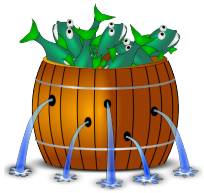

ADVANCED OPERATING SYSTEMS
Milestone 4: Self Paging and Process Management
Fall Term 2016

Assigned on: **28.10.2016** Due by: **04.11.2016**

1 Overview

In this milestone, you will implement a page fault handler as the core component for *self-paging* user processes. You will use the functionality you implemented in the previous milestones to obtain and manage RAM capability to serve your application's memory needs. The reading material for self-paging is on the course webpage, under week 6.

You won't be building complete demand paging to disk (or some other storage device at this stage), but you will have to handle page faults. The goal for this week is to perform *lazy allocation* of the program heap (the memory allocated by `malloc`): instead of making sure that all the physical memory for the heap is there at the start of the program, you will simply reserve the *virtual* memory, and then fill it on demand with physical memory in response to page faults in that area of the address space.

Additionally, you have to implement a spawning service, which applications can use to spawn processes. This includes implementing some RPCs that allow clients of the spawning service to request applications to start, and to get a list of all the applications that exist in the system.

The work consists of:

- implement an exception handler
- Heap management with lazy allocation
- Implement the self-paging principle
- Handling of page-faults in userspace
- Spawning service and associated RPCs

2 Getting prepared

As with the previous milestone, you need to pull additional code from the repository we provided to get the additional files required to complete the milestone.

3 Implement an Exception Handler

To handle page faults you need a way to run custom code when an exception occurs. This can be accomplished by setting an `exception handler`. You can set an exception handler for a thread using the function `thread_set_exception_handler` in the threads package in `libaos`.

As a first step it might make sense to just print something when you get an exception and stop. To test, you can trigger an exception by accessing memory that is not mapped yet.

4 Design the Address Space Layout

You need to think about how to represent the address space at user level, so that your paging code knows which regions are occupied, which are free and also which regions actually belong to the heap. You should already have the functionality to ask for a free region or to map memory at a specified address. In this milestone you will be handling page-faults and you need to be able to identify the region where the pagefault happened, only the faults that occur on a heap address are handled.

We won't give you much of a template. You can do this any way you like (a list, a tree, a hash table, ...) but keep in mind which operations need to be efficient at runtime, and be prepared to explain your choice in the marking session.

Note: you should have all the state of your implementation in the `struct paging_state` of which you'll find an instance called `current` setup as a global variable in `lib/aos/paging.c`.

4.1 Implement Page Fault Handling

You should now implement page fault handling in your exception handler. You should also think about detecting NULL pointer dereferences and disallowing any mapping outside the range(s) that you defined as valid for heap, stack, etc. Also consider adding a guard page to the process' stack — this makes debugging overflows a whole lot more pleasant.

4.2 Implement Dynamic Memory Allocation

For this step you should implement `morecore`, which is the backend for malloc and friends. You should do this without resorting to having a large static array (as the supplied code does).

Instead, you should now be able to just reserve a region of virtual addresses as the heap (using `paging_alloc`) and return one of these addresses as new heap space for `morecore`. Following this, the application can handle page faults, and have the page fault handler allocate physical memory on demand to back the reserved virtual addresses.

Note: there may be multiple threads accessing the heap of your programm at the same time. You will need to take care that you are handling a pagefault for a heap page only once and don't install a frame twice.

5 Spawning Service

In the previous milestone you implemented the functionality to spawn new processes. It's now the task to offer this service to other domains that are already running in the system. For this purpose you need to implement a spawning service that handles the following RPCs to spawn, get a list of running processes and query the process name for a given identifier. You can either implement the functionality in `init` or as a separate domain.

Domain spawning On the service side you will need to provide the appropriate message handler functions that glue messages with your process spawning functionality you have implemented in an earlier milestone. Note: think of which domains can be spawned by the service and which not.

Domain management You will need to keep track on which domains have been spawned, and which are running. We want you to implement domain identifiers which are unique among all the running domains in the system. You can implement the domain identifiers as you like e.g. as a monotonically increasing integer.

At runtime, any domain should be able to get a list of domains that are currently running in the system as well as be able to query the names of those domains. Hence, your implementation should notify the spawn service when your domain has exited for instance.

EXTRA CHALLENGE
Handle the kill command. You may encounter that a domain runs longer than it should and you want to terminate it. In this extra challenge you are asked to implement a way to terminate a running domain. There are multiple ways to accomplish this. For instance you can keep the capabilties of the domain in the spawing service and make the domain unrunnable or you can ask the domain to gracefully terminate by sending a message to it.

RPC Interface The interface you should provide is stated below. You notice, that spawn takes a `core` argument. This will be used in a later milestone and you can just pass 0 for this argument in the current milestone.

include/aos/aos_rpc.h

```
/**
 * \brief Request process manager to start a new process
 * \arg   name the name of the process that needs to be spawned (without a
 *        path prefix)
 * \arg   newpid the process id of the newly spawned process
 */
errval_t aos_rpc_process_spawn(struct aos_rpc *chan, char *name,
                               coreid_t core, domainid_t *newpid);

/**
 * \brief Get name of process with id pid.
 * \arg   pid the process id to lookup
 * \arg   name A null-terminated character array with the name of the process
 *        that is allocated by the rpc implementation. Freeing is the caller's
 *        responsibility.
 */
errval_t aos_rpc_process_get_name(struct aos_rpc *chan, domainid_t pid,
                                  char **name);

/**
 * \brief Get process ids of all running processes
 * \arg pids An array containing the process ids of all currently active
 *           processes. Will be allocated by the rpc implementation. Freeing is the
 *           caller's  responsibility.
 * \arg pid_count The number of entries in 'pids' if the call was successful
 */
errval_t aos_rpc_process_get_all_pids(struct aos_rpc *chan,
                                      domainid_t **pids, size_t *pid_count);
```

THE MILESTONE

- Implement self-paging
- Show your implementation handling page faults.
- Show that you have a running spawning service.

ASSESSMENT

- taking and handling page faults.
- Explain your user-level address space representation.
- `malloc` without static memory allocation
- Your implementation should be able to handle dynamic allocations of total size of at least 32MB.