Asst2: Segmented Paging

Usage

Files needed to use this library:

- `my_pthread_t.h`
- `mylib.h`
- `my_pthread.c`
- `mylib.c`

Use `gcc` to compile the following source files:

- `my_pthread.c`
- `mylib.c`

API

This is the API for the new functions added in Asst2 available to threads created with this library.

Synopsis

```
#include "my_pthread_t.h"

void * threadAllocate(size_t size);
void * shalloc(size_t size);
void threadDeallocate(void * ptr);

#define malloc(size) threadAllocate(size)
#define free(ptr) threadDeallocate(ptr)
```

Description

The `threadAllocate()` function allocates `size` bytes and returns a pointer to the allocated memory. This memory can only be accessed by the calling thread. The memory is not initialized. If `size` is `0`, then `threadAllocate()` returns `NULL`.

The `shalloc()` function is the same as `threadAllocate()` except that the allocated memory is accessible to all threads.

The `threadDeallocate()` function frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `threadAllocate()` or `shalloc()`. Otherwise, or if `threadDeallocate(ptr)` has already been called before, undefined behavior occurs. If ptr is `NULL`, no operation is performed.

Return Value

The `threadAllocate()` and `shalloc()` functions return a pointer to the allocated memory that is suitably aligned for any kind of variable. The difference between the two is that the allocated memory from `threadAllocate()` can only be accessed by the calling thread while the allocated memory from `shalloc()` can be accessed by any thread. On error, these functions return `NULL`. An error occurs if there is not enough memory to allocate. `NULL` is also returned by a successful call to `threadAllocate()` or `shalloc()` with a `size` of zero.

The `threadDeallocate()` function returns no value.

Prelude

How Main Memory is Divided

The "main memory", "RAM", or "physical memory" is a contiguous allocation of `MEM_SIZE` bytes referenced by `memory`. The first `sizeof(struct memoryMetadata)` bytes is metadata. The metadata gives information on where the thread library "partition", page table, memory pages, and shared memory "partition" is located. It tells the how many memory pages and swap file pages there are. The metadata also stores the file descriptor for the swap file.

Definitions

A "partition" is a chunk of memory that consists of at least one "block" of memory. If there is more than one "block" in a partition, the "blocks" must be contiguous.

A "block" is a chunk of memory that consists of a "head", a "payload", and a "tail". the "Head" and "tail" are both identical metadata where the first `sizeof(int)` bytes tell if the "block" is used, and the rest of the metadata tells the size of the payload. As the name implies, the "head" resides in the first `sizeof(struct blockMetadata)` bytes of the "block" and "tail" resides in the last `sizeof(struct blockMetadata)` bytes. The "payload" is placed between the "head" and "tail" metadata.

Main Memory Divisions

The thread library "partition" is used to allocate memory on library calls to `myallocate`.

The page table holds information for all the tables in the in memory and swap file. Each row of the table has the thread's info, page number of the thread, location in memory, and swap file location. If the thread information is `0` the page is free, if the location in memory is `0` then the page is in the swap file.

The memory pages are the pages that are located in memory. This region is aligned with the system page. Used to allocate memory that can only be accessed by the allocator.

The shared memory "partition" is used to allocate memory that can be shared between thread.

Implementation

Initialization

The memory manager is initialized on the first call to either `myallocate()` or `shalloc()`. Initialization starts off storing the system page size and allocating `MEM_SIZE` page-aligned bytes to `memory`. After that, different numbers are calculated for later use when assigning sizes to the thread library "partition" and threads' memory aswell as finding the number of pages in memory and swap file. The calculations divide memory for library and threads based on `LIBRARY_MEMORY_WEIGHT` and `THREADS_MEMORY_WEIGHT` ensuring the memory pages for the threads are aligned with the system pages. Now `memory`'s metadata is initialized using the calculations to create "partitions" for the library and shared memory, storing the address to threads' memory and the number of pages, plus creating the swap file. A signal handler is initiated to close the swap file on exit and then the swap file is set to 16 megabytes. Finally, the page table is initialized, the memory pages are mprtected, and a signal handler is instantiated to handle bad access to protected memory pages.

Allocation

The `allocateFrom()` function allocates memory from a given partition. It uses a first fit algorithm allocating from The first free "block" it finds that is big enough to satisfy the request. If the "block" is too big, it is split up into two "blocks" where the first "block's" payload is set to the size of the request and used for the allocation. A "block" is too big if its payload is greater than the requested size plus "block" metadata. If the found "block" is not too big it's used for the allocation without splitting. A "block" used for allocation is set to used and a pointer to the payload is returned. If there are no "blocks" in the specified "partition" to satisfy the request, `NULL` is returned.

Allocating as the Thread Library

Allocating from as the thread library with `myallocate()` simply returns a call to `allocateFrom()` using the thread library's partition.

Allocating as a Thread

All threads share the same memory space. This is possible because the threads' memory space is divided into pages giving an illusion of contiguous memory. There are also pages that reside in the swap file giving an illusion of an abundance of memory so when. Pages in memory are protected so if a thread tries to access an address that currently points to a page it doesn't own, the signal handler `onBadAccess()` will be fired. When `onBadAccess()` is called, it first calculates the page number the current thread tried to access. The signal handler then searches the page table for the appropriate page, and if it can't find the page it assigns a new page to the thread. The target page and the page currently at the accessed address are both unprotected and swapped. After the swap, the page that was swapped out is protected. If the thread has been assigned a new page and the new page is the first page assigned to the thread, the page is initialized by setting the metadata and creating a partition that fills the page.

Allocating as a thread should be done using the `threadAllocate()` function. This function initializes the thread library if it hasn't already been initialized. This allows for the use of thread allocation without having to create a thread first. `threadAllocate()` also returns NULL if the requested size is `0`. If the requested size is greater than 0 a call to `myallocate()` as a thread is returned.

When the `myallocate()` function is called as a thread, it blocks the scheduler to ensure thread safety. It then calls `allocateFrom()` using the thread's "partition" and stores the return value in `ret`. As long as `ret` is equal to `NULL` and the current thread can be assigned a new page, the thread's "partition" keeps getting increased the size of one page and `allocateFrom()` is called again with the extended "partition" and the return value is stored in `ret`. Once this is done, `ret` is returned. Essentially, this process only returns `NULL` if the thread's current "partition" doesn't have the requested memory, there are no new pages, or the thread already has enough pages to occupy the whole memory space.

Because an allocation from `myallocate()` is only accessible by the thread that called the function for that allocation, the `shalloc()` functions allows for allocations that can be shared between threads. If the specified size is 0, `shalloc()` returns `NULL`. If the specified `size` is greater than 0, `shalloc()` simply returns a call to `allocateFrom()` with the shared memory "partition".

Deallocation

The `deallocateFrom()` function deallocates memory that was previously allocated with `allocateFrom()`. If the supplied pointer resides within the given "partition" deallocation proceeds otherwise no action is taken. Deallocation starts by finding the "head" and "tail" of the "block" that's referenced by the supplied pointer. `deallocateFrom()` then coalesces the "block" with its immediate neighbors if they are free. Finally, the resulting block is set to free.

Deallocating as the Thread Library

Deallocating with `mydeallocate()` as the thread library simply calls `deallocateFrom()` using the thread library's "partition".

Deallocating as a Thread

Calling `mydeallocate()` as a thread first tries to call `deallocateFrom()` as with the thread's "partition" and if that isn't the correct "partition" it calls `deallocateFrom()` again with the shared memory "partition".