardar en fabricarlas, con un rand entre el tiempo de fabricación mínimo y máximo, y con un sleep simulamos ese tiempo, “durmiendo” al hilo y que no pueda seguir. Cabe destacar que, si llegando al final justo el random de crear vacunas se pasa del objetivo, lo controlamos modificando el número de vacunas creadas por justo las que le faltan para llegar al objetivo, para no tener más vacunas creadas que el objetivo marcado. Ahora, mostramos por pantalla que dicha fábrica va a crear x vacunas, y para mostrarlo en el fichero, como las tres farmacias trabajan en concurrencia le metemos un mutex para que vayan escribiendo de una en una y no tengamos problemas. Almacenamos en el acumulador las vacunas que van creadas (en numVacCreadas) y las enviamos a los centros.

Para enviar las vacunas a cada centro, primero indicamos un tiempo de reparto aleatorio entre 1 y el tiempo de reparto indicado en los datos y con un for de 0 a 5, vamos repartiendo según la demanda a cada centro. Primero, controlamos que en dicho centro hay demanda de vacunas, ya que, si no hay, en este momento no se le reparte nada. Después, controlamos que ese valor no sea mayor que el numero de vacunas que hay en el centro, básicamente para la hora de realizar operaciones tengan sentido y no otorguemos vacunas negativas. Creamos un array aumentado que vaya guardando la demanda y “enviamos las vacunas”, es decir, restamos a las vacunas totales la demanda del centro y aumentamos las vacunas del centro por la demanda. Cada vez que tocamos la variable demanda, necesitamos un mutex, pues como pueden haber más de un paciente modificando dicha variable (veremos en el siguiente apartado), podemos tener problemas o datos incorrectos al estar en concurrencia. Por esto mismo existe el array auxiliar aumentado, para poder trabajar con el dato de la demanda sin tener problemas de concurrencia con los pacientes. Por lo tanto, cubrimos la demanda y la ponemos a cero para los siguientes pacientes. Cuando ya hemos cubierto la demanda y siguen quedando vacunas por repartir, lo que hacemos es seguir el mismo patrón que el anterior pero ahora en vez de cubrir la demanda las repartimos por igual en los cinco centros, con el convenio de que si no es divisible entre cinco repartir el resto al primer centro. Cuando ya el centro dispone de vacunas, manda una señal para despertar a los hilos (pacientes) y que se vacunen en los centros, utilizando el broadcast para despertar a todos a la vez. Para finalizar, cabe destacar que, como hay tres farmacias en concurrencia, para evitar problemas al añadir vacunas a

los centros se añade un mutex por centro, de esta manera nos evitamos el problema de que dos farmacias quieran añadir vacunas a la vez y como resultado que se acumulen mal. Finalmente, mostramos por pantalla y por el fichero de salida (con su mutex, como antes), que la fábrica ha entregado x vacunas al centro y, y del mismo modo cuando han terminado de fabricar todas lo mostramos por salida y por el fichero.

3- REPARTO FINAL DE LAS VACUNAS

Una vez todas las farmacias terminan de fabricar todas las vacunas, tenemos el problema de que van a haber centros con pacientes esperando a vacunas que no van a llegar nunca porque las farmacias no van a fabricar más. Para solucionar este problema, hemos optado por reorganizar las vacunas totales que tienen los cinco centros según su demanda a través de un hilo independiente, que ejecuta en concurrencia la función repartirFinal. Dicho hilo, va a estar esperando a través de un while a que las tres farmacias hayan terminado de fabricas todas sus vacunas. Luego, a través de un while que va a estar comprobando hasta que se vacunen todos los pacientes (su condición es que el numero de vacunados sea menor que los habitantes totales) iniciamos el reparto. Primero, tenemos que poner un sleep, por convenio el random de tiempo de reparto, para que a los pacientes le den tiempo a seguir aumentando la demanda en cada centro, porque si se hace de manera inmediata la demanda no va a aumentar y no se van a repartir bien. Después, a cada centro le realojamos las vacunas que necesitan para cubrir la demanda (utilizando los mutex igual que en el repartir normal anterior) y mandamos la señal pertinente para que los pacientes se vacunen.

4- VACUNACIÓN. LOS PACIENTES

Cada paciente se trata de un hilo, que trabaja sobre la función vacunarse a la que se pasa por parámetros un struct de la información del paciente, es decir, el número del paciente y el centro dónde se quiere vacunar. Para que cada hilo obtenga la información correctamente, se utiliza un mutex, de esta forma no se sobrescriben datos al trabajar todos a la vez. La vacunación se realiza en diez tandas, por lo tanto, se realiza un for de 10, y dentro otro for para que se vayan vacunando todos los de la tanda. Además, a través de un while, controlamos que no se avance de tanda hasta que todos los de dicha tanda se hayan vacunado. También dejamos controlado que, si dicho número no es divisible entre 10, por convenio el resto se vacunan en la primera tanda.

Entramos en la función vacunarse y dormimos al hilo con el tiempo de cita (random entre 1 y el tiempo de cita de los datos). Mostramos por pantalla y por el fichero que el habitante x elige el centro y volvemos a dormir el hilo con el tiempo de desplazamiento. Ahora, empezamos con la vacunación, pero primero, controlamos que si el centro donde se van a vacunar no tiene vacunas tenemos que aumentar la demanda de dicho centro y poner a esperar el hilo con un mutex con condición (5 centros, 5 mutex con 5 condiciones). Cuando ya hay vacunas en el centro, vacunamos al paciente restando una vacuna a las vacunas del centro y sumando una a los vacunados de la tanda, variable vista en el anterior apartado que nos permite avanzar de tanda. Cada variable que van a tocar más de un hilo a la vez, es decir la demanda, los vacunados de cada Tanda y las vacunas de cada centro, se utiliza un mutex para controlar que se modifican correctamente.

5- ESTADÍSTICAS FINALES

Cuando ya hemos terminado de vacunar a todos los pacientes, se debe mostrar cuántas vacunas ha fabricado cada farmacia, cuántas ha enviado a cada centro, cuántas vacunas han habido en cada centro, cuántos pacientes se han vacunado en cada centro y hemos añadido también cómo se han reubicado las vacunas finales tras el repartir final. Para ello, hemos creado unas variables acumuladoras donde vamos almacenando dichos valores. En estadísticaFarma vamos guardando las vacunas que manda cada fábrica a cada centro. Se trata de un array de 15, pues guardamos del 0 al 4 las vacunas de la farmacia 1 al centro 1,2,3,4 y 5; y así con el resto. EstadisticaPaci se trata de un array de 5 donde guardamos el número de pacientes que se han vacunado en cada centro. Finalmente, mostramos:

Threads utilizados: Cada paciente y cada farmacia se tratan de un hilo cada uno, y cuando terminan su función correspondiente se matan con el pthread_exit(0).

Estructuras de datos utilizadas:

Struct de las farmacias: guardamos de qué farmacia se trata y el número de vacunas que deben crear.

Struct de los pacientes: guardamos de qué paciente se trata y en qué centro se va a vacunar.

Array vacunadosCentro y arrayMutex: array donde guardamos las vacunas que tiene cada centro y su mutex correspondiente para poder modificar su valor correctamente.

Array hayVac: array de 5 donde se guardan las cinco condiciones de los cinco centros para avisar a los pacientes que ya tienen vacunas y pueden vacunarse.

Array demanda: array de 5 donde almacenamos la gente que está esperando en cada centro.

Array terminado: array de 3 donde es 0 si la farmacia sigue fabricando y 1 si ha terminado.

Array estadísticaFarma: array de 15 donde almacenamos las vacunas que manda cada fabrica a cada centro: 1-[0,4] 2-[5,9] 3-[10,14].

Array estadisticasPaci: array de 5 donde almacenamos el número de pacientes que se ha vacunado en cada centro.

Mutex mutexDema: para poder modificar correctamente la variable demanda.

Mutex mutexVacTanda: para poder modificar correctamente la variable vacunadosTanda.

Mutex mutexEstaPaci: para poder modificar correctamente la variable del array estadisticaPaci

Mutex mutexOut: para poder modificar correctamente el puntero de fichero de salida

Mutex v: para que la función de cada paciente lea correctamente su struct.

Condiciones de carrera y recursos compartidos:

Como ya hemos mencionado anteriormente, por cada recurso disponible en el código al cual varios hilos puedan acceder a él de manera simultánea (lo que se llama condiciones de carrera), va a existir un mutex para controlar dichos accesos y que se produzcan de manera correcta.

Para la creación del struct del paciente, modificar variables como la demanda, los vacunados en cada tanda (vacunadosTanda), el número de tandas que lleva el programa (tanda), que las fábricas accedan al puntero del fichero (fout) y para modificar las vacunas de cada centro y vacunarse (vacunasCentro). Para cada uno de ellos existe un mutex que regula su acceso y control.

Además, para controlar que se vacunen los pacientes solo si hay vacunas en su centro, se utilizan los mutex con condición. El paciente, cuando no hay vacunas en su centro, se queda esperando a recibir una señal. Dicha señal se manda cuando las fábricas han mandado vacunas a su centro.

Principales Funciones

Main:

	Nombre Función	Nombre	Tipo	Descripción
Argumentos	Argumento 1	argc	int	Número de argumentos
	Argumento 2	argv	char* []	Puntero a cada uno de los argumentos
Variables Locales	Variable 1	in	char []	buffer para almacenar el nombre del fichero de entrada
	Variable 2	out	char []	buffer para almacenar el nombre del fichero de salida
	Variable 3	dato	**int**	Variable para almacenar los datos leídos del fichero de entrada
	Variable 4	i	int	índice for
	Variable 5	j	int	índice for
	Variable 6	farm1	pthread_t	Hilo farmacia 1
	Variable 7	farm2	pthread_t	Hilo farmacia 2
	Variable 8	farm3	pthread_t	Hilo farmacia 3
	Variable 9	vacunasFinal	pthread_t	Hilo para repartir las vacunas al final
	Variable 10	paciente	pthread_t	Hilo del paciente a vacunarse
	Variable 11	p	struct paciente	Cada paciente con su número y el centro en el que se vacuna
Valor Devuelto			int	0-> fin del programa (correcto) 1,2-> fin del programa con errores
Descripción de la Función	Lee los datos de entrada del fichero, los muestra por pantalla y por fichero, crea todos los hilos, controla las tandas de vacunación y muestra por pantalla y por el fichero las estadísticas finales.			

-CrearVacunas:

	Nombre Función	**Nombre**	**Tipo**	**Descripción**
Argumentos	Argumento 1	arg	void*	Puntero al argumento que pasa el main
Variables Locales	Variable 1	farma	struct farmacia	Cada farmacia con el número de vacunas que tienen que fabricar y cuál es (su nº)
	Variable 2	numVacCreadas	int	Número de vacunas creadas por cada fábrica.
	Variable 3	aumentado	int []	Variable auxiliar para cada farmacia (array) para ir acumulando las demandas.
	Variable 4	mostrar	int []	Variable auxiliar para cada farmacia (array) para mostrar el nª de vacunas que entrega a cada centro.
	Variable 5	numVac	int	Nº de vacunas que se fabrican en ese momento.
	Variable 6	tiempoFab	int	Tiempo de fabricación.
	Variable 7	tiempoRep	int	Tiempo de reparto
	Variable 8	i	int	Variable para for
	Variable 9	vacuRepaIgua	int	Variable para repartir las vacunas restantes por igual tras satisfacer la demanda.
Valor Devuelto			void*	-
Descripción de la Función	Crea las vacunas de cada farmacia y las reparte de manera ininterrumpida hasta que las farmacias cumplen con su objetivo de vacunas fabricadas.			

-Vacunarse:

	Nombre Función	Nombre	Tipo	Descripción
Argumentos	Argumento 1	arg	void*	Puntero al argumento que pasa el main
Variables Locales	Variable 1	dentro	int	Variable para controlar que la demanda solo se aumenta una vez por paciente.
	Variable 2	paci	struct paciente	Cada paciente representado con su nº y el centro en el que se va a vacunar
	Variable 3	k	int	Almacenar el centro de cada paciente
Valor Devuelto			void*	-
Descripción de la Función	Cada paciente se vacuna en el centro en el que ha elegido y si no hay vacunas disponibles, espera hasta que haya a su disposición y se vacuna.			

-RepartirFinal:

	Nombre Función	Nombre	Tipo	Descripción
Variables Locales	Variable 1	h	int	Variable for.
	Variable 2	tiempoRep	int	Tiempo de reparto otra vez, ya que volvemos a juntar las vacunas que no sobran
Valor Devuelto			void*	-
Descripción de la Función	Una vez fabricadas todas las vacunas, estas se reorganizan entre los centros según la demanda para que ningún paciente se quede sin vacunar.			

Variables Globales y Recursos Compartidos

VARIABLE	TIPO	DESCRIPCIÓN
fout	FILE *	**Recurso compartido**, descriptor de fichero para mostrar la salida por fichero
hab	Int	Dato de entrada. Se necesita global para controlar los fors en los threads
vacI	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
vacMin	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
vacMax	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
fabMin	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
fabMax	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
tReparto	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
tCita	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
tDesp	Int	Dato de entrada. Se necesita global para hacer los rand() en cada thread
tanda	Int	Variable donde almacenamos por que tanda de vacunación vamos. La usamos para controlar el while de repartoFinal
vacunadosTanda	Int	**Recurso Compartido**. En ella vamos aumentado cuando vacunamos a un habitante. Cuando acaba una tanda se resetea
vacunasCentro	Int[5]	**Recurso Compartido**. En el vamos almacenando las vacunas que tiene cada centro en todo momento
demanda	Int [5]	**Recurso Compartido**. En el vamos almacenando los habitantes que tenemos esperando en cada centro en todo momento
terminado	Int [3]	Array donde almacenamos cuando los hilos de las fábricas han acabado
estadisticaFarma	Int [15]	Array donde vamos almacenando las vacunas que manda cada fábrica a cada centro
estadisticaPaci	Int [5]	**Recurso Compartido**. Array donde vamos almacenando cuantos

		habitantes se vacunan en cada centro. Se va aumentando cada vez que un habitante se vacuna en un centro concreto.
f1, f2, f3	struct farmacia	Structs donde almacenamos el número de cada fábrica/farmacia y el número de vacunas que debe fabricar cada una
v	pthread_mutex	Mutex con el que controlamos el struct de paciente
mutexDema	pthread_mutex	Mutex con el que controlamos el array de demanda
mutexVacTanda	pthread_mutex	Mutex con el que controlamos la variable vacunadosTanda
mutexEstaPaci	pthread_mutex	Mutex con el que controlamos el array de estadisticaPaci
mutexOut	pthread_mutex	Mutex con el que controlamos la variable fout
hayVac	pthread_cond [5]	Condiciones relacionadas con el array de Mutex que se explica debajo
arrayMutex	pthread_mutes [5]	Array de mutex para controlar cada uno de los campos del array de vacunasCentro

Casos de Prueba

Los casos de prueba que hemos ido probando en esta práctica no han sido más que ir variando los valores de la entrada estándar (que se nos proporciona en la práctica) para comprobar que todo se ejecuta correctamente.

Donde hemos puesto el foco principalmente ha sido:

-Reducir/Aumentar el tiempo de fabricación de vacunas

De esta forma comprobábamos si estábamos realizando mutex con condición y si estaban esperando los hilos como queríamos.

-**Variar el número de habitantes**

De esta forma comprobábamos que el programa funcionara, aunque el número no fuera divisible por diez (que es el número de tandas que se especifica que se han de tener).

Siendo la entrada estándar la siguiente:

-Número de habitantes totales – 1200

-Vacunas iniciales por centro – 15

-Número mínimo de vacunas producidas por una fábrica en una tanda – 25

-Número máximo de vacunas producidas por una fábrica en una tanda – 50

-Tiempo mínimo de fabricación de una tanda en una fábrica – 20

-Tiempo máximo de fabricación de una tanda en una fábrica – 40

-Tiempo máximo de reparto de vacunas – 3

-Tiempo máximo para que un habitante se dé cuenta – 4

-Tiempo máximo de desplazamiento a un centro de vacunación - 2

Comentarios Personales

Valoración de la práctica:

La práctica ha sido de gran utilidad para terminar de comprender el concepto de concurrencia y cómo está presente en el día a día y en prácticamente todos los programas que ejecutamos a diario y hasta ahora no nos habíamos parado a pensar en ello.

También nos ayuda a hacernos ver que todo lo que usamos es un poco más complejo que lo que en verdad nos parece y a los que nos gusta entender cómo funciona todo por debajo es un placer hacer este tipo de prácticas.

Problemas encontrados:

-**Concurrencia:** Al principio nos costó comprender que todos los hilos de los habitantes se ejecutaban a la vez y cómo lidiar con ellos aparte de controlar las condiciones de carrera que se nos producían.

-**Interbloqueos:** Tuvimos unos cuantos días donde el programa iba ciertas veces y otras no, hasta que descubrimos donde teníamos el interbloqueo paso un tiempo, pero nos vino bien para darnos cuenta de que alternar semáforos suele ser una señal de un posible interbloqueo.

Evaluación del tiempo dedicado

En comparación con la práctica 1 (Minishell) esta ha sido más llevadera, no nos ha quitado tanto tiempo y se ha hecho más amena. De todas formas, hemos tenido que dedicar un tiempo considerable para que tratar que todo salga lo mejor posible.

Aun así, no considero que el tiempo que hemos dedicado a la práctica (alrededor de semana y media) sea excesivo, me parece aceptable y más que suficiente (es posible que al haber trabajado la práctica 1 esta vaya más encauzada).

Crítica constructiva:

Nos hemos dado cuenta de que a veces, darse cuenta de donde estamos cometiendo los errores en el código puede ser muy tedioso, puesto que hay que esperar a que la ejecución termine y esto hace perder muchísimo tiempo.

También nos hemos dado cuenta de que si fuera posible debuggear el código de alguna manera estaría genial y si se contará en clase o se diera alguna pista de cómo hacerlo vendría muy bien. Hemos tenido que ir debuggeando mostrando por pantalla, así que, si es posible debuggear programas con threads, estaría genial que se contará un poco en clase si fuera posible.