TLSF: новая система распределения динамической памяти для систем реального времени

M. Masmano, I. Ripoll, A. Crespo, J. Real
Universidad Politécnica de Valencia, Spain.
{mmasmano, iripoll, alfons, jorge}@disca.upv.es

Тезисы

Алгоритмы распределения динамической памяти (Dynamic Storage Allocation – DSA, далее используется аббревиатура РДП) играют большую роль в современных парадигмах и технике разработки ПО (например, в объектно-ориентированном программировании). Использование РДП повышает гибкость и функциональность приложений. В литературе подобные алгоритмы рассмотрены достаточно хорошо. В то же время использование РДП в системах реального времени рассматривалось как источник неопределённости ввиду отсутствия ограничений на время отклика алгоритмов РДП и проблем с фрагментацией памяти.

Сегодня новые приложения реального времени требуют большей гибкости: возможности изменения конфигурации системы в ответ на изменения в рабочем окружении и переконфигурировании приложения. Это обстоятельство заставляет вносить коррективы в проектирование и реализацию алгоритмов распределения динамической памяти.

Указанные соображения говорят о необходимости разработки новых алгоритмов распределения памяти, имеющих предельные и желаемые временные характеристики, пригодные для использования в операционных системах реального времени (ОСРВ). В настоящей статье представлен новый алгоритм РДП, именуемый Two-Level Segregated Fit memory allocator (TLSF) и разработанный специально для использования в ОСРВ. Алгоритм TLSF обеспечивает прозрачное выделение (далее также используются термины резервирование и распределение) и освобождение блоков памяти при временных затратах порядка $\Theta(1)$.

Ключевые слова: операционные системы реального времени, распределение памяти.

1 Введение

Изучение алгоритмов распределения динамической памяти (РДП) является важным моментом при разработке и реализации операционных систем и должно быть тщательно исследовано. Совместно с другими основными вычислительными проблемами, такими как поиск и сортировка, управление памятью является одной из самых исследуемых проблем. Уилсон и др. [14] выпустили отличный обзор о состоянии исследований в области распределения памяти с 1961 по 1995 годы, который можно рекомендовать как для ознакомления, так и в качестве пособия. Зная о существовании огромного количества алгоритмов РДП, читатель может решить, что проблема распределения динамической памяти давно уже решена. Для большей части приложений это может быть и так, но для приложений реального времени ситуация совсем иная.

При анализе пригодности системы реального времени нам необходимо заранее знать максимальное время реакции. Задачей алгоритмов РДП является предоставление приложениям требуемого объёма памяти, запрошенной в процессе работы. В целом эти алгоритмы спроектированы с целью обеспечить *хорошее среднее время отклика*, в то время как приложения реального времени требуют, чтобы время исполнения операций резервирования и освобождения памяти было *известным*.

Кроме того, управление динамической памятью при длительной работе приложения может вызвать значительную фрагментацию памяти, и, как результат, снижение надёжности вычислений.

Отметим важность того факта, что малое время отклика и высокая производительность являются теми параметрами, которые важны для любой системы, а для систем реального времени – тем более. Однако главным требованием, отличающим такие системы, остаются жёсткие временны**ы**е рамки. Большинство алгоритмов РДП спроектированы так, чтобы обеспечить наименьшее время отклика в большинстве возможных случаев, что даёт хорошую общую производительность, но в наихудшем случае время отклика может быть велико или даже бесконечным.

По этим причинам большинство разработчиков и исследователей ОСРВ избегают использования динамической памяти вовсе или используют её ограниченно, например, только во время загрузки системы: «Разработчики систем реального времени избегают использовать управление динамической памятью, так как они боятся наихудшего варианта при исполнении процедуры распределения динамической памяти, когда время исполнения процедуры неограниченно либо не укладывается в заданные границы». [10]

Настоящая работа представляет новый алгоритм – TLSF, - призванный распределять динамическую память с известным наихудшим временем исполнения и временн**ы**ми затратами порядка Θ(1). Кроме того, в случае продолжительной работы системы фрагментация также становится проблемой с точки зрения производительности. Предлагаемый алгоритм одновременно обещает небольшую и ограниченную фрагментацию.

В следующих разделах рассматриваются требования реального времени, которые должны выполняться алгоритмами РДП, чтобы они были применимы в приложениях реального времени. Раздел 3 описывает проблему фрагментации, появляющуюся вследствие работы алгоритмов. В разделе 4 даётся обзор типов распределителей динамической памяти и их стратегий. Раздел 5 даёт представление об общей модели поведения распределителей памяти. В 6-м разделе детально описываются критерии, использованные при разработке распределителя TLSF. Структура и характеристики TLSF рассмотрены в разделе 7. В 8-м разделе даётся оценка сложности алгоритма. В 9-м разделе сравниваются результаты,

полученные предложенным и конкурирующими алгоритмами. Наконец, в 10-м разделе мы подводим итоги и намечаем направления будущих работ.

2 Требования реального времени к алгоритмам РДП

Одной из ключевых проблем систем реального времени является исследование планирования с целью определения способности приложения уложиться во временные ограничения в процессе исполнения. Вне зависимости от используемых методов исследований и планирования необходимо определить наихудшее время исполнения (worst-case execution time - WCET) всего исполняющегося кода, включая код самого приложения, библиотечных функций и операционной системы.

Другим свойством, отличающим системы реального времени от всех прочих, являются приложения реального времени, работающие в течение продолжительного времени. Большинство обычных приложений (не реального времени) требуют всего нескольких минут или часов для того, чтобы выполнить задачу и завершить работу. Приложения реального времени обычно работают в течение всего жизненного цикла системы (месяцы, годы…). Это свойство напрямую определяет один из критических аспектов управления динамической памятью – проблему фрагментации памяти.

Учитывая все вышеуказанные особенности, мы можем суммировать требования, предъявляемые к приложениям реального времени применительно к динамической памяти:

- Заданное время отклика (Bounded response time). Наихудшее время исполнения при распределении и освобождении памяти должно быть известно изначально и не должно зависеть от приложения. Это основное требование.

- Малое время отклика (Fast response time). Кроме того, что время отклика задано, оно должно быть достаточно малым, чтобы алгоритм распределения памяти мог быть использован на практике. Алгоритм с известным временем исполнения, но в 10 раз медленнее прочих, в действительности никому не нужен.

- Требование на доступ к памяти всегда должно быть удовлетворено. Обычное приложение может получить нулевой указатель или просто быть закрытым ОС, если свободная память закончилась. И хотя очевидно, что не всегда возможно выделить запрошенную память, алгоритм РДП должен позаботиться о минимизации возможности утечки из кучи, уменьшив степень фрагментации и бесполезной потери памяти.

Говоря о требованиях к памяти, следует иметь ввиду, что существует множество систем реального времени, от небольших встроенных систем с небольшой памятью, не имеющих поддержки виртуальной памяти (MMU) и без долговременного накопителя, до больших резервированных многопроцессорных систем. Наше исследование и предложенный

алгоритм касаются, в основном, небольших систем с дефицитом памяти и без поддержки MMU.

Рассматривая способы управления распределением и освобождением [памяти], мы обнаружим, что РДП используют два основных подхода: а) явное распределение и освобождение, при котором приложение должно явно вызывать примитивы алгоритма для требования памяти (например, `malloc`) и для её освобождения (например, `free`), когда память больше не нужна; б) и неявного освобождения (также известного как *сборка мусора*), когда средствами РДП реализуется сбор блоков памяти, ранее запрошенных, но уже не нужных, с тем, чтобы вновь сделать их доступными для распределения. Мы сфокусируемся на явных низкоуровневых примитивах и оставим сборку мусора за рамками настоящей статьи.

3 Фрагментация

Кроме проблем с синхронизацией (вызванных нефиксированным временем отклика в конкретной реализации РДП), также возможны проблемы с распределением памяти из-за такого явления, как утечка. Утечки памяти случаются по двум причинам: а) приложение требует больше памяти, чем имеется в системе вообще; б) либо алгоритм распределения не позволяет повторно использовать ранее распределённую, но уже освобождённую память.

В первом случае алгоритм РДП бессилен, должно быть или переработано приложение, или увеличен объём памяти. Вторая причина исторически рассматривается в двух различных аспектах, известных как внутренняя и внешняя фрагментация. Новый подход [14] считает эти две категории фрагментации проявлением общей проблемы, известной как *утечка памяти* или просто *фрагментация*.

Фрагментацию следует рассматривать как серьёзную проблему. Многие ранние экспериментальные исследования основаны на искусственных или произвольно выбранных рабочих нагрузках, что привело к мнению о том, что проблема фрагментации может ограничить или даже сделать невозможным использование динамической памяти. Джонстоун (Johnstone) и его коллеги [4] показали, что проблема фрагментации в *реальных* (а не искусственных) условиях работы не является фатальной и может эффективно решаться грамотно спроектированными системами распределения памяти. Их заключение по этому вопросу звучит так: «в действительности проблема фрагментации – не более чем проблема плохо спроектированных распределителей памяти». Есть в этой работе и другие интересные выводы:

- политика подбора наиболее подходящего фрагмента [памяти] или первого подходящего фрагмента физической памяти даёт небольшую фрагментацию;

- блоки памяти должны объединяться сразу же после освобождения;

- желательно повторно использовать блоки памяти, возвращённые в систему после использования.

Практическим результатом работы Джонстоуна и его коллег [4] стала система распределения памяти Дуга Ли (Doug Lea) [6], использующая метод хорошо подходящего фрагмента (близкого к наилучшему подходящему фрагменту по методу разделённых фрагментов) и демонстрирующей очень хорошие показатели в плане фрагментации.

Несмотря на использование Джонстоуном обычных приложений в качестве рабочей нагрузки (gcc, ghostscript, perl и т.п.), у нас нет поводов думать, что приложения реального времени поведут себя слишком по-другому, чтобы свести полученные результаты на нет. Главное отличие приложений реального времени от обычных заключается в том, что многие приложения реального времени не используют динамическую память; память статически распределяется в первоначальный момент, что совершенно безопасно, но лишает гибкости.

4 Алгоритмы динамического распределения памяти

Назначение алгоритмов РДП – предоставление приложению доступа к блокам памяти из множества блоков свободной памяти. Различия в стратегии алгоритмов заключаются в различиях способов поиска свободных блоков памяти наиболее подходящего размера. Исторически алгоритмы РДП разделяются на следующие категории [по используемым стратегиям] [14]:

- **Последовательных фрагментов (Sequential Fit).** Это классические алгоритмы; они базируются на одинарных или двойных связанных списках свободных блоков памяти. В качестве примеров таких алгоритмов можно привести алгоритмы Fast-Fit, First-Fit, Next-Fit и Worst-Fit. Метод последовательных подходящих фрагментов не самый лучший с точки зрения систем реального времени, так как основан на последовательном переборе, затраты времени на который зависят от количества свободных блоков. На поиск могут быть наложены ограничения, но обычно эти ограничения неприемлемы.
- **Раздельных списков свободных фрагментов (Segregated Free Lists).** Алгоритмы на основе раздельных списков фрагментов используют массивы списков, содержащих ссылки на свободные блоки памяти определённого размера (или находящиеся в определённом диапазоне размеров). Когда блок памяти возвращается системе, он вставляется в список свободных блоков в соответствии с его размером. Особо отметим, что блоки разделены логически, но не физически. Существуют два варианта этого метода: (Simple Segregated Storage) и (Segregated Fit). Как пример

можно привести алгоритм Стендиша и Тадмана (Standish and Tadman) Fast-Fit, использующий массив списков для малых свободных блоков и двоичное дерево для списков блоков большого размера; и ещё алгоритм Дугласа Ли (Douglas Lea) [6]. Так как в этом случае затраты времени на поиск не зависят от числа свободных блоков, данный метод подходит для ОСРВ.

- **Метода близнецов (Buddy Systems).** Метод близнецов [5][9] является разновидностью метода раздельных списков свободных фрагментов с эффективными операциями выделения и освобождения памяти[1]. Существует несколько вариантов этого метода, например, Binary Buddies, Fibonacci Buddies, Weighted Buddies и Double Buddies.

  Метод близнецов демонстрирует хорошие временны**е** характеристики, вполне подходящие для применения его в РТОС; но у него есть и недостаток в виде существенной внутренней фрагментации, доходящей до 50%.

- **Индексированных фрагментов (Indexed Fit):** Эта стратегия основана на развитых структурах индексации свободных блоков памяти и имеет некоторые интересные черты. Приведём примеры алгоритмов, использующих метод индексированных фрагментов: алгоритм наилучших подходящих фрагментов (Best-Fit), использующий сбалансированные деревья, быстрый распределитель Стефенсона (Stephenson's "Fast-Fit"), основанный на декартовых деревьях (Cartesian tree) для хранения свободных блоков с двумя индексами [13], размеров, адресов и прочего. Метод индексированных фрагментов может выигрывать у метода раздельных списков при условии, что время поиска свободного блока не зависит от количества свободных блоков.

- **Битовых полей (Bitmap Fit).** Является разновидностью метода индексированных фрагментов и использует битовую карту для указания на занятые и свободные блоки. Примером этой стратегии может быть алгоритм половинных фрагментов (Half- fit) [8]. Такой подход имеет преимущество перед описанными выше, так как вся информация, необходимая для поиска, находится в небольшом объёме памяти, обычно в 32 разрядах, что позволяет снизить вероятность промаха кэша и улучшить время реакции [12].

---

[1] Разделение и слияние требуются для перестройки списков после обслуживания запросов на выделение и освобождение памяти.

5 Операционная модель РДП

РДП представляет собой некий абстрактный тип данных, позволяющий отслеживать состояние блоков памяти и знать, какой из них свободен, а какой – нет. Распределитель должен обеспечивать, по крайней мере, две операции: для распределения и для освобождения памяти. Характеристики РДП определяются, главным образом, структурой данных, используемой для управления блоками. Для лучшего понимания того, как влияет внутреннее представление данных на характеристики, посмотрим, как управляют памятью большинство распределителей динамической памяти:

- Изначально память, которой будет управлять РДП, представляет собой один большой блок свободной памяти, обычно называемый *первичной кучей* (*initial pool*). Также может случиться, что распределитель запросит у ОС, в которой он работает (или у аппаратного модуля управления памятью - MMU), дополнительную область памяти.
- Первый запрос на выделение памяти удовлетворяется путём выдачи блоков памяти из первичной кучи. Соответственно сокращается размер первичной кучи.
- Когда ранее выделенный блок возвращается обратно в систему, в зависимости от его физического местонахождения могут быть реализованы два варианта поведения:
    1. освобождённый блок может быть объединён с первичной кучей или одним или б**о**льшим количеством расположенных по соседству блоков; либо
    2. [этот] свободный блок окружён занятыми блоками и не может быть слит [с ними].

    В первом случае алгоритм РДП может объединить блок [с другими] или вернуть его в первичную кучу. Во втором случае алгоритм РДП вставляет свободный блок в структуру данных, описывающую свободные блоки.
- В том случае, если есть свободные блоки, запросы на распределение памяти могут удовлетворяться либо за счёт изначальной кучи, либо поиском достаточно большого свободного блока, не меньшего запрошенного объёма. Внутренняя структура данных, хранящая свободные блоки, и метод поиска подходящего блока – суть сердце РДП и определяют его характеристики. Если свободный блок, использованный для удовлетворения запроса о выделении памяти, превышает запрошенный размер, блок разделяется и нераспределённая часть возвращается в структуру данных свободных блоков.

Программы, использующие динамическую память, требуют выполнения двух операций: `malloc()` для динамического выделения блока памяти, и `free()` для освобождения ранее затребованного блока. Эти операции обслуживаются алгоритмом РДП,

от которого требуется обеспечить базовые операции, необходимые для управления свободными блоками (вставка, удаление или поиск). Рассмотрим упомянутые операции подробнее:

Вставка свободного блока: востребована обеими – `free` и `malloc` – функциями. В первом случае вставляемый блок является одним из освобождённых или наибольшим из тех, что были объединены. Для `malloc` вставляемый блок является остатком разделённого блока, размер которого превышал запрошенную величину.

Поиск свободного блока заданного или б*о*льшего размера: находит свободный блок, достаточно большой, чтобы удовлетворить требования приложения. Как мы уже заметили ранее, критерии, используемые при поиске блока (первый подходящий, наилучший подходящий и т.д.), и тип структуры данных в наибольшей степени определяют характеристики РДП.

Поиск смежного блока: при объединении свободного блока с другим(и) необходимо найти смежный по расположению в физической памяти свободный блок (если они существуют).

Удаление свободного блока: эта операция также востребована как `free`, так `malloc`. В первом случае это требуется, когда свободный блок можно объединить, и тогда соседний свободный блок удаляется и оба «первичных» свободных блока становятся новым единым свободным блоком. Второй случай имеет место, когда `malloc` находит подходящий блок в структуре данных свободных блоков.

6 Критерии, принятые при разработке TLSF

Не существует какого-либо универсального алгоритма РДП, одинаково пригодного для разнотипных приложений. Как уже было сказано, приложения реального времени совершенно не похожи на обычные приложения. В этом разделе мы расскажем о новом алгоритме двухуровневых списков разделённых фрагментов (Two-Level Segregated Fit, сокращённо TLSF), удовлетворяющем большинству требований реального времени: заданное и малое время реакции, известная и малая фрагментация.

Более того, для встроенных систем реального времени должны быть соблюдены дополнительные условия:

- Дружелюбное окружение (Trusted environment): программисты, имеющие доступ к таким системам не считаются враждебно настроенными, то есть они не имею намерений умышленно воровать или искажать данные приложения. Защита

осуществляется на уровне интерфейса конечного пользователя, но не на уровне программиста.

- Физической памяти очень мало.
- Для реализации виртуальной памяти не предусмотрено специального аппаратного модуля управления (MMU).

Чтобы совместить данные ограничения и требования, мы, разрабатывая TLSF, руководствовались следующими положениями:

Немедленное объединение блоков: Как только блок памяти вернётся обратно в систему, TLSF сразу же объединит его с соседними (если они существуют) блоками, чтобы получить крупный свободный блок. Другие алгоритмы РДП могут откладывать слияние блоков на более позднее время или же не проводить этой операции вовсе. Стратегия отложенного слияния полезна в системах, приложения в которых регулярно пользуются блоками одного размера. В этом случае отложенное слияние позволяет избежать постоянного проведения весьма затратных операций объединения и разделения блоков.

Хотя отложенное слияние и позволяет улучшить характеристики РДП, появляется фактор неопределённости (при очередном запросе может потребоваться объединить неизвестное количество свободных, но ещё необъединённых блоков), а также увеличивается фрагментация. То есть в системах реального времени стратегия отложенного слияния использоваться не должна.

Порог разделения: Минимальный блок распределяемой памяти занимает 16 байт. Большинство приложений, и приложения реального времени здесь не являются исключением, резервируют память не под простые типы данных вроде целых чисел, указателей или чисел с плавающей точкой, а под более общие структуры данных, содержащие, по крайней мере, один указатель и набор данных.

При ограничении минимального размера блока 16-ю байтами имеется возможность хранить внутри блоков информацию (включая список указателей на свободные блоки), необходимую для управления ими. Такой подход позволяет оптимизировать использование памяти.

Стратегия хорошо подходящих блоков: TLSF *попытается* выделить минимальный объём памяти, достаточный для размещения в нём блока требуемого размера. Поскольку большинство приложений используют, как правило, блоки памяти из небольшого диапазона размеров, метод наиболее подходящих фрагментов (Best-Fit) приводит к минимальной фрагментации в реальных системах по сравнению с другими стратегиями, например, в сравнении с методом первого подходящего фрагмента [14, 4]. Стратегия наилучшего подходящего фрагмента (или почти

наилучшего, также называемая стратегией *хорошо* подходящего фрагмента) может быть реализована эффективно и предсказуемо с помощью раздельных списков свободных блоков.

С другой стороны, реализация таких стратегий, как метод первого подходящего фрагмента или следующего подходящего фрагмента как алгоритмов с предсказуемым поведением, вызывает трудности. В зависимости от последовательности запросов алгоритм, работающий по методу первого подходящего фрагмента, может потерять в производительности, проводя поиск в длинной последовательности связанных списков.

В алгоритме TLSF реализован метод хорошо подходящего фрагмента с использованием большого набора списков свободных блоков, где каждый список является **неупорядоченным** списком свободных блоков, принадлежащих в зависимости от размера одному из классов (то есть, диапазону размеров). Каждый отдельный список состоит из блоков одного класса.

Отсутствие повторного выделения памяти: Мы условились считать изначальную кучу единым большим блоком свободной памяти, а функцию `sbrk()`[2] – отсутствующей. Так как многие ОС общего назначения (например, UNIX®) имеют механизм виртуальной памяти, большинство алгоритмов РДП были разработаны так, чтобы использовать все предоставляемые этим механизмом преимущества. Используя этот способ организации памяти, можно создавать РДП, оптимизированные для управления относительно небольшими блоками памяти, предоставляя операционной системе заниматься крупными блоками, динамически расширяя память.

TLSF был разработан исключительно для управления динамической памятью, так что он может использоваться как приложениями, так и самой ОС при отсутствии аппаратной поддержки виртуальной памяти.

Одинаковый подход для блоков всех размеров: Для всех блоков, вне зависимости от затребованного размера, используется один метод распределения. Один из наиболее эффективных алгоритмов РДП – распределитель Дугласа Ли (Douglas Lea’s allocator) [6] – использует четыре разных стратегии в зависимости от затребованного размера: первого подходящего фрагмента, кэшированных блоков, раздельных списков и предоставленной системой возможностью расширения кучи. Этот тип алгоритмов не обеспечивает единообразного поведения, поэтому наихудшее время исполнения бывает обычно велико или даже с трудом поддаётся определению.

---

[2] sbrk() – системная функция для увеличения памяти, доступной приложению

Отсутствие очистки памяти: Ни изначальная куча, ни свободные блоки не заполняются нулями. Алгоритмы РДП, предназначенные для многопользовательских систем должны очищать память (обычно заполняется нулями), исходя из соображений безопасности. Выделение пользователю памяти без предварительной её очистки противоречит политике безопасности, так как вредоносное ПО может перехватить конфиденциальные данные.

И всё-таки мы решили, что TLSF будет работать в дружелюбном окружении, приложения для которого создаются проверенными программистами. Таким образом, инициализация памяти совершенно не имеет смысла, но при этом требует ресурсов. Практически программисту рекомендуется не использовать инициализированную память.

7 Структуры данных TLSF

В данном разделе описываются структуры данных, используемые предлагаемым алгоритмом распределения динамической памяти. Алгоритм TLSF пользуется механизмом раздельных списков свободных блоков для реализации стратегии хорошо подходящих фрагментов.

В общем случае механизм раздельных списков основано на использовании массивов свободных списков, где каждый массив содержит свободные блоки в соответствии с разбиением на классы по размеру. Чтобы увеличить скорость доступа к свободным блокам и для управления большим набором раздельных списков (для уменьшения фрагментации) массив списков представлен как двухуровневый массив. Массив первого уровня описывает свободные блоки, разбитые на классы, определяемые по степеням двойки (16, 32, 64, 128 и т.д.); массив второго уровня линейно разбивает каждый класс первого уровня на отрезки, количество которых (так называемый *индекс второго уровня* (*Second Level Index, SLI*)) определяется параметром, задаваемым пользователем (подробнее этот параметр будет рассмотрен в пункте 7.3). Каждый массив списков ассоциирован с битовой картой, указывающей на пустые списки [не имеющие свободных блоков] и списки, содержащие свободные блоки. Информация, относящаяся к конкретному блоку, хранится в нём самом.

На рис. 1. показана двухуровневая структура данных. Первый уровень – массив указателей, указывающих на списки свободных блоков второго уровня. В нашем случае, если посмотреть на битовую карту первого уровня, мы увидим, что доступны только свободные блоки из диапазонов $[2^6, 2^7[$ и $[2^{15}, 2^{16}[$ (*по всей видимости, правая открывающая квадратная скобка означает, что от верхней границы следует отнять единицу – прим.*

перев.). Битовая карта второго уровня делит каждый диапазон первого уровня на четыре отдельных списка.

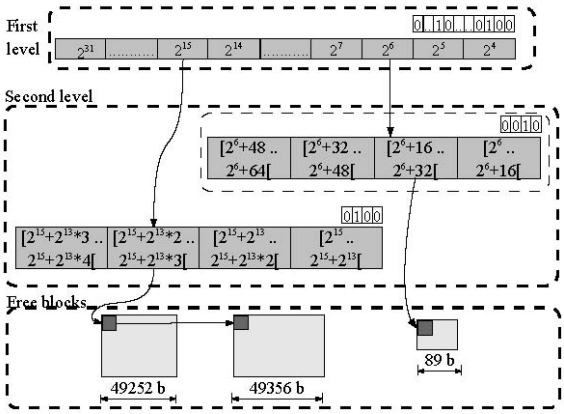

Рис. 1 Обзор структуры данных TLSF

Для большей лёгкости объединения свободных блоков в TLSF применена техника *связанного распределения памяти* (*boundary tag*), предложенная Д. Кнутом (D. Knuth) в [5], изначально предполагавшая добавление дополнительного поля (далее «подвал») к каждому свободному или занятому блоку с указателем на начало такого же блока. Когда блок освобождается (возвращается системе), «подвал» предыдущего блока (расположенного на одно слово ранее освобождённого блока) используется для доступа к заголовку предыдущего физического блока, чтобы узнать, свободен ли он, и, соответственно, объединить оба блока. Есть возможность упростить технику связанного распределения памяти, разместив указатель на заголовок блока не в конце каждого блока, а внутри заголовка следующего блока.

Таким образом, мы получаем ситуацию, когда каждые три блока связаны двумя разными двунаправленными списками: 1) раздельным списком, включающим блоки одного размерного класса и 2) список в соответствии с адресом в физической памяти.

7.1 Заголовок блока TLSF

TLSF хранит в каждом блоке информацию, необходимую для управления блоком (неважно, занят он или свободен) и указатели для связи в двух списках: списке блоков примерно равного размера и списке, отсортированном по адресам физической памяти. Эта структура данных называется *заголовком блока*.

Поскольку занятые блоки не включены в раздельные списка, их заголовки меньше, чем заголовки свободных блоков.

Заголовки свободных блоков содержат следующие данные (рис. 2): а) размер блока, необходимый для освобождения блока и для связи этого блока со следующим в списке адресов физической памяти; б) указатель связанной памяти для связи с заголовком

предыдущего блока в физической памяти; в) два указателя для интеграции блока в соответствующий раздельный список (двунаправленный связный список).

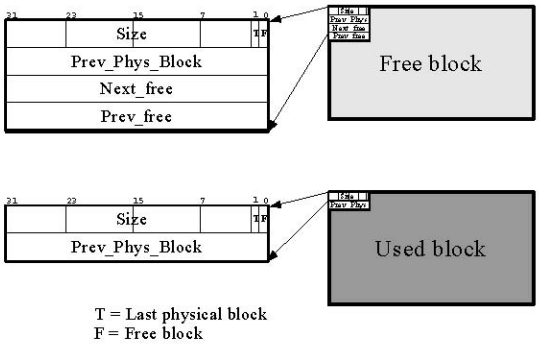

Рис. 2 Заголовки свободных и распределённых (занятых) блоков

Заголовок занятого блока содержит только его размер и указатель для связи в физической памяти.

Так как размеры всегда кратны четырём (единица распределения – четырёхбайтное слово), два младших значащих бита в поле размера используются для хранения статуса блока: блок занят или свободен (бит F) и является ли блок последним в куче или нет (бит T).

7.2 Структура функций TLSF

Большинство операций TLSF основаны на функции преобразования `segregate_list()`. Используя размер блока, функция вычисляет индексы в двух массивах, указывающие на соответствующий отдельный список, содержащий требуемый блок.

$$\text{mapping}(size) \rightarrow (f, s)$$

$$\text{mapping}(size) = \begin{cases} f := \lfloor \log_2(size) \rfloor \\ s := (size - 2^f)\dfrac{2^{SLI}}{2^f} \end{cases}$$

Эта функция может быть эффективно реализована при использовании инструкции для побитового поиска (имеется в большинстве современных процессоров) и знании некоторых свойств чисел. Индекс первого уровня ($\lfloor \log_2(size) \rfloor$) определяется как позиция наибольшего значащего разряда, равного единице, [если представить размер в двоичном виде]. Индекс второго уровня получается выделением следующих [после разряда, указывающего на индекс первого уровня, слева направо] SLI разрядов указанного размера блока. Положим, например, SLI = 4 и задан размер 460, тогда индекс первого уровня $f = 8$, а индекс второго уровня $s = 12$:

$$size = 460_d = \overset{15\,14\,13\,12\;11\,10\,9\;\overset{f=8}{8}\;7\,6\,5\,4\;3\,2\,1\,0}{0\,0\,0\,0\;0\,0\,0\;0\;\underbrace{0\,0\,0\,0}_{s=12}\;0\,0\,0\,0_b}$$

Структура TLSF обеспечивает проведение следующих операций:

- **Инициализация структуры TLSF:** эта функция создаёт структуру данных TLSF в начале распределяемой памяти* (оставшаяся часть памяти является изначальной свободной памятью). В структуру входят три параметра: указатель на [распределяемую] память, объём этой памяти и индекс второго уровня. Для TLSF существует возможность создания нескольких независимых объёмов распределяемой памяти.
- **Удаление структуры TLSF:** эта функция объявляет указанную распределяемую память неиспользуемой.
- **Получение свободного блока:** возвращает указатель на свободную область памяти требуемого или большего размера. Требуемый размер округляется в б**о**льшую сторону до ближайшего из имеющегося в списке свободных блоков. Эта операция работает так:

  Первый шаг: вычисляются индексы «*f*» и «*s*», используемые для получения первого элемента (head) свободного списка, содержащего (отдельный) список блоков, ближайших по классу. Если список не пуст (то есть содержит свободные блоки), то блок из начала списка удаляется из него (объявляется занятым) и выдаётся пользователю; иначе

  Второй шаг: ищет следующий (с блоками большего размера, чем запрошенный) непустой отдельный список в структуре данных TLSF. Время поиска постоянно при использовании битовой карты и инструкции для поиска первого установленного (ненулевого) разряда (find first set (ffs) bitmap instruction).
- **Вставка свободного блока:** эта функция вставляет свободный блок в структуру данных TLSF. Функция преобразования вычисляет индексы «f» и «s» для поиска отдельного списка, в который этот блок должен быть вставлен.
- **Объединение блоков:** при использовании технологии связанного распределения памяти, проверяется заголовок предыдущего блока, чтобы определить, свободен ли он. Если блок свободен, он удаляется из отдельного списка и объединяется с [возвращаемым в систему] указанным блоком. Такая же операция проводится и со следующим блоком физической памяти. Если блок может быть объединён с соседними свободными блоками, вновь полученный большой блок вставляется в соответствующий [по классу] отдельный список.

---

* в оригинале использован термин «pool», но, поскольку речь идёт именно о том объёме памяти, с которым работает РДП, будем называть его распределяемой памятью, чтобы избежать путаницы. – *прим. перев.*

7.3 Параметры описания структуры TLSF

Структура TLSF характеризуется тремя основными параметрами:

- **Индекс первого уровня (FLI):** номер отдельного класса первого уровня. Классы различаются по степеням двойки. FLI вычисляется динамически при инициализации TLSF по следующей формуле: $FLI = \min(\log_2(memory\_pool\_size),31)$.
- **Индекс второго уровня (SLI):** этот индекс определяет линейное деление на отрезки диапазона первого уровня. С целью повышения эффективности SLI является степенью двойки и должен находиться в диапазоне [1,32], чтобы можно было воспользоваться инструкциями процессора для работы с битами в коротких словах. Для удобства SLI вычисляется как $\log_2$ от количества отрезков во втором уровне (так, при SLI = 4 каждый отдельный класс первого уровня разбивается на 16 отдельных списков).

  Поскольку количество разных отдельных списков определяется значением SLI, то фрагментация тем меньше, чем больше значение SLI (см. раздел 8.1). Этот параметр устанавливается пользователем. Оптимальным значением можно считать 4 или 5.
- **Минимальный размер блока (MBS):** Этот параметр устанавливает предел минимального размера блока. По причинам, обусловленным особенностями реализации, MBS – величина постоянная и равняется 16 байтам.

Как производные от первичных параметров FLI, SLI и MBS могут быть вычислены некоторые дополнительные параметры:

- Общее количество списков: $2^{SLI}(FLI - \log_2(MBS))$.
- Диапазон размеров, принятый для каждого списка:

$$\left]2^i + \left(\frac{2^i}{2^{SLI}} \cdot j\right), next\_size\right]$$

$$\forall i, j \in \quad ^+ \left| \left(\log_2(MBS) \le i \le FLI\right) \wedge \left(0 \le j \le 2^{SLI}\right)\right.$$

и

$$next\_size = \begin{cases} 2^{i+1} - 1, \text{ если } \mathrm{j} = 2^{\mathrm{SLI}}, \\ 2^i + \left(\left(\frac{2^i}{2}\right)^{SLI} (j+1)\right), \text{если иначе.} \end{cases}$$

- Размер структуры данных:

$$TFH+(PS\cdot 2^{SLI}\cdot(FLI-\log_2(MBS))),$$

где TFH – фиксированный заголовок структуры данных (40 байт), PS – размер указателя (4 байта). Для максимального размера распределяемой памяти в 4 ГБ

(*FLI*=32) и максимального *SLI* (*SLI*=5) структура данных потребует 3 624 байта. Распределяемая память на 32 МБ (*FLI*=25) и *SLI*=5 потребуется 2 856 байт.

7.4 Оптимизация структуры TLSF

Для повышения эффективности структуры данных TLSF и операций поиска, можно провести некоторую дополнительную оптимизацию. Это возможно при использовании нескольких подходов:

- **Стратегия использования кэша и буфера быстрого преобразования адреса (Translation Lookaside Buffer - TLB):**
  - Структура данных TLSF организована как двухуровневый, а не двумерный, массив. Это позволяет получить преимущество в использовании кэша. Некоторые тесты показывают, что такая организация более эффективна, чем двумерный массив.
  - Кроме того, структура данных TLSF создаётся динамически в начале области распределяемой памяти. Хотя это отбирает часть свободной памяти, зато позволяет разместить структуру данных TLSF и саму распределяемую память на одной странице памяти.
- **Другие полезные подходы:**
  - Битовые карты, применяемые для описания непустых списков, являются причиной того, что FLI и SLI должны быть меньше или равны 32. Это позволяет уменьшить количество обращений к памяти и использовать не слишком много инструкций процессора для осуществления поиска.
  - Функция преобразования вычисляется как

```
Mapping (size_t size, unsigned *f1, unsigned *s1) {
   // fls() => Find_Last_Set bitmap function
   *f1 = fls(size);
   *s1 = ((size ^ (1<<f1)) >> (*f1 - SLI));
}
```

8 Исследование TLSF

Попробуем оценить временную сложность операций `malloc` и `free` с помощью псевдокода TLSF. Для `malloc` псевдокод будет таким:

```
void *malloc(size) {
   int f1, s1, f12, s12;
```

```
    void *found_block, *remaining_block;
    mapping (size, &f1, &s1);                          // O(1)
    found_block=search_suitable_block(size,f1,s1)      // O(1)
    remove (found_block);                              // O(1)
    if (sizeof(found_block)>size) {
       remaining_block = split (found_block, size);
       mapping (sizeof(remaining_block), &f12, &s12);
       insert (remaining_block, f12, s12);             // O(1)
       }
       remove (found_block);                           // O(1)
}
```

Хотя `malloc` должна осуществлять поиск по структуре TLSF (`search_suitable_block`), затраты на него асимптотически стремятся к O(1), так как поиск по структуре TLSF всегда имеет постоянное наихудшее время выполнения благодаря наличию инструкций процессора для поиска в битовой карте. Функция `remove` просто убирает ссылку на блок из отдельного списка подходящих фрагментов; функция `insert` осуществляет вставку в заголовок соответствующего отдельного списка.

Псевдокод для функции `free` выглядит, как показано ниже:

```
void free(block) {
   int f1, s1;
   void *big_free_block;
   big_free_block = merge(block);                      // O(1)
   mapping (sizeof(big_free_block), &f1, &s1);
   insert (big_free_block, f1, s1);                    // O(1)
}
```

Функция `merge` проверяет предыдущий и следующий блоки на предмет незанятости и пытается объединить их с освобождённым блоком. Так как освобождённый блок имеет ссылки на предыдущий и следующий физические блоки, поиск не требуется.

Все внутренние операции функций `malloc` и `free` имеют постоянные затраты на исполнение, не используют циклы или рекурсию, поэтому их наихудшее время исполнения асимптотически приближается к O(1) как `malloc()`, так и для `free()`.

8.1 Фрагментация

Фрагментация имеет место в TLSF постольку, поскольку он не обеспечивает поиск методом полного перебора в соответствующем отдельном списке, чтобы найти подходящий

блок для удовлетворения запроса. TLSF использует функцию преобразования и битовые карты, чтобы непосредственно находить непустые наименьшие отдельные списки, содержащие блоки того же или большего размера, чем затребованный. Если соответствующий список найден, то для удовлетворения запроса используется первый блок из списка. Таким образом, может случиться так, что существуют свободные блоки достаточно большого размера, чтобы удовлетворить требование на выделение памяти, но хранятся они в предшествующем отдельном списке (то есть в списке, расположенным перед тем, что вычислила функция преобразования).

Наихудший в плане фрагментации случай имеет место, когда наибольший свободный блок является максимально большим в своём списке (размер свободного блока на одно слово меньше, чем минимальный размер из следующего отдельного списка), а приложение просит блок, размером всего на одно слово больший, чем начальный размер в списке. TLSF попытается найти подходящий блок для удовлетворения запроса, начиная со следующего отдельного списка, имеющего свободные блоки, поэтому запрос не будет выполнен.

Фрагментация может быть вычислена по формуле $fragmentation = \frac{2^{FLI(pool\_size)}}{SLI} - 2.$

Двоичное представление интуитивно более понятно для описания источника фрагментации в TLSF. Пусть, например, SLI = 5, максимально доступный блок, который может быть в отдельном списке с индексами f = 12 и s = 13, равняется 5 887:

$$5887_d = 0\,0\,0\,\overset{f=12}{1}\,\underbrace{0\,1\,1\,1\,1}_{s=13}\,1\,1\,1\,1\,1\,1\,1_b \quad (15\,14\,13\;\;12\;\;11\,10\,9\,8\,7\;\;6\,5\,4\,3\,2\,1\,0).$$

Но поиск блока размером 5 761 байт ($5\ 761_{10} = 0001011010000001_2$) начнётся со списка с индексами f = 12 и s = 14, который является пустым. В нашем примере фрагментация составит 126.

Возможна модификация алгоритма TLSF для проведения поиска с точным соответствием (с возвратом к предыдущему отдельному списку), когда изначально установленное время поиска превышено. Хотя это улучшит ситуацию с фрагментацией, наихудшее время реакции больше не будет известным.

9 Сравнительный анализ

В литературе мы можем найти два разных подхода к оценке характеристик РДП. Один опирается на использование стандартных приложений (gcc, espero, cfrag и др.) или искусственных моделях рабочих нагрузок [16], основанных на трассировке реальных

приложений. Второй подход состоит в том, чтобы искусственно сконструировать рабочую нагрузку, воспроизводящую наихудший сценарий [11].

Принимая во внимание, что требования к приложениям реального времени отличаются от требований к обычным приложениям, результаты, получаемые при использовании стандартной нагрузки, не показывают поведения алгоритма в наихудших ситуациях. Поэтому для получения экспериментальных результатов, демонстрирующих важные для систем реального времени параметры производительности, была использована искусственная рабочая нагрузка. Каждая тестовая нагрузка подобрана так, чтобы воспроизвести наихудший сценарий для каждого распределителя. Мы тестировали следующие распределители памяти: First-Fit (первый подходящий фрагмент), Best-Fit (наилучший подходящий фрагмент), Douglas Lea's malloc (распределитель памяти Дугласа Ли), Binary Buddy (разновидность метода близнецов) и TLSF.

9.1 Создание рабочей нагрузки

Для алгоритмов, выделяющих первый попавшийся подходящий фрагмент и наилучший подходящий фрагмент, сценарии наихудшего случая похожи [11]. Такие случаи имеют место, когда списки свободных блоков заполнены (распределяемая память заполнена чередующимися свободными и занятыми блоками минимального размера) и запрос приложения на выделение памяти может быть удовлетворён только последним свободным блоком [в списке].

Алгоритм Дугласа Ли использует различные подходы в зависимости от размера блока: стратегию кэшированных блоков для небольших размеров (<64 Б); раздельных списков для блоков, меньших 512 Б; первого подходящего фрагмента (<128 КБ) и полагается на системный распределитель (`sbrk`) при требовании очень больших объёмов. То есть для него наихудший случай похож на таковой для стратегии первого подходящего фрагмента, но при размерах, больших 512.

Для алгоритма Binary Buddy наихудший сценарий имеет место, когда вся память свободна (представлена одним большим блоком) и приложение запрашивает блок наименьшего размера. Распределителю приходится разбивать память на $\log_2(pool\_size)$ фрагментов. Возвращение этого маленького блока также является наихудшим случаем для операции освобождения памяти.

Поскольку работа TLSF не зависит от количества свободных или занятых блоков и в коде нет циклов, мы будем иметь небольшие отличия во времени исполнения в зависимости от того, какая из ветвей алгоритма выполняется. Для `malloc` наихудший случай встречается при наличии одного большого свободного блока и запросе на выделение небольшого блока.

Для операции освобождения `free` наихудший сценарий имеет место, когда возвращённый блок должен объединяться с двумя соседними.

9.2 Результаты тестов

Измерения проводились на машине с процессором Intel Pentium P4 с частотой 2,8 МГц, объёмом кэша 512 КБ и с оперативной памятью объёмом 512 МБ. В таблице 1 приведены результаты замеров, выраженные в количестве циклов процессора. Каждый тест проводился 1024 раза для получения максимального и среднего значений. Содержание тестов:

Тест 1 – наихудший случай для операций `malloc` и `free` для алгоритма First-Fit. Объём распределяемой памяти 1 МБ. Вся память, за исключением последних 32 байт, сначала была распределена блоками минимального размера (16 байт), затем нечётные блоки были возвращены. После этого наихудший случай имеет место при запросе на выделение блока размером 32 байта.

Тест 2 является наихудшим случаем для операций `malloc` и `free` для алгоритма Дугласа Ли. Объём распределяемой памяти 256 КБ, проведена та же последовательность запросов, как для Теста 1, но размер блоков – 512 байт. Последний блок имеет размер 530 байт.

Тест 3 является наихудшим случаем для операций `malloc` и `free` для алгоритма Binary Buddy. Объём памяти – 2 МБ. Изначально блоки не распределялись. Наихудший случай встречается при запросе блока размером 16 байт.

Тест 4 – наихудший случай для операции `malloc` для алгоритма TLSF. Объём памяти – 2 МБ. Изначально блоки не распределялись. Объём запрашиваемого блока – 40 байт.

Тест 5 – наихудший случай для операции `free` для алгоритма TLSF. Объём памяти – 1 МБ. Распределены три блока по 512 байт, затем первый и третий возвращены. Время возвращения второго блока является наихудшим случаем.

Таблица 1. Результаты измерений для операций malloc и free. Результаты представлены количеством циклов процессора.

malloc()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	худший	средний	худший	средний	худший	средний	худший	средний	худший	средний
Тест 1	**25636208**	**11256641**	**17007384**	**10322776**	168	81	4140	1239	155	148
Тест 2	2124	1971	2124	1971	**16216**	**15974**	244	220	172	148
Тест 3	568	201	592	197	128	76	**5660**	**5448**	120	115
Тест 4	792	235	536	193	124	75	5460	5309	**189**	**168**

(продолжение таблицы 1)

free()	*First-Fit*		*Best-Fit*		*DL's malloc*		*Binary Buddy*		*TLSF*	
	худший	средний	худший	средний	худший	средний	худший	средний	худший	средний
Тест 1	528	103	2012	899	460	67	2728	345	140	97
Тест 2	428	150	428	150	**96**	**71**	1976	1448	152	96
Тест 3	340	107	324	113	560	131	**6512**	**3504**	124	93
Тест 4	348	157	372	204	136	68	3728	2881	**188**	**164**

Из таблицы 1 видно, что наихудшее время реакции для TLSF всегда ограничено и демонстрирует стабильные [временн**ы**е] характеристики. С другой стороны, разработанный для каждого распределителя сценарий наихудшего случая приводит к существенному ухудшению времени отклика. Исходные тексты тестов и более подробные результаты для различных FLI и SLI можно найти в [7].

10 Заключение и планы на будущее

Основные проблемы алгоритмов РДП, такие как неизвестное и небыстрое время реакции исполняемых примитивов и высокая степени фрагментации, являлись препятствием для использования РДП в системах реального времени. Некоторые алгоритмы, вроде метода близнецов, позволили улучшить некоторые моменты, такие как время отклика, но сохранили фрагментацию на неприемлемом уровне.

В настоящей статье мы представили алгоритм, названный TLSF, основанный на двухуровневых раздельных списках, обеспечивающих явное выделение и возвращение блоков памяти при временн**о**й сложности $\Theta(1)$. В ходе экспериментальных проверок предлагаемый алгоритм показал отличные результаты как по времени отклика, так и по фрагментации.

Использование битовых карт и раздельных списков для доступа к разным блокам свободной памяти разного размера эффективным и предсказуемым способом позволяет получить постоянные [временн**ы**е] затраты при исполнении базовых операций *malloc* и *free*. Кроме того, анализ предложенной структуры данных показывает низкую и ограниченную степень внутренней фрагментации, зависящую от величины индекса второго уровня.

Сравнение предлагаемой системы с другими РДП даёт впечатляющие результаты.

Предложенный РДП положен в основу проекта OCERA [3] и реализован в *RT-LinuxGPL* [15, 2] и *MARTE OS* [1].

Требуются дополнительные исследования, необходимые для изучения широкого спектра приложений реального времени как для получения данных о характеристиках

реальных приложений, так и для определения степени соответствия случайного кода требованиям, озвученным в разделе 2 настоящей статьи. Также целью будущих работ является внедрение этого алгоритма в систему управления памятью, что позволяет обеспечить качество обслуживания с учётом [возможности] интегрирования системы управления памятью и ЦПУ.

Благодарности

Мы благодарны Энди Уэллингсу (Andy Wellings), консультировавшему нас при подготовке последней версии статьи. Также мы благодарим всех, кто прочитав предварительный вариант, дали нам множество полезных советов. Наконец, мы выражаем признательность Герману Hrtig за его комментарии в обозрении OCERA о системах управления памятью, которые и подвигли нас провести настоящее исследование.

Литература

[1] M. Aldea and M. González-Harbour. MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. *Reliable Software Technologies – Ada Europe, Lecture Notes in Computer Science*, 2043:305–316, 2001.

[2] RT-Linux-GPL distribution. www.rtlinux-gpl.org.

[3] OCERA: Open Components for Embedded Real-Time Applications. 2002. IST 35102 European Research Project. (http://www.ocera.org).

[4] M.S. Johnstone and P.R. Wilson. The Memory Fragmentation Problem: Solved ? In *Proc. of the Int. Symposium on Memory Management (ISMM'98), Vancouver, Canada*. ACM Press, 1998.

[5] D.E. Knuth. *The Art of Computer Programming, volume 1: Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, USA, 1973.

[6] D. Lea. A Memory Allocator. *Unix/Mail*, 6/96, 1996.

[7] M. Masmano. TLSF Implementation and Evaluation in OCERA. Technical Report OCERA D4.4, Real Time Research Group. Universidad Politecnica de Valencia, 2004. Available at: http://www.ocera.org and http://rtportal.upv.es.

[8] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. *2nd Int. Workshop on Real-Time Computing Systems and Applications*, page 21, 1995.

[9] J.L. Peterson and T.A. Norman. Buddy Systems. *Communications of the ACM*, 20(6):421–431, 1977.

[10] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. *14 th*

Euromicro Conference on Real-Time Systems, page 41, 2002.

[11] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. Technical Report 1429, Institut de Recherche en Informatique et Systemes Aleatories, 2002.

[12] Filip Sebek. Cache Memories in Real-Time Systems. Technical Report 01/37, Mälardalen Real-Time Research Centre, Sweden, October 2 2001.

[13] C.J. Stephenson. Fast Fits: New Methods for Dynamic Storage Allocation. In *Proc. of the 9th ACM Symposium on Operating Systems Principles*, volume 17 of *Operating Systems Review*, pages 30–32. ACM Press, 1983.

[14] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In H.G. Baker, editor, *Proc. of the Int. Workshop on Memory Management, Kinross, Scotland, UK*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany, 1995. Vol:986, pp:1–116.

[15] V. Yodaiken and M. Barabanov. A real-time linux. Online at http://rtlinux.cs.nmt.edu/rtlinx/u.pdf.

[16] Benjamin Zorn and Dirk Grunwald. Evaluating Models of Memory Allocation. *ACM Transactions on Modeling and Computer Simulation*, pages 107–131, 1994.