## Lab 7 general design

Idea: free physical pages by swapping or just deleting when under memory pressure

Direct reclaiming:

* When out of memory error in pmap.c using pagealloc lock
* Try to swap memory out first, if this is not possible, use OOM killing
* Difficulty: we are out of memory, but these functions need memory to actually work (stack variables etc?). We need to reserve a small memory section for this.

Periodic reclaiming:

* Every X time, check if you are under memory pressure and if so, swap out
* Use 1 kernel thread for this since you can’t swap with 2 threads because of locking
* If swap is not possible, don’t do OOM killing, only do this when out of memory since this is a last resort

Periodic and direct reclaiming use partially the same function since most of the functionalities between these two are shared. Differences:

* Direct wants OOM killing, periodic not -> use argument to check this
* Periodic has to check if reclaiming must be done at all (check with watermark) since direct reclaiming is always called when out of memory but for periodic it may not be needed
* Call from direct reclaiming should be not interruptible since we need to free memory else nothing can function anymore, while calls from periodic reclaiming don’t have that high of a priority. See section “swapping”.

**It’s happening in kernel so it’s not interruptible.**

Other issues:

* How to actually implement the small section of free physical memory that is reserved for the case when we’re out of memory? We only have to use these pages as stack since we’re not going to actively ask for pages. (Should fit the whole RSS score list at least)

**No problem, just alloc a page and map it on a fixed address. When needed, we will use it by the OOM killer. It actually doesn’t have to be used for stack, we can have global variables preallocated on that page.**

* Use the page\_free\_list to determine how many free pages are left?

**Now we have a counter for this, that is updated with each page\_alloc/page\_free.**

And when should be exactly have memory pressure?

**Now I set it to 512 small pages (which is 1 huge page btw) but we can change it.**

And until which limit do we have to free pages, so we don’t have memory pressure anymore (just freeing 1 won’t be enough since we will have a lot of 1 page swaps then which is completely inefficient)

**The same number (512)? Or more?**

## OOM killing

Last resort: kill a process

* Kill an environment by freeing its whole page table and all the memory that is used
* This is probably easy to implement since (I think that) the env\_free and env\_destroy functions already handle this.
* Use RSS heuristic to determine which env to kill

RSS heuristic: Total amount of memory used by an environment

* So only the physical memory used in the user space part of the pml4 since kernel is shared
* How to determine this? We can just walk the pml4 and for every user space virtual address (< kern\_start or user\_end variable) do +1 to counter if its actually mapped
* But what about multiple virtual pages in one environment that are mapped to the same physical page? And isn’t this algorithm way too slow? But I don’t think there are many other options then to brute force this
* Maybe use a kernel thread to periodically update the list of RSS heuristic scores? In that case the list must be global and not local to the OOM killing function and we don’t have to update the list in the OOM killing function but since the chance that we will ever do OOM killing in a normal system is super low, I think that this is just a waist of computation time (to use a kernel thread for the updating).
* **I would add new fields in the env struct – counters of used memory. This can be updated during process initialization and then on demand (in page fault handler). We can update it if we page\_alloc (= total number of physical pages used) or page\_insert (= total number of virtual pages used). That way, the algorithm is not slow and we are always ready.**

**OOM killer can then walk all envs and read the values/compute the scores (using its memory).**

What should the OOM function do:

* Get called in the main memory\_pressure function if swapping did not work and if we had an out of memory error (thus not periodic reclaiming)
* Update the RSS heuristic
* Pick the environment with the highest score (which uses the most memory) and do env\_free and env\_destroy for this environment
* Need an env lock to check all the mapping of all the envs, no pagealloc lock needed (since env functions will handle the actual freeing)

Other issues:

* What if the environment you want to delete is still running? Then just send a system call that the environment is finished so it kills itself? Seems like the easiest thing to do.

**But that’s tricky, how are you going to tell it to kill itself? What do you mean by “send a system call”? System calls are sent from userspace to kernelspace, not vice versa. The best thing would be to send it a signal but we don’t have this.**

**Can’t we send an interrupt to a core that is running the environment and the core will stop executing and destroy the environment?**

* You are out of memory while handling this, so we need to reserve a free space, so this function can work (see lab 7 general design – direct reclaiming)

**UPDATE October 13th**

Design decisions & how we implemented it:

* Two new fields are added in env struct – num\_alloc and num\_swap.
* Field num\_swap is the number of swapped out pages and is modified only by functions swap\_in/swap\_out.
* Field num\_tables is a counter for allocated page tables. It is incremented in entry\_in\_table() function for curenv (if there is any) and in fork syscall in copy\_pml4() for new env.
* It is decremented when the environment is destroyed. For this, I had to change prototype of env\_free\_page\_tables() which is not good so it has to be solved somehow else.
* Field num\_alloc is the number of allocated pages for the environment. It is incremented on page fault for the current environment (if a page was loaded) – because all pages of the environment are demand-paged, and during swapping in.
* It is decremented when the environment is destroyed (there is no other way how to free a page from the environment, correct?) and during swapping out.
* The RSS heuristic is implemented in oom\_kill\_process() which is called from direct page\_reclaim() if swapping was not possible

TODO

* Send IPI to another core to actually kill the process
* Pre-allocate the memory OOM killer uses (now there are local and global variables but that probably wouldn’t work)

## Page replacement FIFO

Algorithm and data structure to decide which pages to swap out

**Part 1: swap out**

FIFO queue for page faults general design

* A FIFO queue is needed which stores all the page faults of all the environments in chronological order.
* Multiple designs possible (head or tail is oldest page fault), doesn’t matter much
* Examples here is for tail is the oldest, this is done so inserting a page fault is O(1) while getting the oldest one is O(n). This is better than the reverse since we will have more page faults then swapping cases so its more efficient
* OR we can use a global variable to point to the end of the list, then it won’t make much of a difference which of the 2 designs we choose

FIFO queue workings (don’t worry about design, this is a general thing)

* Append or prepend the page faults in chronological order to the linked list so the head is the oldest
* Swap the oldest (one or multiple) pages which are linked to the page faults out.
* For this we need the following: The actual physical page which was pointed loaded in the page fault. All the page table entries (virtual memory) which point to this physical page since we have to clear those. Last thing is to map these virtual addresses to the segments on disk so on page fault we know where to load them from

Other issues:

* How to determine how many pages we need to swap out? For out of memory, swapping 1 out wont help since we will then have a out of memory error the next time someone asks for a page so that wont help. Maybe free to the same limit as we would when under memory pressure. The limit will be somewhere X pages below the memory pressure limit.

**I agree with a limit, we just need to find a number.**

* Have to check if enough pages in the page fault linked list exist to actually satisfy the request to free pages until X under the memory pressure limit, if this list is too short (somehow, don’t know how that would be possible, maybe its impossible) then go to OOM killing.
* How to check if there is enough space on the swap disk to store our data to? If there isn’t enough space, error so we will start OOM killing

Reverse mapped list between physical pages and page table entries

* **I would update this on each page mapping.**
* **Do we map a page in any other way that page\_insert? If so, when? When two processes share a page, it’s the same PTE, so it shouldn’t be a problem.**

Mapping between page table entries and swap entries

**Can we set a “swapped out” bit in the PTE**

Swap-out short:

* Check how many physical pages we need to free to satisfy our memory pressure
* Get physical pages which we can free by getting the first X page fault pages from the queue (or use the CLOCK algorithm to update the list, see “page replacement CLOCK” section)
* Write the contents to swap disk and free the actual memory pages
* 0 all the page table entries pointing to the physical page using reverse mapping
* Update the swap entries to page table entries data structure

Swap-in:

* Check in the page fault handler if the faulting page table entry maybe has an entry in the swap -> PTE data structure before just getting a new physical page with value 0 like we do now
* If there is such an entry, read the information from the disk (see “swapping” section)
* Allocate a new physical page and copy the just read data to this page
* Update the faulting PTE as usual
* Update the swap -> PTE mapping, if no PTE points to the swap area anymore, clear the swap area somehow so it can be reused
* **!!! How are we going to keep track of the free sectors on the disk? Are we going to use a datastructure in memory or on disk? Because as you say, the pages can be swapped in/out randomly, so it won’t be linear! We need to keep track of it:**

**a) 1 bit per half-sector on the disk (used/unused)**

**b) free list**

* **Also, beware that a sector on a disk can possibly store 2 swapped out pages. How are we going to handle the cases when just one of them is swapped in?**

**a) swap both in and store the other one in a cache**

**b) read the whole sector and then write just a half of it back**

**c) read the whole sector, don’t change it (= don’t zero the other half), just set this half of a sector as unused in the datastructure**

TODO

* Where to store data structures -> pmap.c etc
* How will the data structures exactly look like?

**UPDATE October 13th**

Design decisions & how we implemented it:

* FIFO queue of faulting pages is a double-linked list of physical pages that were loaded as a result of a page fault. Fields fault\_next and fault\_prev were added in the page\_info struct. We store head and a tail of the faulting list in page\_fault\_head and page\_fault\_tail variables.
* Under memory pressure, we will always swap out more than 1 page → we want at least FREEPAGE\_THRESHOLD free pages
* We have a counter in swap.h nfreepages that keeps track of the number of free physical pages (updated with each page\_alloc and page\_free). It can be queried by function available\_freepages(size\_t num) (= are there num available pages?)
* How to check if there is enough space on swap disk – we have a new datastructure keeping track of available swap slots (=8 aligned continuous sectors on disk that can store 1 page). For this, we have a new struct swap\_slot and double-linked list free\_swap\_slots.
* slot2sector and sector2slot macros are used to convert a swap\_slot datastructure to the number of sector where to write/read
* swap\_slots is an array of all swap\_slots (just like pages) and is used by these macros
* swapped\_pages is a linked list of all non-free swap slots – this will be used if we have a page fault on a swapped out page and we need to search fast where it was swapped out.
* Structure swap\_slot has a field ptes (or similar) to keep track of all PTEs that pointed to the physical page before it was swapped out, so that the mappings can be recovered after it is swapped back in
* Functions swap\_in and swap\_out take care of the communication with disk to swap a single page in or out. They also update all needed datastructures affected by the swapping. Right now, the functions are only partially implemented but there are leading comments.
* If a page is swapped, its PTE is cleared but SWAPPED bit is set. If we want to swap it back in, we need to walk the swapped\_pages datastructure and look for the corresponding PTE/VA/env before we bring it back

TODO

* Remove physical page from the FIFO queue if it is already there before appending it to the end.
* Check whether disk sector 0 can be use. Currently we are using it but maybe we should skip it.
* Finish implementation of swap\_in and swap\_out and make it non-blocking
* Problem – if we are going to record only PTEs that pointed to the physical page being swapped, we can’t flush the TLB! For that, we also need the virtual address and environment! Also, TLB cannot be flushed for any other environment than the current one! We need to solve this. Maybe change TLB implementation and remember more than PTEs in the swap\_slot structure?
* Create a reverse mapping between physical pages and their mappings (as above, don’t know what should be there. PTEs? Envs? VAs?)
* Set a swapped-out bit in PTEs if a page is swapped and change all the checks “entry != 0”
* Update the page fault handler so it will check the swap -> PTE mappings

## Swapping

How to exactly use the disk so memory pages can be written and read from it

Don’t know how the driver exactly works yet. I guess its something with sections on the disk.

We have to read and understand ide.c / ide.h as fast as possible so we can complete our design but it doesn’t really matter how this works since the general idea of a swap disk remains the same.

**Update October 13th**

Interaction with disk is in swap\_in and swap\_out functions. For now, you don’t have to bother with it more – we only need to come back to it when we want to make it non-blocking.

## Page replacement CLOCK

Improvement of the simple queue data structure to check which physical pages to swap out

* Principle of second chance: When determining which pages to swap out, append to head to the back of the list if R=1, and set R=0 for this entry
* When the head of the list has R=0, swap this page out
* Using this, every faulting page has a second chance since it will be appended to the back of the list once
* Handy to have not only a pointer to the front of the list, but also to the back since it will be much faster. This means that the list also has to be doubly linked since it would be easier to replace head and tail since we know our neighbors.
* In summary: don’t have to change much to make this working

**UPDATE October 13th**

Other TODOs

1. Rewrite VMAs because we are out of memory. Currently, with your and my fields in page\_info, it crashes, so I commented the list of PTEs out from the page\_info structure.

2. Design thing. If we remember PTEs and not VAs, how do we invalidate cache? Also, TLB can be invalidated only on a currently running process, this needs to be changed.

But even if we remember VAs, we don’t know whose process’ TLB needs to be flushed :(

So do we want to walk all the page tables? Or remember a lot of data?

3. Add SWAPPED OUT bit and modify how entries are checked.

4. Is block 0 on the disk always reserved? If so, we shouldn’t use the first 8 sectors

5. RSS: alloc + tables are updated with page\_alloc (it’s one shared counter), but we need to solve fork separately.