Guía de programación de FSMEM

Versión 01.01

Fixed Size Memory

Un sistema robusto y eficaz de alojo de memoria dinámica y detección de fallos de acceso en tiempo real

Bayes Decision S.L.

Gran Vía, 39, 5ª planta
28013 Madrid (España)
Tel. (34) 915327440
Fax. (34) 915322636
www.bayesforecast.com
www.tol-project.org

Contenido

Resumen

La asignación de memoria dinámica tiende a verse como un sistema oscuro y mágico del que no hay que precuparse porque él ya sabe lo que tiene que hacer y lo hace siempre lo mejor posible. Veremos que en determinadas circunstancias esto no es cierto en absoluto y se darán distintas alternativas que mejorarán los resultados de forma drástica al mismo tiempo que permitirán el seguimiento de ciertos fallos de acceso inválidos de memoria que es en realidad el objetivo fundamental.

¡Atención!

Documento en construcción

Uso obligatorio de casco y botas

Copyright	2006, ***Bayes Decision S.L.***
Título	Guía de programación de FSMEM
Asunto	Fixed Size Memory
Categoría	Documento técnico
Archivo	C:\users\vdebuen\prj\tolp\trunk\doc\kernel\MemoryHandlers\FSMEM.odt
Edición	13/05/06 17:20:25
Claves	memoria dinámica, invalid access, memory leaks, garbage collector
Distribución	Pública

1. Descripción de los problemas de memoria en TOL

El problema fundamental en el mantenimineto de TOL es el seguimiento de accesos invalidos a memoria. Recientemente se han hecho varios intentos con Valgrind en linux y DUMA[1] (cross-plattform) llegándose a la conclusión de que estos sistemas de depuración son inviables para perseguir problemas que no estén convenientemente aislados para reproducirse en apenas décimas de segundo, porque suponen multiplicar por 1000 o más veces el tiempo de ejecución, convirtiendo prácticamente los segundos en horas.

Por otra parte la adopción de un sistema integral de gestión de memoria como MPS[2] sería aconsejable en la creación desde cero de un nuevo proyecto, pero es inasumible en el mantenimineto del TOL actual pues tal tarea debe acompoñar al proceso de diseño.

En Tol se generan diferentes problemas relacionados con la asignación de memoria

1. Asignación de objetos escalares de tamaño pequeño y mediano con estructuras de referencias secuenciales, arbóreas, reticulares y de todo tipo, como los nodos de las listas en general y de los árboles sintácticos y semánticos y toda la jerarquía de objetos y funciones TOL en general.
    1. Memory Leaks (ML): objetos redundantemente referenciados ocupan memoria para nada.
    2. Invalid Access (IA): objetos defectuosamante referenciados son destruidos prematuramente y provocan fallos de acceso posteriores.
    3. Excessive Searching (ES): cada llamada a new o malloc supone una búsqueda para evitar problemas de fragmentación de memoria lo cual no es muy oportuno si el tiempo de vida del objeto es muy breve. Estos objetos tienen un amplio espectro de esperanza de vida predominando normalmente los objetos de alta caducidad en el ámbito local y los de baja caducidad y perennes en el ámbito global. También hay que tener en cuenta que la memoria de estos objetos no se modifica nunca por ser de tamaño fijo, o sea, sólo se crean y se destruyen, nunca se modifican.
2. Asignación y reasignación por recrecimiento de objetos vectoriales
    1. Textos: Objetos de tamaño pequeño a mediano con actualización muy frecuente y especialmente en sentido creciente, debido al uso predominante de los operadores como << y +. Se produce un excesivo movimiento de bloques de memoria por el continuo recrecimiento.
    2. Arrays genéricos: Objetos de tamaño mediano a grande y muy grande pero con actualización de poca o mediana frecuencia, también en sentido

[1] http://duma.sourceforge.net/
[2] http://www.ravenbrook.com/project/mps/

creciente en la mayoría de los casos. Se usan fundamentalmente en Set, Matrix, Serie y en la caché de TimeSet. Pueden dar lugar a problemas de paginación de memoria virtual.

El punto 1.1 está aún sin tratar pero se propondrán tácticas de control para evitar el problema. El punto 1.2 es el que va a salir más fortalecido tanto en tiempo de depuración como en ejecución normal, aunque esto último aún no está implementado.

Los puntos 1.3, 2.1 y 2.2 son tres grupos de circunstancias diferentes entre sí que tienen en común sólo una cosa: el gestor de memoria estándar está diseñado para manejarlas de una forma bastante robusta y general pero no muy eficaz en estos casos.

Buscando un método de control y depuración de accesos inválidos se ha conseguido al mismo tiempo un método de gestión bastante más eficaz del punto 2.1 que podría utilizarse en algunos casos para los problemas vectoriales de tamaño pequeño, para los cuales, debido a su complejidad, se debería buscar una solución en las clases de la STL[3] std::string y std::vector respectivamente. Para los casos de objetos realmente grande se debería usar un gestor adecuado como la STXXL[4].

2.Descripción del método de asignación de memoria fija

Lo que pretende este método es dar todas las prestaciones y la flexibilidad de la memoria dinámica (HEAP) pero con velocidades próximas a la memoria estática (STACK) restando importancia a los problemas de *overhead*[5] derivados de fragmentación de tipo interno y de la informacion auxiliar precisada por el gestor, auqnue siempre de forma paramétrica y configurable.

La estructura básica interna es la página de memoria[6] que es capaz de almacenar un número `pageSize_` de objetos, con un máximo de 65535, en un campo `pool_` y todos ellos del mismo tamaño `baseSize_`. Ambas magnitudes no son modificables una vez creada la página.

El direccionamiento interno se realiza mediante una pila que almacena las direcciones libres en un array de `pageSize_` enteros cortos sin signo de 2 bytes. El asignador de memoria sólo tiene que dar la última dirección libre almacenada en el campo `available_`, mediante un método puramente aritmético sin apenas coste:

```
pool_+(avail_[--available_]*baseSize_)
```

La devolución de memoria es del mismo nivel de complejidad pues sólo es preciso anotar la posición que queda libre:

```
avail_[available_++] = int(address-pool_)/baseSize_
```

[3]http://en.wikipedia.org/wiki/Standard_Template_Library
[4]http://stxxl.sourceforge.net/
[5]desperdicio
[6]BFixedSizeMemoryHandler::BPage

En modo de depuración, además de la comprobación de accesos fuera de la página, se cuenta además con un array `assigned_` de booleanos[7] para poder comprobar si se realizan cierto tipo de accesos inválidos como doble borrado o doble asignación de una misma dirección.

La clase `BFixedSizeMemoryHandler` permite almacenar páginas del mismo tamaño base pero de longitud variable. Para permitir un rápido acceso sin desperdiciar excesiva memoria se crea primero una página de tamaño reducido especificado por el campo `_initPageSize_` cuyo valor por defecto es `BFSMEM_def_initPgSz_` y conforme se hace necesario se van creando páginas en orden de tamaño creciente en función de un factor de crecimiento dado por el parámetro `growingFactorPercent_` con valor por defecto `BFSMEM_def_grow_`.

El campo `pgNm_` almacena el número de página de la que se extrajo la última asignación a la que llamaremos página en curso, y `lastDelPgNm_` el de la página en que se hizo la última liberación. Cuando se solicita memoria se comprueba en primer lugar si en la página en curso es `available_>0` y si no es así se hace lo propio con la página `lastDelPgNm_`.

Si ambas páginas están completas, antes de proceder a buscar una página incompleta se comprueba si esto merece la pena realmente, observando si el número `assigned_` de objetos asignados es menor que cierto umbral `threshold_` calculado como un porcentaje `percentThreshold_` del total de memoria reservada entre todas las páginas, de forma que es posible decidir el porcentaje de fragmentación interna del gestor.

En caso contrario se procede a la búsqueda secuencial en orden inverso pues se ha observado empíricamente que es más probable que exista espacio libre en las páginas de más reciente creación. En cuanto se encuentra una página con al menos tantas posiciones libres como páginas en uso existen actualmente se designan la misma como página en curso. Si ninguna página cumple dicha condición se tomará como tal la que más posiciones libres contenga. Por último si todas las páginas están totalmente ocupadas o se había superado el umbral de asignación se procede a crear una página en curso nueva.

Además de la dirección de memoria asignada se devuelve el número de página en la que reside de forma que la liberación de memoria sea un proceso aritmético inmediato.

Mediante los templates BFSMSingleton<sizeof(myclass)> se generan de forma automática instancias únicas de los gestores de memoria apropiados para cada clase que se desee.

Queda pendiente un análisis exhaustivo de la sensibilidad del tiempo de proceso a los diferentes juegos de parámetros de configuración así como la creación de los métodos de control de los mismos desde el propio TOL para crear archivos de configuración óptimos para los diferentes tipos de programas.

[7]Hay que implementarlo como un array de bits en lugar de bytes para ahorrar espacio y usarlo también en modo release.

3.Asignación y reasignación de memoria para textos

En la siguiente tabla se muestra una análisis de frecuencia de las reasignaciones con recrecimiento de memoria durante la ejecución de un caso típico de programa TOL de estimación de un modelo ARIMA.

min bytes	max bytes	log(size)	number of reallocs	freq. of reallocs	cumulated freq.
2	3	1	137255	24,27%	24,27%
4	7	2	173772	30,73%	54,99%
8	15	3	162033	28,65%	83,64%
16	31	4	45492	8,04%	91,69%
32	63	5	26633	4,71%	96,40%
64	127	6	13611	2,41%	98,80%
128	255	7	3693	0,65%	99,46%
256	511	8	1818	0,32%	99,78%
512	1023	9	751	0,13%	99,91%
1024	2047	10	316	0,06%	99,97%
2048	4095	11	133	0,02%	99,99%
4096	8191	12	31	0,01%	100,00%
8192	16383	13	16	0,00%	100,00%
16384	32767	14	7	0,00%	100,00%
32768	65535	15	1	0,00%	100,00%

El tamaño inicial por defecto de las cadenas de texto se dispuso como 1 para almacenar el carácter nulo de fin de cadena. En cada registro de la tabla se muestra un bloque de tamaños en escala logarítmica de base 2 con el número de ocasiones en que se reasignó memoria de texto para un tamaño dentro de ese bloque, así como la frecuencia resultante y la acumulación de la frecuencia con los bloques anteriores. De ello se deduce que si se hubiera elegido un tamaño inicial por defecto de 32 bytes en lugar de 1, más del 90% de los realojos se hubieran evitado.