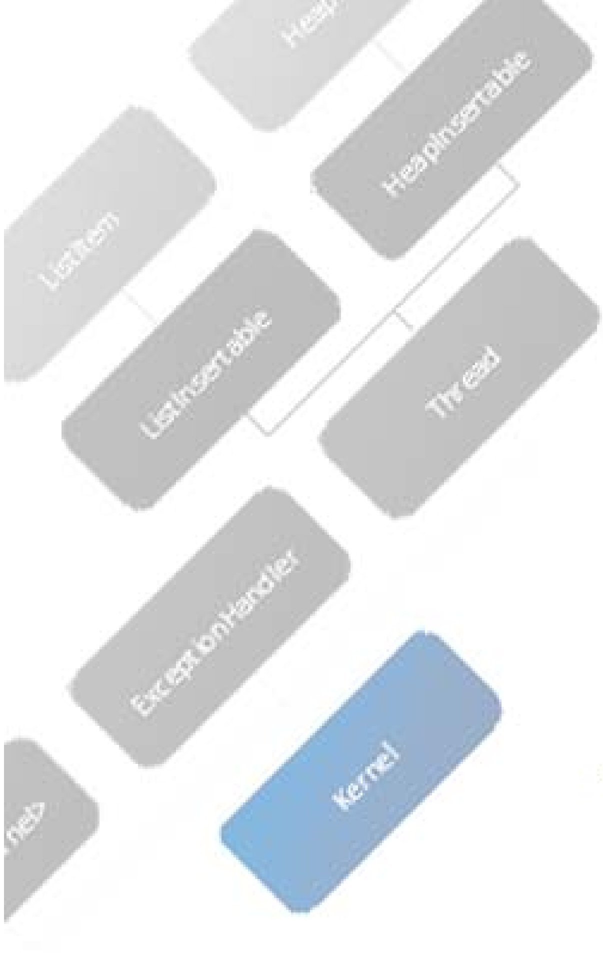

BunnyKernel

for MIPS

Documentation

```
       _     _
       \`\ /`/
        \ V /
        /. .\
       =\ T /=
        / ^ \
     {}/\\ //\
     __\ " " /__
jgs (____/^\____)
```

Matúš Dekánek, Tomáš Petrůšek,
Ľuboš Slovák, Ján Veselý
2008

Contents

Introduction

What is BunnyKernel?

BunnyKernel is an experimental operating system kernel implementation for MIPS R4000 processor. It was developed by the students of Charles University in Prague, Faculty of Mathematics and Physics. Its main aim as the semester work for Operating Systems[1] course is to explore the possibilities and demonstrate understanding of basic operating system programming techniques, the difference between application and kernel programming as well as the basic layout of an operating system.

The task was further simplified by using emulated environment provided by the MSIM software.[2]

[1] http://dsrg.mff.cuni.cz/~ceres/sch/osy/main.php

[2] http://dsrg.mff.cuni.cz/~holub/sw/msim/

Object-oriented design

Kernel and C++

The task of writing kernel of an operating system differs from the standard development of user applications, mostly because of the environment that the program should run in. User applications rely on the support of the operating system. This support includes, but does not restrict to, initialization before the application is started and clean-up after it has finished. However there is no such support when developing kernel and the decision to go for C++ language made this difference even greater.

C vs. C++

Unlike the rather straightforward C language C++ does a lot of work in less visible way. Calling constructors, destructors, operators, choosing which member function to run in inherited classes, are all done implicitly. For us the most important thing was the ability to develop using object oriented programming techniques. This required the presence of some linking symbols and function that would take care of the situations the compiler cannot. First example would be the `_cxa_pure_virtual()` function, which is called when a pure virtual member function is invoked. Others would include operators `new, new[], delete` and `delete[]`.[3] Other features like templates did not require any additional work, but some like C++ exceptions were disabled as it would require even more work with little gains for the kernel development.

BunnyKernel structure

OOP was one of the major gains we utilized in the development of the BunnyKernel. Separation of code into largely independent classes made not only the design clearer, but also enabled us to divide the work evenly contributing to the convenience of development.

The kernel itself consists of several standalone (Singleton) classes that provide and use services of each others. These would be:

- `Kernel` used for initialization
- `Scheduler` schedules threads and chooses next thread to run
- `FrameAllocator` guarding access and distributing physical memory
- `KernelMemoryAllocator` provides Heap allocation within kernel space

[3] More info can be found here: http://wiki.osdev.org/C_PlusPlus

Overall look at the structure

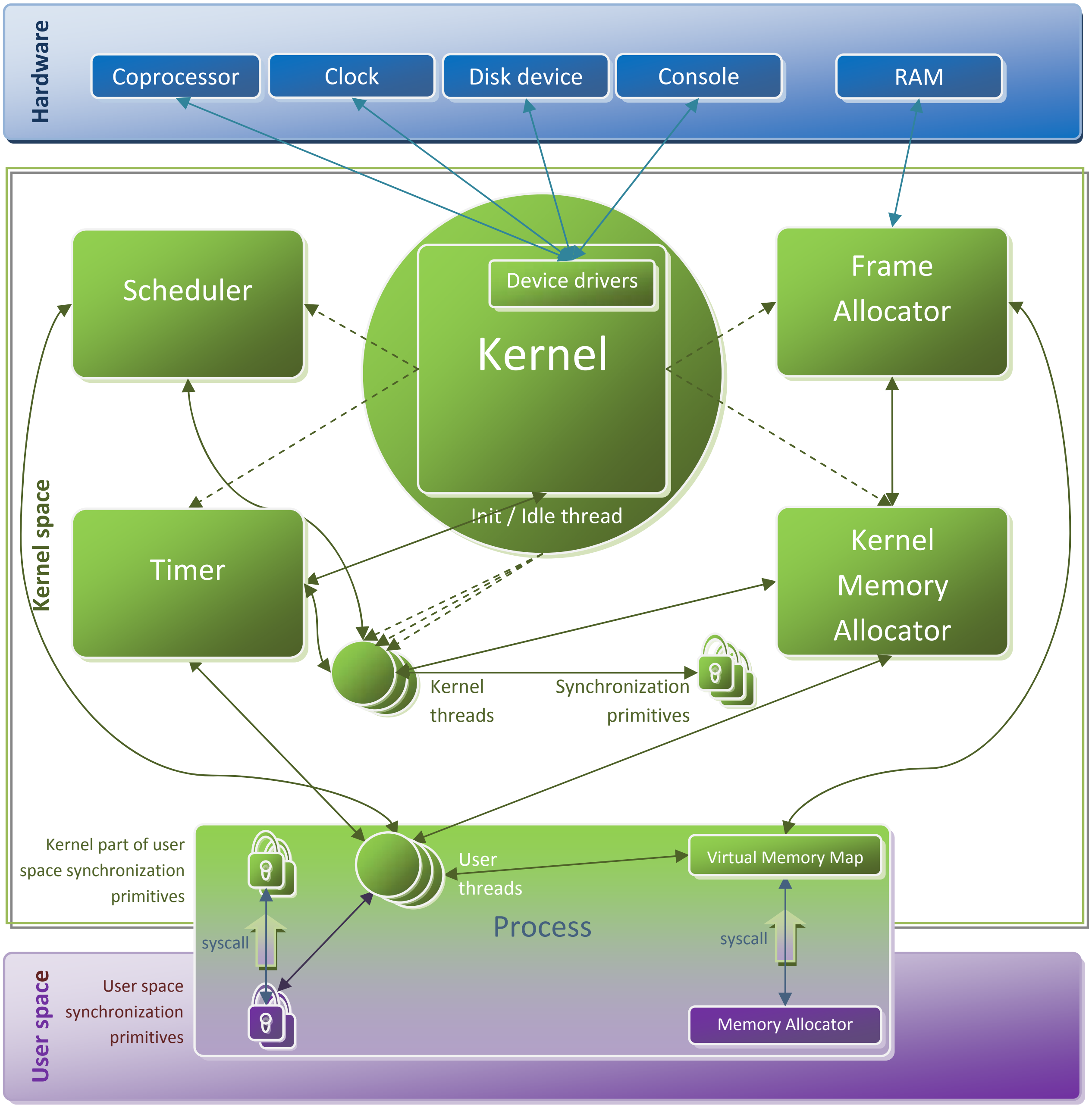

Figure 1 - Internal structure of the BunnyKernel

Class hierarchy – examples

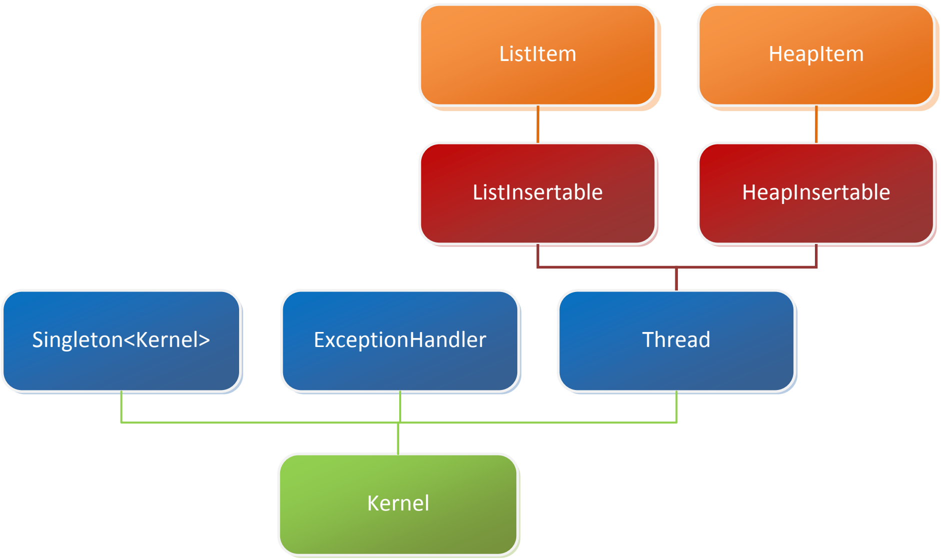

Figure 2 - Kernel class inheritance diagram

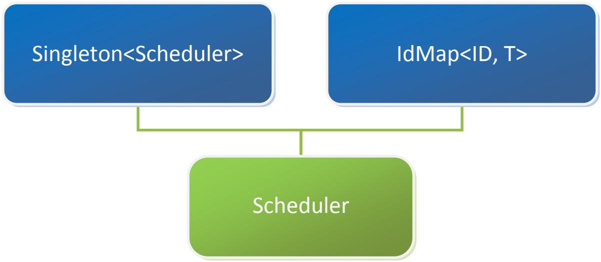

Figure 3 - Scheduler class inherintance diagram

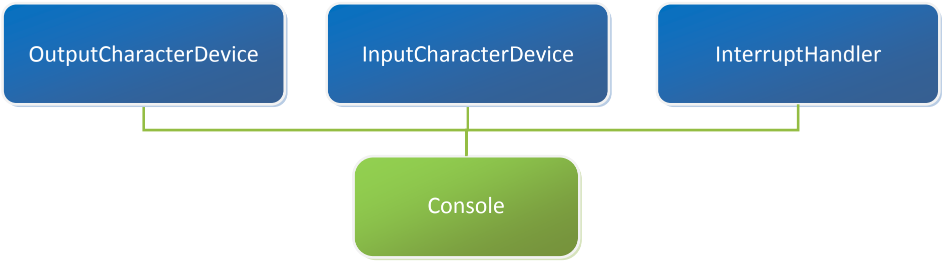

Figure 4 - Console class inheritance diagram

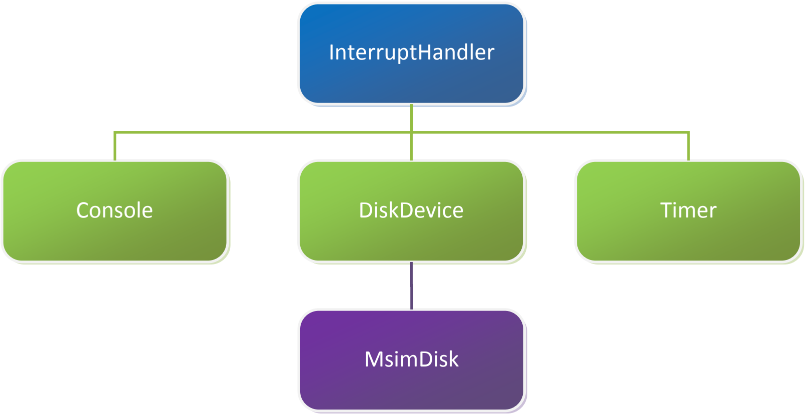

Figure 5 – InterruptHandler class inheritance diagram

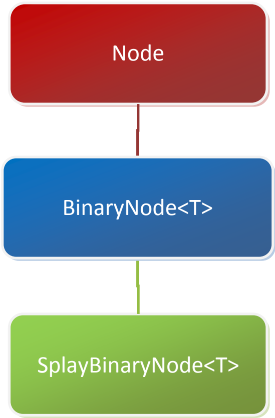

Figure 6 - Node class inheritance diagram

Starting the kernel

Bootstrap code

Kernel initialization begins with the assignment of a static stack. This stack is located just after the end of compiled kernel code. Its memory will never be allocated or deallocated using the Frame Allocator. Moreover, code running on different processors is assigned different stacks to prevent stack overwriting. After the stack has been assigned, the rest of the initialization takes place. Simple lock is utilized to prevent multiple processors entering main initialization routine at the same time. As multi-CPU environment was not part of our assignment, this is where the considering more CPUs end. All processors except the first one are put to sleep and the rest of the kernel runs on single CPU.[4]

Initialization

After all processors but one were put to rest, initialization of the main kernel structures takes place.

Devices and TLB

Just before going to sleep or initializing routine each processor flushes its TLB with invalid entries. First thing to initialize are the devices. Without the console device (the output part of it), it would be difficult to track the booting process and thus these are mapped as the very first thing in the BunnyKernel.

Exception and Interrupt vectors

Each exception is identified by its exception code. As these codes are rather small numbers, they are used as indices to the array of exception handlers. When an exception occurs, its code is identified, used as an index, and appropriate handler is called. This vector of exception handlers is filled during startup.

Like exceptions, the interrupts[5] are identified by their rather small numeric codes. Unlike exceptions, the interrupts do not appear exclusively, and thus it may be the case, that more than one interrupt has occurred. Also unlike exceptions, it is not possible to choose one handler and the whole vector has to be iterated. Interrupts are mostly handled by devices and this vector is filled at startup too.

Measuring CPU frequency

Intel suggests[6] that to measure the frequency of the mobile processors it's a good idea to use firm amount of time and measure the ticks used, rather than vice-versa. As mobile and emulated processors are similar in the way that they lack the firm frequency, we used this approach to determine the ticks per microsecond value of MSIM CPU. This value is later used for conversion between time and CPU ticks.

[4] MSIM emulation generates all hardware interrupts on cpu0, thus making this setup possible

[5] Interrupt being an exception of its own

[6] http://software.intel.com/en-us/articles/how-to-calculate-the-clock-speed-of-a-mobile-processor/

Measuring RAM size

RAM size is measured using a very simple assumption that it is possible to read something from an address in memory what was written there before. Using this assumption, a simple algorithm was designed that repeatedly writes and rereads values from the memory and each time increases the used address.

As directly mapped areas of the virtual address space are too small to be able to access the whole physical memory, this algorithm had to be used on mapped segment of the virtual memory (in our case KUSEG). However this made the task simpler as not the pointer but the mapping of the virtual page can be moved instead. This way it is possible to detect main memory of bigger size that than the size of one segment.

Kernel Structures

After hardware specific values have been measured, the structures dependant can be initialized. First of these would be the Frame Allocator. Frame Allocator is given the address beyond kernel and used static stacks as well as the memory size. The rest of the memory is in its care from now on.

First Thread

After the Frame Allocator has been prepared the very first “real” thread is created. The creation of the thread causes more initializations to occur. Its stack is allocated on the kernel heap causing the initialization of the Memory Allocator and as it is scheduled for execution, the Scheduler is made ready.

Init to Idle

Though the bootstrap code has finished it still uses some resources, like stack, that may come in handy later. At this point initializing routine changes its function and it becomes the idle thread. Next time the execution switches to this thread, it would call wait instructions in an endless cycle. This idea[7] solved two problems - first being the initialization of the idle thread and second is the waste of the resources used.

[7] Inspired by the Linux kernel

Devices and drivers

We provided no modular support for our Kernel and thus all the necessary drivers are present in the Kernel itself. Luckily there are not many devices that needed a driver.

Console

One of the most important devices would be the console. The console driver joined two devices into one, the character input device and the character output device. The former reads input from the user, using interrupt if the key was pressed, and the latter is used for output of characters. Driver for this device is rather simple reading and writing into two separate registers and it contains an Interrupt Handler.

Clock

Clock is a read-only device providing the source of time into the emulated environment. It's represented by two registers, first containing time in the format of UNIX time-stamp, second containing microsecond part of the time information. Only service this driver provides, is the time value.

If however a programmable interrupt was added to this device, it could replace the processor as the source of timer interrupts, thus solving the problem with tick-less scheduling on emulated CPU[8].

Dorder

Dorder is a special device used in multi-CPU environment. Reading from its register provides the number of the processor that read that value.[9] Writing to this register causes an interrupt on every processor the index of which "1" was written to.[10] As the BunnyKernel is not fully multi-CPU capable, only the read functionality is needed to identify cpu0.

Disk

The MSIM Disk is probably the most complicated device that is used by the BunnyKernel. Disk provides access to rather large quantity of stored data. For the sake of simplicity, the BunnyKernel only implements a read-only driver for this device.

Four registers are used to control this device. Disk commands (read/write) take two arguments - first is the number of sector to perform the operation on, and the other is the physical address of the data buffer it should use. When the operation finishes, it is signaled by both the flag in one of the registers and by generating of an interrupt.

[8] See chapter *Threads and scheduling*, part *Tick-less idea* on page 16

[9] It is constant for repetitive calls from the same processor

[10] Writing 1 produces an interrupt on the first processor only, writing 0xffffffff produces interrupts on all processors

BunnyKernel disk driver interface more or less copies the disk capabilities with a bit of an added value. It is possible to read from any location (no just start of the block). This is achieved by using an internal buffer and copying only the requested data. It is also possible to request larger data than one block from the driver. In that case operations are repeated by the driver without the need to call it repeatedly by the process.

Beside the added functionality, one small optimization is also present. The driver tests the location of the offered buffer, and if the disk is able to write to that buffer directly[11], the internal buffer is bypassed and data are written directly to the destination location.

TarFS

The only data structure supported for the disk device is the TarFS. Though being far from the full scale file-system, using the structure of the TAR format makes further expansion possible. Even in the current limited implementation the TarFS provides several advantages. The size of Tar block is the same as the size of MSIM disk block. The other would be easy creation of the disk image file. The currently implemented functions support file reading and directory listing.

[11] Its physical location is known and in one piece (i.e. it is in the KSEG0 segment)

Threads and Scheduling

Threads' representation

In order to start, stop and restart thread's execution we need to store information about this thread. This task is fulfilled by the `Thread` class. Thread needs its stack, sometimes heap to run and a few more things to remember when it is not running in order to know when and whether at all it should be restored to execution. As transitions from one state to another cannot fail, all memory that is required has to be available. This memory as acquired during the thread creation as creation is allowed to fail. In the BunnyKerne,l all that is required to move freely from one structure to another without requiring any additional memory, was included in the `Thread` class. Utilizing inheritance capabilities of the C++ language seemed just right to do the job.

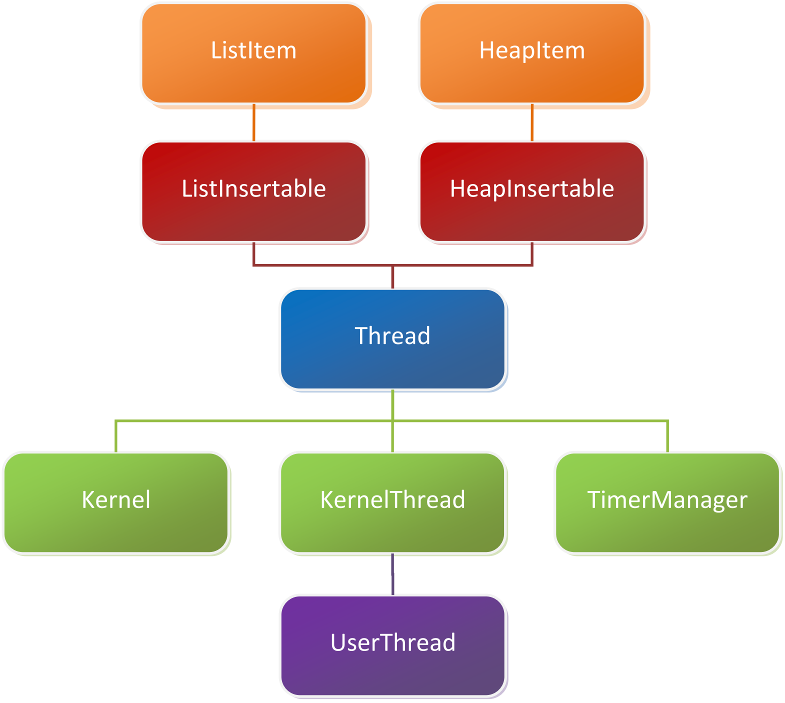

Figure 7 - Thread class inheritance diagram

Other things, that `Thread` stores are its status and the assigned id. Each thread is assigned a unique id that is used for identification. We decided to identify threads by ids as it is simpler to check the existence of an id as opposed to the validity of pointer.

Life of a Thread

Each thread needs to have few values prepared before it can be scheduled for execution. During the time from its creation to the state of full readiness, its state is referred to as *UNINITIALIZED*, during this stage it can have some flaws that would prevent its successful execution, it may lack the stack, or the starting context is not yet prepared on the stack.

Starting and stopping

After successful initialization the thread is enqueued in the Scheduler, being *READY* for execution. When Scheduler chooses next thread to run and switches to its context, it becomes *RUNNING*. Only one thread may be *RUNNING* on one processor. Threads may cycle between *READY* and *RUNNING* states almost indefinitely, switching either voluntarily or preemptively. There are, however, several ways that break this cycle.

Thread that has successfully completed its function becomes *FINISHED.* In this state its return value is stored and Thread stays in the memory until some other thread asks for this value.[12] Finishing is not the only way to permanently end threads execution. The other way, besides pulling the plug, is killing threads. One thread may kill other threads. After killing, the thread becomes *KILLED* - this state is similar to *FINISHED* in terms that the thread will never run again and it stays in the memory until someone asks about it.[13]

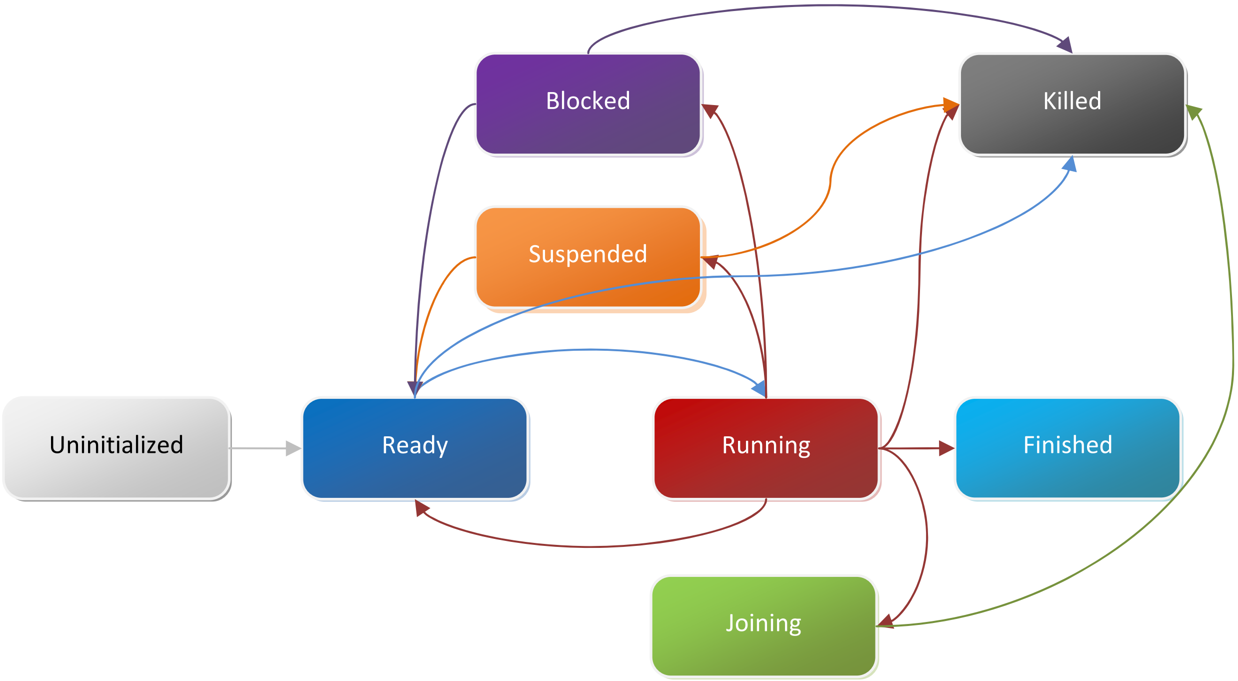

Figure 8 - Thread lifecycle

[12] Unless in detached mode

[13] Unless in detached mode

Pausing threads

Other than finishing threads' execution it is also possible to remove them from the Scheduler and later resume back. This pause is divided into two possible states, depending on how much control the other threads posses that could influence resuming to the Scheduler.

The first state would be *SUSPENDED*. Thread may enter this state by calling `thread_suspend().` *SUSPENDED* threads stay out of the Scheduler until some other thread resumes them by calling `thread_wakeup()`, with respective id.

The indirect paused state is called *BLOCKED*. Threads may be *BLOCKED* for several reasons: there might something preventing their further execution (synchronization primitive), or they might decide to rest for a certain amount of time (calling `sleep()`). *BLOCKED* threads cannot resume execution until the reason of their blocking is gone (time has run out, mutex is unlocked, etc.). They can however still be killed.

Joining

The last state that threads can enter is called *JOINING*. This state is special because although it may pause threads execution and remove it from the Scheduler. The state can be controlled directly, by one thread, or indirectly by a timer (in timed version). The Thread may decide that it needs to know the value the other thread left behind (or will leave when it finishes). This is called joining. Thread that decides to join may not be paused at all, if the other thread is *FINISHED* or *KILLED*, or may be paused for a long period, until the other thread finishes.

Besides the states, thread may enter special mode of execution, this mode is called *DETACHED*. *DETACHED* threads are special in the way they interact with their surroundings. After killing of finishing they do not wait to be joined, and they cannot be joined during their life. Other than that they work as any other thread.

Thread bin

When thread life ends, it is not immediately destroyed as some of its resources, like stack, may be in use. It joins other dead threads in so called `THREAD_BIN` changing its status back to *UNINITIALIZED* and are deleted when next alive thread resumes execution.

It might be a good idea to reuse threads from the `THREAD_BIN`, but as there are more types of Threads it would require separate bins or at least separate buckets in one bin, so we decided to go for the deletion.

The Idle Thread

The idle thread is a very special thread, not only that it was created out of nothing.[14] But its main job is to actually do nothing. Idle thread is run whenever there is no other thread that wishes to run, thus only switching between two states, *RUNNING* and *READY*.

[14] Transforming from the initialization thread

Scheduler

The Scheduler is a special structure that holds all threads that wish to run and decides which thread will be the next to run. This decision is usually made by some sophisticated algorithm based on the information about the nature of the threads purpose. BunnyKernel’s sophistication went as far as the standard FIFO list, as it perfectly fitted the job and was easy to debug and maintain. If there are no threads in the queue, the idle thread is chosen to run.

Timer

The Timer is another special structure but in contrast to the Scheduler it holds threads that are not running for one reason - they decided to wait for a certain amount of time. Timer uses a heap structure[15] to keep track of what thread is the first to resume. It uses processor timer interrupts to be activated at the times of waking up. Other than waking threads, the Timer also keeps track of how long the current thread has been running, and uses the interrupts to plan its switching. All threads, except the Idle Thread, have a finite amount of time they may run. If this time is exceeded, thread is forced to yield.

Callback timer

Callback timer is responsible for scheduling general timed events. Interface allows specifying what should be done, what parameters should be used and when this should happen.

Our callback timer uses classes `ClassTimer` and `TimerManager`. `ClassTimer` is actually a C++ class created on place of the C-structure timer. Structure `timer` is used only as a placeholder for a `ClassTimer` instance. `ClassTimer` instances define timer events.

`TimerManager` class is responsible for handling events defined in `ClassTimer` instances. These events are stored in a heap and they are executed in `TimerManager` thread. If there are no events ready to be executed, `TimerManager` sleeps.

The tick-less idea

A standard way of handling scheduling and thread switching was to have timer interrupt appear every n milliseconds.[16] During this repetitive interrupts, the running thread was checked whether it has exceeded its quantum and another thread was planned. If there were no threads to run, the idle thread was planned repeatedly.

The idea behind the tick-less approach is that these unnecessary interrupts can be avoided. The Timer knows when the next rescheduling or other time dependant action should occur and using this knowledge, we can program the interrupt directly to the time of the next action, skipping all the regular ticks that would occur using the standard way. We can further round up this time to a certain quantum,

[15] See *Appendix 1 – Data Structures*

[16] Mostly because it was slow to reprogram the timer, and repetitive ticks had to be programmed just once at the beginning

creating time slots. Time slots are events in time that may or may not be used depending on whether there was a planned action in the period before the slot. Extended periods of doing nothing can significantly reduce the amount of used time interrupts and the energy used by the processor.

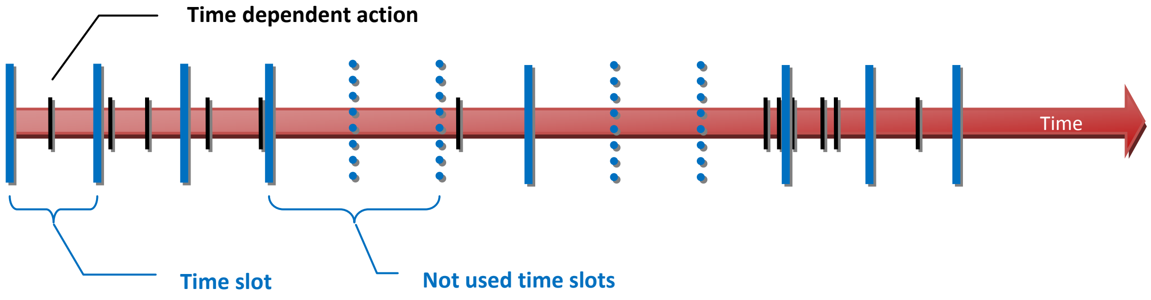

Figure 9 - Time slots

Though this approach works on real processors, it's not the case on emulated hardware. Emulated CPU does “nothing” faster than “something”, thus generating timer interrupts[17] sooner, slowly iterating to the exact time they were planned to.

[17] If processor ticks are used to plan timer interrupts

Synchronization

On a single processor machine it is possible to use interrupt disabling as the only synchronization primitive. It is enough because it assures the atomicity of the operations while the interrupts are turned off. As more processors or more threads come to the play, this simple interrupt disabling can't satisfy the needs. Synchronization is one of the main issues in multithreaded application development. There is a wide range of available synchronization primitives we could implement. We decided for **mutex locks** (as it is compulsory in assignment one), two different implementations of **semaphores**, **spinlock** and a lightweight condition variable which we called **event**.

Interrupt disabling

We used a small class called `InterruptDisabler` which in its constructor turned off the interrupts and remembered the last state and in its destructor restored the last state. We used it as a local variable with prohibited `new` operator. This small class was very useful as in C++ the destructors are called on object disposal while unwinding the stack.

Mutex and Semaphore

The mutex lock is implemented in the `Mutex` class with the use of `InterruptDisabler`. Interrupt disabling is used also in the `Semaphore` class. Both, `Mutex` and `Semaphore` support waiting queues of threads (`ThreadList`) and timed lock (stop waiting after given time). `Mutex` lock can behave as a binary semaphore, but the default is to check the owner, the thread which locked it. One of the differences in implementation is that `Mutex` guarantees fairness and `Semaphore` releases all the waiting threads and allows the scheduler to choose who will get the lock next time. This difference shows up when there are a lot of waiting threads for the lock and the `Mutex` starts to play bigger role in scheduling than the scheduler itself.

Active semaphore and spinlock

The second implementation of semaphore (called `ActiveSemaphore`) and the `Spinlock` don't use interrupt disabling. Instead of that they are programmed in the MIPS Assembler using LL and SC instructions. This doesn't allow us to implement waiting queues of sleeping threads, but it is still ok if the threads are waiting in the scheduler (on multiprocessor machines) or yields every time they can't get the lock.

Event

`Event` is in fact a very simple condition variable. It manages a queue of threads waiting for some condition (or ‘event’). When the event happens, it wakes up all waiting threads.

User space mutex

The user space mutex is functionally similar to the kernel mutex, but it utilizes the Event synchronization primitive to handle the queue of blocked threads. Therefore, in some cases, syscalls must be used – e.g. for blocking the thread on the mutex or for waking up the waiting threads when the mutex is being unlocked. These are making its operations much slower, so the user space mutex additionally remembers number of threads waiting for it to be released, and decrements the number to zero when unlocking the mutex. If no thread is waiting while unlocking, no syscall has to be used.

Memory Management

Physical Memory Management

The physical memory is divided into frames of different sizes (8 KB, 32 KB, 128 KB, 512 KB, 2 MB, 8 MB and 32 MB) managed by the Frame Allocator, which is represented by class `FrameAllocator`. This class provides functions `frameAlloc()` and `frameFree()` propagated to the C API through `frame_alloc()` and `frame_free()` functions as wrappers. Moreover it offers public functions `allocateAtKseg0()`, `allocateAtKuseg()`[18], and `allocateAtAddress()` to avoid using bit flags and to provide some more functionality (see below).

Frame sizes and data structures

Frame sizes correspond with page sizes used by the Virtual memory manager, but the Frame Allocator is actually independent of them as long as it knows the smallest and the largest frame sizes and the difference between two subsequent sizes (which must be the same for any two subsequent sizes).

The Frame Allocator uses a bitmap to store information about free and used frames. For each frame size there should be a bitmap and as the frames have to be aligned to their size (thus a 128 KB frame should start on an address aligned to 128 KB), these bitmaps would create a hierarchical structure similar to a structure used by buddy system allocators.

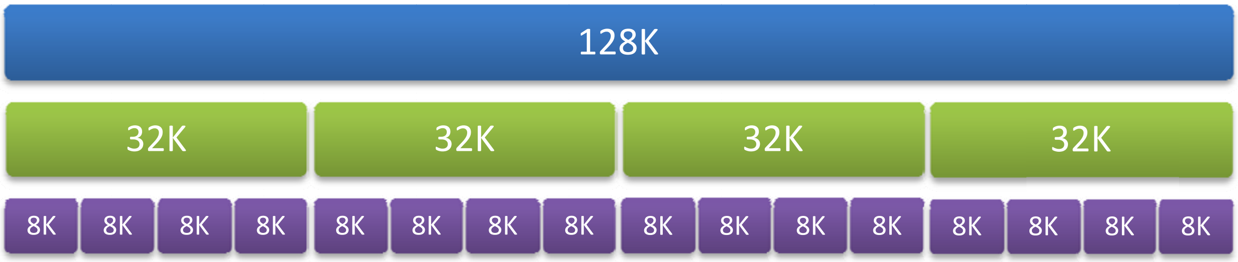

Figure 10 - Part of the frame size hierarchy

Initialization

The Frame Allocator is initialized after the Kernel itself and before anything else. It places its structures in the memory right after the Kernel code and the static stacks. However, it manages the whole memory, including the part used by Kernel and Frame Allocator structures (which is marked as used while also keeping the address of the end of this memory). As it is necessary that all frames are aligned (see above),

[18] In the FrameAllocator class, we use term ‘KSEG’ (or KSEG0 or KSEG1) to shortly denote the physical address range 0 – 512 MB and term ‘KUSEG’ to denote the addresses starting from 512 MB.

it will be too difficult to manage the memory starting on an address not aligned to one of the frame sizes.

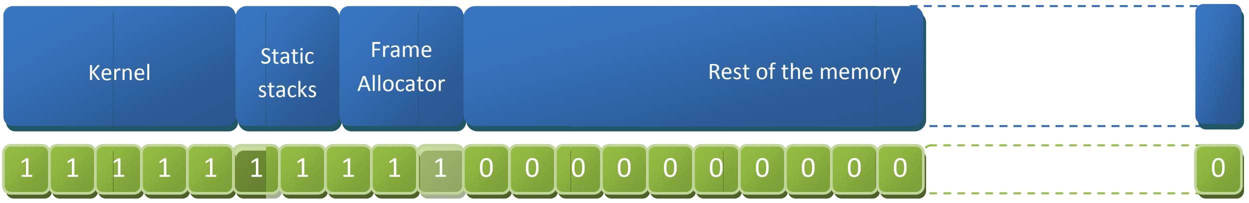

Figure 11 - State of the memory right after initialization

Allocation and deallocation

When a frame of a particular size is being marked as used or free, the frames of other size included in this frame, or containing it should be marked properly as well, thus achieving an invariant that *a frame is marked as free if and only if all of the smaller frames contained in it are free and it is marked as used otherwise*. The process of marking all affected frames (not only frames of the required size) slows down the allocation and freeing of physical memory, but this restraint is compensated by the speedup achieved by the lookup algorithm.

When the Frame Allocator receives a request to allocate some frames, it calculates how many larger frames would satisfy the request and starts to search for as large frames as possible. (It is 'possible' when there are enough free frames of that size – not necessarily consequent). E.g. if a process wants to allocate 10 frames of size 32 KB, then 3 frames of 128 KB or 1 frame of 512 KB will satisfy his request as well (for more information, see function `FrameAllocator::DefaultFrame()`). So far this is similar to the mentioned buddy allocation system, but unlike it, our Frame Allocator uses this only to improve the lookup. When a sufficient number of free consecutive frames is found the Allocator will allocate (and mark as used) only the requested amount of memory. The negative of this algorithm is the possible external fragmentation of the memory.

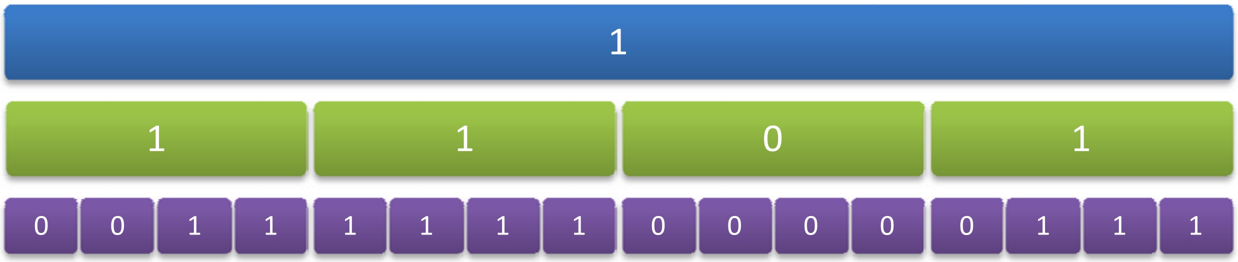

Figure 12 - Example of a state of the bitmaps

Extra features

In our implementation we also decided to use a single bitmap instead of multiple (one for each frame size). This is achieved by virtually putting all the bitmaps one after another, starting with bitmap for the largest frame size and ending with the smallest frames. This approach has several odds:

- The initialization of the bitmap is easier as we don't have to determine number of bitmaps
- Together with the optimized bitmap operations[19] it speeds up lookups even more, as large blocks of used frames are skipped even across different levels of frames.

To boost the performance of the Frame Allocator even more, there are few more improvements, e.g.:

- It remembers the address and size of recently freed block frames and tries allocation at this address before searching.
- When not enough free frames are found, the `allocateAtKseg0()` and `allocateAtKuseg()` functions return the number of frames in the largest free block found and the address where this block starts. When allocating, the functions will first try to allocate at given address which may be the one previously returned.

Virtual Memory Management

In operating systems where multiple processes are being executed at the same time, it is necessary to use virtual memory to keep them separated. The added value of this mechanism is that one whole virtual memory block doesn't have to be continuous in the physical memory. This is achieved by splitting the virtual memory area into one or more subareas whose size is aligned to the physical page size. KSEG0/1 segments must be however treated differently, as they are not being mapped by the TLB. Hence the whole area in KSEG0/1 must be continuous in the physical memory.

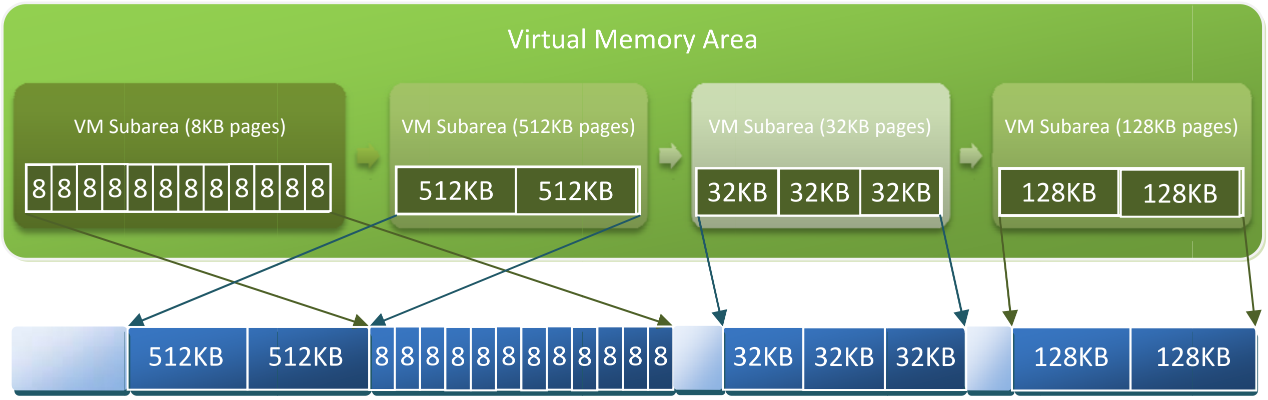

Figure 13 - Virtual Memory mapping to the physical memory

[19] See *Appendix 1 – Data Structures*

TLB

Transaction Lookaside Buffer is a device that processor uses when accessing virtual memory that resides in the mapped segment of the Virtual address space, such as USEG or KSEG3, SSEG, etc.. TLB does not translate every address separately but in groups. Groups of virtual addresses are called pages, groups of physical addresses are called frames. Page translates to frame mapping all the addresses from the group directly.

Translations have to be filled in to the TLB and this filling is under control of the operating system. Using this control OS decides which parts of the memory are accessible to the running process.

It is obvious that the bigger the pages the less TLB entries are needed to cover certain memory area, on the other hand some memory may be wasted. Ideally OS would choose the right page size and map big areas using big pages and small areas using small pages.

Multiple page sizes

In order to achieve the best possible performance we decided, despite the discouragements, to support variable page sizes simultaneously. By dividing the virtual memory areas into subareas of the largest possible page size, we are able to eliminate unnecessary page faults which lead to fewer TLB refills. However, we soon found out the problems this decision has brought. MIPS TLB emulated by MSIM, uses 48 double entries. That means that two translations are stored in one TLB entry. This duality causes a lot of problems when supporting multiple page sizes. Borders of the pages of different sizes and smaller pages being included in unmapped bigger page are just some of them.

Solution to this problem proved to be quite simple, *doubling the size of the pages*. This solution not only removed additional exceptions (the only TLB related exception is now the TLB refill), but also greatly simplified the role of the TLB managing class. There is no need to decide whether left or right entry was accessed as there are either both or none. While it made the TLB managing class simpler it put more stress on the Virtual Memory Management classes.

ASID Management

The MIPS processor provides the means to distinguish entries in the TLB by different ASID identifiers. As ASID is 1 byte number there is a limited number of available ASIDs. Assigning of ASIDs to a virtual memory map thus has to be dynamic. If there is no free ASID available it has to be taken from an existing virtual memory map, its entries flushed from the TLB and only then it can be reused.

Data structures

The virtual memory areas are being stored in a splay-tree[20] instead of page-table or balanced tree. We chose this because the splay-tree offers an interesting property of keeping recently allocated node near the root element.[21] The order of allocated subareas is fixed, so we decided to keep it simple and used a

[20] See *Appendix 1 – Data Structures*

[21] For more information, see *Appendix 1 – Data Structures*

simple list to store them. With a small memory overhead it would also be possible to use a tree, but we haven't tried that.

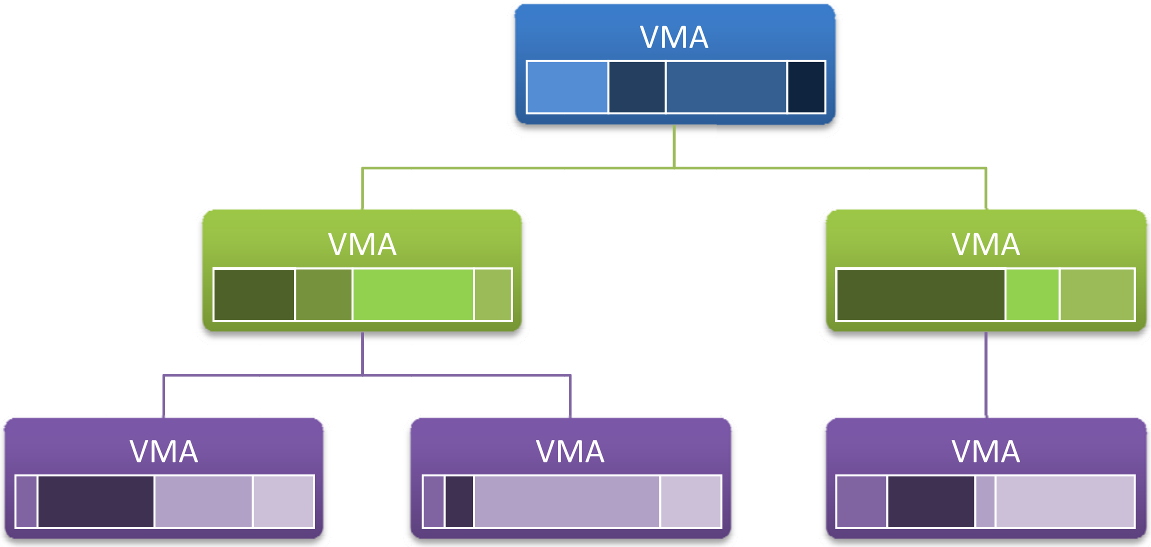

Figure 14 - Virtual Memory Areas (with their subareas) in a tree

Choosing the right page size

When the memory area is to be mapped, the right size of pages needs to be chosen. In the BunnyKernel we designed a simple algorithm that would suggest the right page size:

```
foreach (sizes as size)
{
     aligned <- ceil(request / size) * size
     loss <- (aligned - request) / aligned
     count <- (aligned / size)
     result <- (loss * coef * 100) + (1 - coef) * count;

     if (result < best_result)
     {
          best_result <- result
          winner <- size
     }
}
```

Changing the `coef` parameter changes the priority from the number of TLB entries to the percentage of memory wasted by assigning bigger areas than requested.

The alignment of the virtual memory area starting address must be considered too, and therefore the resulting page size is the minimum of the suggested and largest possible of the address alignment. The page frames to store the area are being allocated in the selected size until there are no more available.

After that the subarea is closed and a new one is created with smaller page sizes.

This method works well in typical use cases, but in some cases it degrades. For example if you allocate a 1032 KB block (1MB + 8KB) the existing solution would choose 8K pages for the whole area. However it would be possible to implement a more sophisticated algorithm which would suggest different page sizes for each part of the virtual memory area (they will be the subareas). In that case it would try to allocate a 1MB subarea with two 512KB pages and then the remaining 8KB.

The operations supported by the virtual memory management try to preserve the page size usage while being reasonably fast. However it is not possible in some cases, such as `vma_split()` having to split the virtual memory area in the middle of a subarea.

Other features

The `vma_merge()` function is very fast, as it just concatenates two lists of subareas. The `vma_remap()` function only has to check if the destination is available and then it adjusts the starting address of the virtual memory area.

Memory Allocator

Object design

Our kernel has a common parent class for both kernel- and user space memory allocators, named `BasicMemoryAllocator`. This class is responsible only for managing the assigned memory. It does not handle how to get physical memory to manage and does not handle synchronization. For this purpose it provides pure virtual methods `getNewChunk()` and `returnChunk()`, which are responsible for communication with lower memory management. The synchronization is implemented in a child class in the overridden `getMemory()` and `freeMemory()` methods.

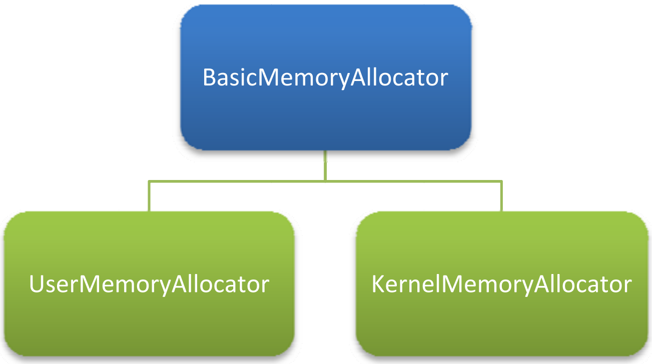

Figure 15 - Memory allocator classes' hierarchy

In user space, the heap allocation is synchronized by a yielding spin lock, in kernel space by disabling interrupts. Considering that the kernel does not allocate as much and as often as a normal program, allocation in the kernel should not last long and therefore disabling interrupts for the allocation is a suitable solution. It even does not involve any thread management as in case of mutexes or active waiting as in case of spinlocks.

Memory structure

Memory structure is similar to the glibc allocator in Linux: memory blocks are stored in lists and each memory block has a header and footer with block information.

Most important structures in the allocator, except the allocator itself, are `BasicMemoryAllocator::BlockFooter` and `BasicMemoryAllocator::BlockHeader`. Both of them store information, whether they are used, free or on the border of a memory chunk, represented by a magic value, which may be used for header or footer correctness and block size (including header and footer). The header also stores a pointer to the next and the previous header in the list.

In the memory structure an invariant that right after each footer there is some header and right before each header there is some footer is achieved. If a block is the first in its memory chunk, then right before its header there is a footer with state 'border'. Similarly for last block in memory chunk, after its footer there is a header with state 'border'. This means that on beginning of each chunk there is a `BasicMemoryAllocator::BlockFooter` and on the end, there is a `BasicMemoryAllocator::BlockHeader`. Border headers and footers store the size of the memory chunk (including borders).

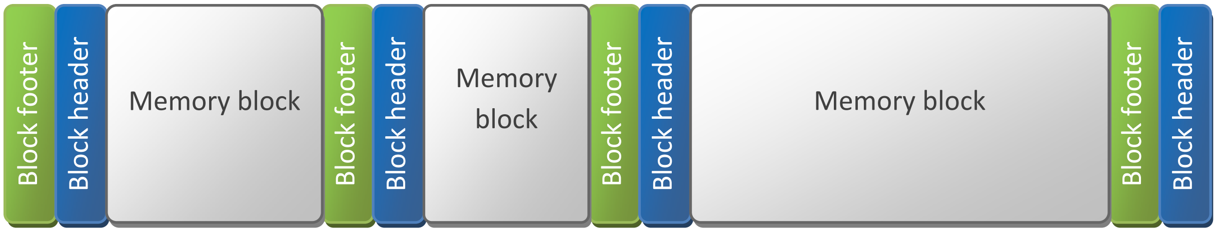

Figure 16 - Memory chunk structure

All headers are stored depending on their state in the list of free or used blocks or in the list of memory chunks. There is only one list of free blocks. On every `getMemory()` call, the free blocks list is searched for suitable block (according to the allocation strategy). If no suitable block is found, a new memory chunk is demanded. A big free block is created in the new chunk.

Allocation strategies

Heap allocator implements five allocation strategies: **first fit** and **next fit**, which sort free blocks according to their address, **best fit** and **worst fit**, which sort free blocks according to their size and a strategy which is called **'default'**, which does not sort free blocks at all.

All these strategies are easy to switch, because they use the same structures (lists of blocks) and their most significant difference is in the way, how they sort free blocks in list. This enables to switch them during runtime, through method `setStrategyXXXX()`, where XXXX denotes name of some strategy, or trough `mallocSetStrategyXXXX()` function in the librt API. This function sets pointers to functions related to the new allocation strategy (these are `getFreeBlockFunction()`, `insertIntoFreeListFunction()` and `setSizeFunction()`) and if needed, reorders list of free blocks.

To implement a new strategy, `setStrategyXXXX()` function must be implemented, `getFreeBlockXXXX()`, `insertIntoFreeListXXXX()` and `setSizeXXXX()` functions can be implemented or strategy can use these from other already implemented strategies (for example `setSizeFunction()` is the same for default, first fit and next fit strategies). These strategies can only be based on sorting free blocks in list and searching through this list. For a more complex allocation strategy, using for example more free blocks lists, a more complex change in the allocator code would be needed.

Additional features and optimization

What do we want from the allocator

Already mentioned features can be interesting from the system programmer's point of view, but normal programmer of applications is more concerned in the overall performance of his application, which includes memory allocation as well.

Cooperation with lower memory management

First of all, physical (or virtual) memory, which is managed by the allocator, should be requested in advance and in big chunks, so that it is not needed to request new memory often. This should be a matter of course.

Another optimization is based on the fact, that BunnyKernel uses variable page sizes. This can much improve performance, because there can be much less TLB refills. But to use only (or mostly) as big pages as possible, it is necessary, that the heap allocator, for instance, requests only memory chunks aligned to some 'big' page size, usually a power of two, in case of BunnyKernel it is 512 KB.

The interface of the syscall responsible for getting the required virtual memory allows the kernel to choose a better chunk size (e.g. bigger) and to return this optimized size together with address to the new memory chunk. This allows optimization of virtual memory requests for size on the kernel side (that means also considering hardware configuration) and also that the allocator is aware of the fact, that he received more memory than he requested. Although this feature is now used mostly for correct alignment of memory requests, it can be easily extended to optimize memory allocation. This interface also allows allocator to use all the memory he got from the virtual memory allocator, because he exactly knows how much he got.

Chunk resizing

BunnyKernel user allocator requests memory mostly by 512 KB and sometimes by its multiples. Therefore most of the memory would be in 512 KB pieces, what disables the use of bigger pages. On hardware with big amount of memory this might cause problems with often TLB refills, and that is exactly what we do not want to happen.

Another problem with having many smaller memory chunks is that memory is scattered into relatively small pieces, most of them having small unusable free block on the end, increasing external fragmentation.

To solve these problems, our allocator enables to increase size of memory chunk as well as to decrease it, using support for virtual memory area resizing. This is done by 512 KB pieces.

Extra features

In user API there are simple functions for getting total size of the memory managed by heap memory allocator (`mallocatorGetTotalSize()`) and amount of the memory stored in all free blocks (`mallocatorGetTotalSize()`). For more extended features of the allocator it is needed to use C++ code and to include `BasicMemoryAllocator.h`, which enables to get the list of free or used blocks and also memory chunk list (currently used memory allocator instance can be got by `UserMemoryAllocator::instance()`). These features are mostly useful for debugging or benchmarking purposes.

User space, processes and syscalls

User space thread

The structure representing the user space thread is very similar to the structure that represents threads running only in the kernel space. User space thread adds user space stack, and switches to user mode just before starting the execution, this switch is done in the branch delay slot created by the jump instruction. The stack allocated in kernel is still used. It is switched to when handling exceptions. The fact that handling an exception can cause a TLB refill forced us to develop more complex stack switching algorithm than just a simple switch.

Stack switching

When a user space thread is entering kernel, stack is switched to the kernel one, but the currently used stack pointer is saved not in an external location but on the new kernel stack. When returning from the exception handling the former stack is restored. This method enables us to transparently switch from kernel stack to the same kernel stack and back, without knowing the exact location of the previous stack.

Processes

Process structure holds together all resources used by the threads of one process. These would include the virtual address space, used synchronization primitives and opened files. Resources are continuously added during the life of the process and have to be correctly returned on its life's end. Process ends when one of its Threads calls exit function or the first thread returns from the `main()` function.

Syscalls and librt

Syscalls are special exceptions that are raised by the program when it needs to communicate with the kernel. Syscalls in BunnyKernel include input, output, thread manipulation, synchronization, memory management and process manipulation.

Input / Output

Only `puts()` and `gets()` functions are implemented as syscalls inputting and outputting of one character is implemented using these syscalls. Output of `printf()` is buffered and uses the `puts()` syscall.

The other part of I/O handling is file control functions. File-system entities are controlled via ids called descriptors. All functions acquiring releasing or manipulating files and directories are syscalls.[22]

Thread management

Almost all thread management functions had to be implemented as syscalls, notable exception would be

[22] For more information see chapter *Device Drivers*, part *TarFS*, on page 12

thread_get_current() that reads the thread id from the specific location and does not need to call kernel. This information is updated during rescheduling.

Process management

Again all function with the exception of process_self() are syscalls. Current process ID is read from the specific location. This information does not change during the lifetime of the process and thus does not have to be updated.

Memory management

Heap memory allocator does not require any syscalls for its standard operation. Syscalls are used to increase or decrease the amount of memory this allocator distributes.[23]

Synchronization

Fastest way mutex locking and unlocking does not require syscalls and can be made entirely in user space. If thread needs to be blocked it uses Event synchronization primitive controlled by syscalls.[24]

Startup

A simple user space program called init is provided and runs if it's found on the hard disk. It is a simple shell-like application showing all files present on the device and enabling user to run the programs.

[23] See UserMemoryAllocator documentation for details

[24] See Event documentation for details

Future plans

Future development would certainly include adding write capability to the disk subsystem as well as utilizing the directory support. Other improvements would include reusing of the freed user stacks and joining the used stacks into one area to make use of larger page sizes.

The `Bitset::full()` function should be optimized too as it will greatly speed up the allocation of physical memory.

Improving the Scheduler and testing more scheduling strategies would be of interest too. However the most challenging improvement would be the implementation of multi-CPU support.

Future plans for the Memory Allocator

Exact fit with more allocators and size tolerance

If we wanted to create exact fit allocator, it would be needed to have more lists with free blocks, each for exact size of block and one for blocks, which would not fit to other lists, similar as it is in Linux glibc allocator. For this purpose it would be required to design another allocator class, which would not join free blocks and to create new class which would contain more of these allocator classes. This would also require to have shared list of memory chunks (this is not possible right now) and that `UserMemoryAllocator` and `KernelMemoryAllocator` will be children of a new parent class.

Another improvement for this exact fit implementation would be a size tolerance for requested amount of memory. That would increase internal fragmentation, but on the other hand it would decrease count of needed allocators for different sizes. This tolerance could be adjusted even on runtime.

Asking for smaller memory pieces if necessary

If there is not enough unused physical memory, the heap allocator will not try to request smaller amount. This might cause, that there will be up to 512kB unused physical memory. Or if physical memory is heavily fragmented (because of use of another allocator in another process which requests different amounts) there might be much more unusable memory.

On the other hand, such approach might cause, in case of resizing some memory chunk, that this chunk will be mapped trough smaller pages. If we can expect low physical memory fragmentation (what we can), current solution is good enough.

Resizing of memory chunks in kernel space

The `BasicMemoryAllocator` has sufficient logic to resize memory chunks both in the user space and kernel space. In the user space this is done with the help of `vma_resize()` function, in the kernel space this can be achieved by requesting memory right after some existing memory chunk.

This would lower external fragmentation of memory in kernel space and optimize a bit the memory requests of heap allocator. Unfortunately this functionality has not been implemented yet.

Readiness for SMP

Even though this was not required, it is worth to mention, that the current allocator would have poor performance on multiprocessor machine, if it would be able to work at all. Locking of allocator structures is global and therefore blocks all allocation operations on all processors. To avoid such problems it would be needed to have local a free block pool for each processor and to have an ability to send free blocks between global and local free blocks pools.

Appendix 1 – Data structures

Generic tree

The `Tree` class used by the BunnyKernel is a generic container class. Its properties depend on the type of the node that is used to instantiate the tree. There are two types of nodes present - `BinaryNode` and `SplayBinaryNode`. The former forms `BinaryTree` the latter `SplayTree`.

Binary tree

`BinaryTree` is a standard binary tree with no balancing or optimization whatsoever. It is used mainly for comparison if ever.

Splay tree

`SplayTree` is a special kind of binary, self-modifying balanced tree. In addition to keeping more or less balanced it keeps recently accessed elements near the root. Taking locality of reference into consideration, it appeared to be interesting structure to use in kernel memory related classes.

Insertions are done in the exactly same way as in the Binary tree. Deletions are done in the same way as in the Binary tree. If the values of the nodes cannot be switched the Node is rotated down until it rids of one of its children and then removed.

Search is the operation that makes the difference between Splay and Binary trees. In Splay tree each successfully found Node is “splayed” to the root to speed up future searches for the same item.

Heap

Another data structure used in the BunnyKernel is a d-ary min-heap represented by class `Heap`. The number of children (the arity) and the type of data stored in the heap are given by template parameters. This class however does not support inserting of pure data, but this have to be either encapsulated in an object of type `HeapItem`, or should be an instance of a class derived from `HeapInsertable`. In both cases the heap operations are done without allocating or deallocating any memory. This feature is greatly utilized in the `Timer` and `TimerManager` classes.

Bitset

`Bitset` is an optimized container for storing bit values, true and false. When you need to store a huge amount of bits, the bool type in C++, which is aligned to one byte, uses much more memory than it is necessary. For the internal representation we use processor native type, so array of 4byte unsigned integers. Thus place we need is as much as possible and not more. The search in the container is optimized for speed. If we are searching for the next true value, we use a function to get the number of false (empty bits) and add the two numbers (start and the count). The optimization is to not check each bit in the integer value, but check the whole integer to zero, so we can get the result (32 bits are set false) faster. This fast lookup is not implemented yet for searching the next false value, but it would be much the same as searching the next true value.

Hash map

Simple container, which enables to store items into a hash map with linked lists. Items in `HashMap` are not iterable. It is possible to set the size of the hash table (and therefore the hash range). Type used as a key value must have declared and implemented function `unsigned int hash ( keyType key, int rng )`.

List

`List` is standart templated list container with iterator, items in list are instances of `ListItem` templated class. List enables to insert values as well as already created `ListItems`.

`ListInsertable`, similar to `HeapInsertable`, is convenient class. All children of `ListInsertable` class can be inserted into and removed from List. This makes work with the `List` easier, and allows adding functionality into items stored in the `List`. This can be useful for objects which sometimes need to be inserted into the list in a function which cannot fail because of insufficient memory, good example of this is class `Thread`, which sometimes needs to be added to or removed from the thread list.

Appendix 2 – Benchmarks

Memory Allocator performance and benchmarks

We used several tests to measure the difference between allocation strategies. First was the `malloc1` test provided to test the allocator functionality. This test used mostly same-sized blocks and therefore did not simulate real program behavior, but the results are worth to mention.

Second benchmark used (`mallocBenchmark01`) is a single-threaded test which allocates random-sized blocks (with specified maximum size). Program allocates 100 random sized-blocks and after this deallocates part of them (specified as constant to 75%). This is repeated until the specified amount of memory is allocated.

Third benchmark program (`mallocBenchmark02`) is a multithreaded version of previous one.

Measurement methodic

For each run of benchmark program was measured how long does it take to finish it`s task, for the benchmark programs was also measured the count of free blocks in memory (which is related to memory fragmentation), the count of 'small' and therefore unusable free blocks, count of memory chunks (using chunk resize feature caused, that in all cases there was only one memory chunk), total size of free blocks and estimated overhead (this means memory in headers and footer of memory blocks and memory unused because of internal fragmentation).

Each measurement was performed twice, to avoid statistical errors. If measured results were too far from each other, another one was made.

Results

There was made really much work measuring the data from benchmarks and from these data we can make many conclusions and performance analysis, but that would extend this material beyond any acceptable limit - this should not be all about the memory benchmarks. We mention here only some conclusions about basic dependencies between key values and about allocation strategies differences. We also do not provide here all measured data, for more please see */measurements* directory in the repository.

Speed of strategies

In benchmarks, the **fastest** strategy was **first fit**, both in single and multi threaded benchmark, as is also shown on the graph.

Time of running benchmark is in quadratic relation with the amount of finally allocated memory. From measurements we concluded, that quadratic relation is also between the count of threads and the time of benchmark.

We should also mention that in `malloc1` test was best performance achieved by **'default'** strategy. This might mean that this strategy is well suited for allocation blocks of same size.

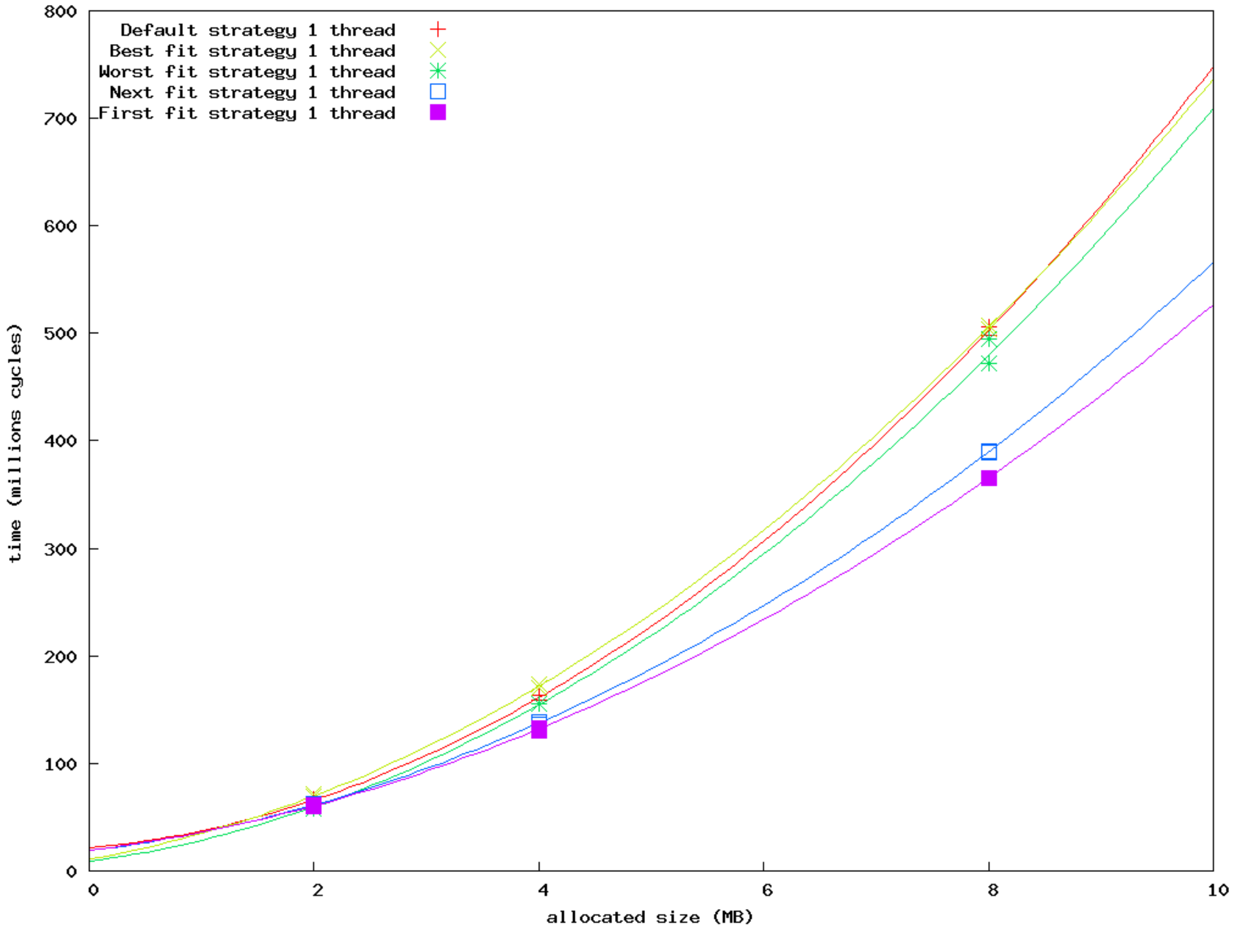

Figure 17 - Benchmark time relation with allocated memory

Fragmentation and overhead

Here is clear, that worst fit strategy generates most free blocks, which are often not used and memory is wasted. **Best fit** strategy maintains **least free blocks** and **wastes least memory**. On the other hand we should mention, that most free blocks left by best fit strategy were smaller than 160B (and therefore could not be used) and **worst fit** strategy had **least blocks smaller than 160B**.

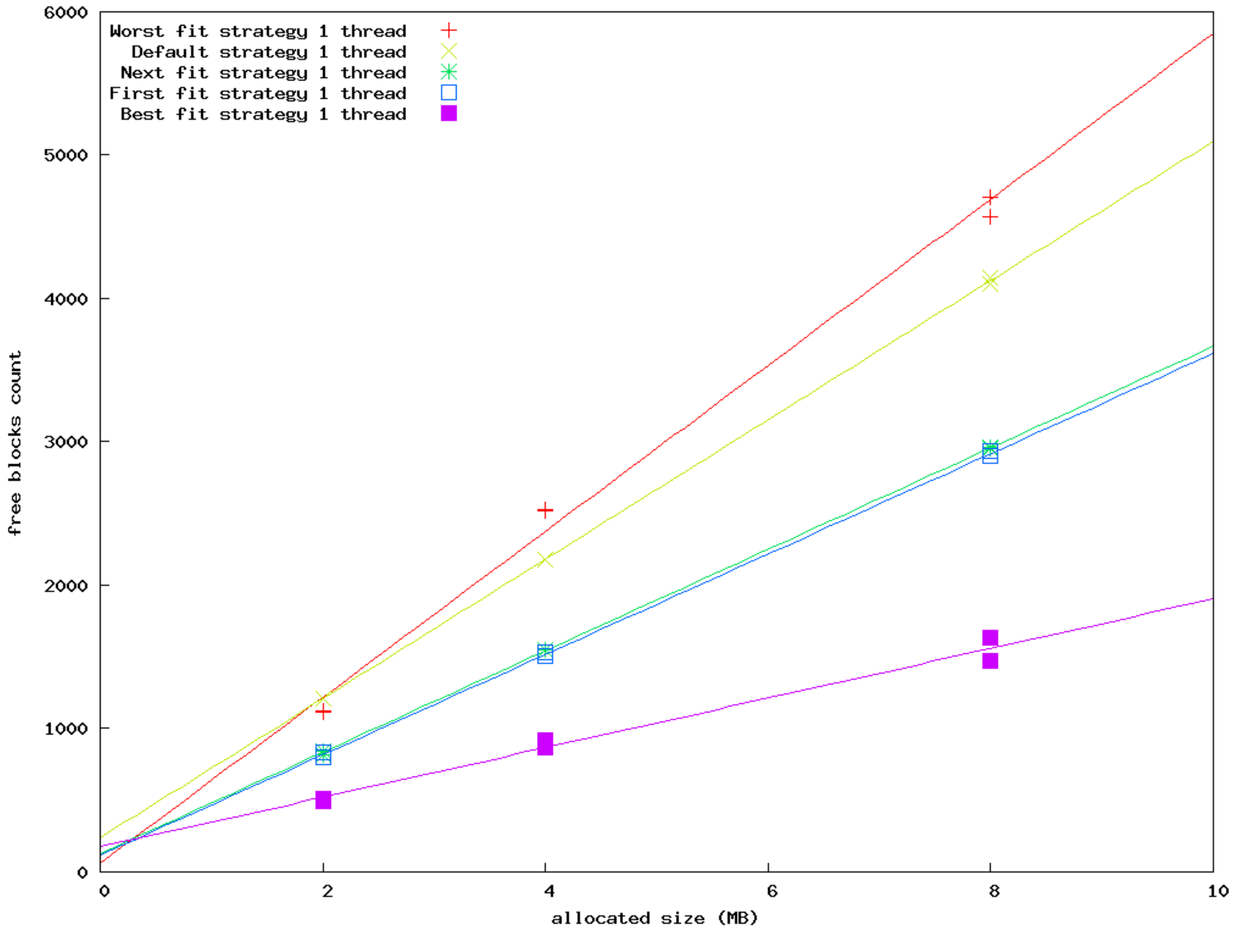

Figure 18 - Count of free blocks vs. allocated memory

Our measured overhead is mostly related to count of blocks. Each block has header and footer which cannot be accessed by program, therefore for each block there is 24B of unusable space. Internal fragmentation did not affect this too much, except the case of the best fit strategy. We can conclude that mostly related to the count of memory blocks.

From results we can also conclude, that **next fit** and **best fit** strategies has **almost the same memory overhead and fragmentation**. **Most unusable memory** has **default** strategy. It is worth to mention, that even though worst fit strategy wastes memory most, it has least memory in unusable blocks. Interesting fact is, that **best fit** does have **least memory in block headers and footers and small-sized blocks**.

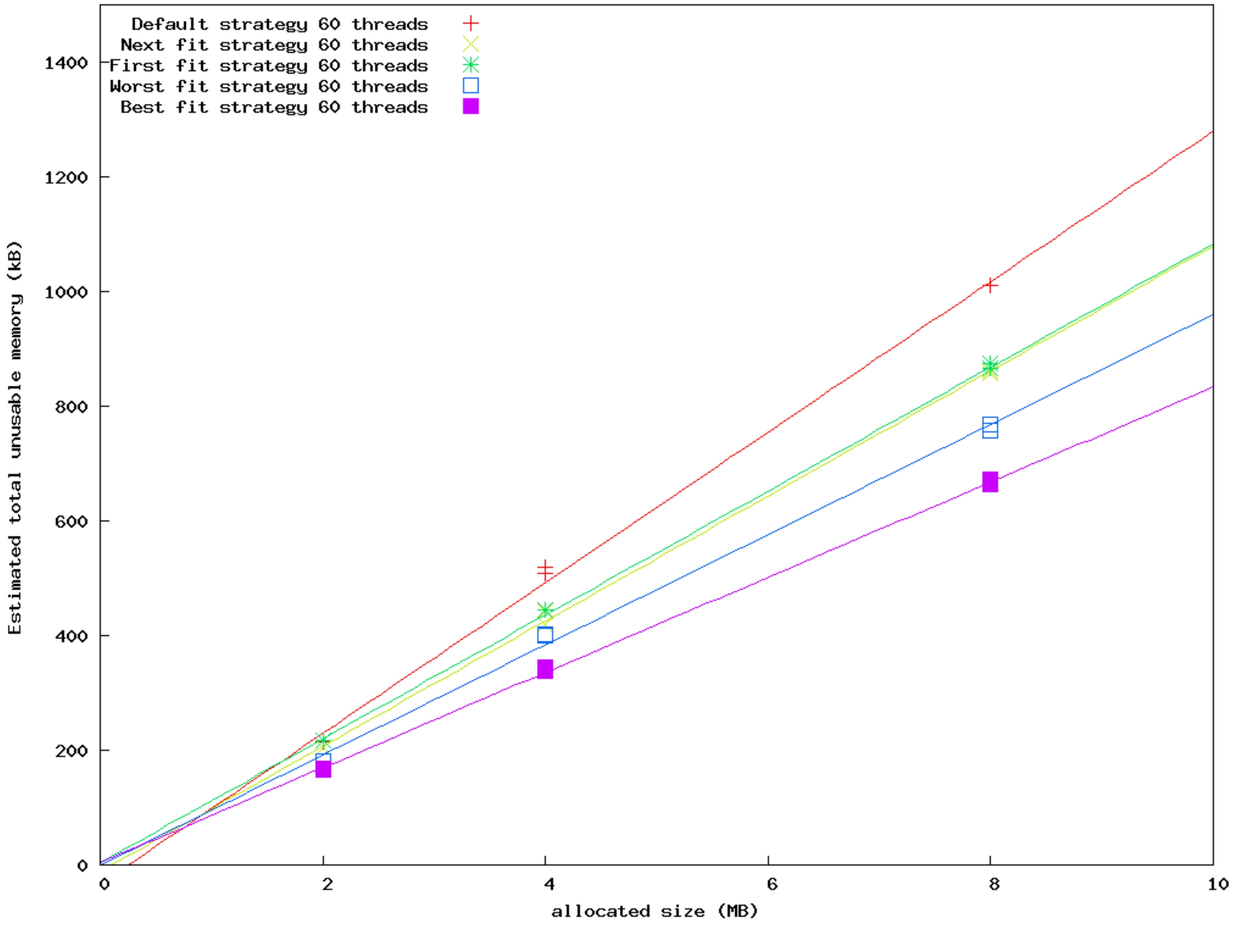

Figure 19 - Size of block headers + internal fragmentation + unusable blocks vs. allocated memory, multithreaded

Other conclusions

It is interesting, that with increasing amount of threads in benchmark, most of strategies increased count of free blocks and therefore fragmentation, but with best fit strategy fragmentation actually decreased. This might mean that best fit strategy is able to minimize memory fragmentation, if run with random allocations and deallocations.

Using variable page sizes increased performance in the `malloc1` test about six times. Using resizable memory chunks lowered the time of the test by another 1/3.

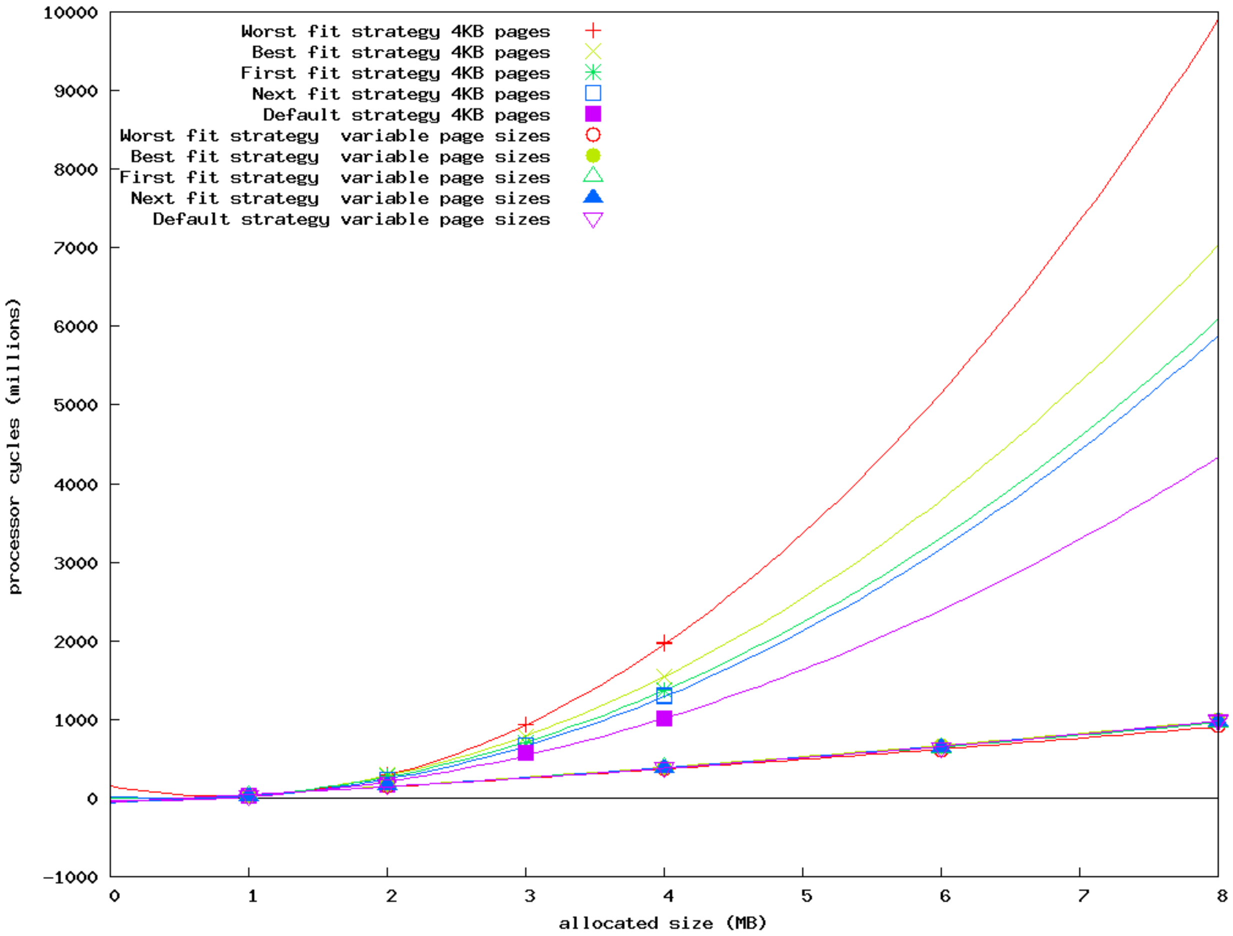

Figure 20 - Maloc test times (4 KB page vs. variable page sizes)

From the benchmarks we could also conclude, that yielding spinlock is suitable for synchronizing small amount of threads, whereas mutex is more suitable for the synchronization of 50 or more threads.

We should be aware that >50 thread testing has problem with often TLB refills, because each thread switch requires access to thread`s stack. Also results of these tests have significant statistical variance.