CSE 506: Operating Systems (Fall 2014)
Prof. Don Porter, Stony Brook University

PAGE SWAPPING IMPLEMENTATION IN JOS

by Sarath Swaminathan Rami **(109777342)** and Subin Cyriac Mathew **(110049777)**

Page Swapping is the method by which OS gives processes the the illusion that there is more mmeory available for use than the amount of Physical RAM. OS writes least recently used pages into a swap device when it is under memory pressure, thus freeing up memory for the current process.

SWAPPING IMPLEMENTATION DETAILS:

1. We initialize the swap system by calling swap_init() in kern/swap.c
2. We probe for all online disks
3. We set the drive number of the swap disk
4. We test IO by writing and reading to the first block and check for consistency
5. We initialize an empty LRU list
6. We initialize the SWAP bitmap to be free (all 1s)

IDE initialization & IO for secondary controller:

JOS uses the first disk (hda) for its boot image and (hdb) for the file server disk image. Adding a third disk requires writing the driver for the secondary controller in ide.c. Qemu uses LBA28 disks so we made the following changes to read and write to the disk in the secondary controller:

To read a sector:

1. We send a NULL byte to port 0x171: outb(0x171, 0x00);
2. We send a sector count to port 0x172: outb(0x172, 0x01);
3. We send the low 8 bits of the block address to port 0x173: outb(0x173, (unsigned char)addr);
4. We send the next 8 bits of the block address to port 0x174: outb(0x174, (unsigned char)(addr >> 8);
5. We send the next 8 bits of the block address to port 0x175: outb(0x175, (unsigned char)(addr >> 16);
6. We send the drive indicator, some magic bits, and highest 4 bits of the block address to port 0x176: outb(0x176, 0xE0 | (drive << 4) | ((addr >> 24) & 0x0F));
7. We send the command (0x20) to port 0x177: outb(0x177, 0x20);

To write a sector:

Same as above, but we send 0x30 for the command byte instead of 0x20: outb(0x177, 0x30)

Checking for device status is done by doing a read of the status port inb(0x177) and checking if the busy bit(0x40) is set.

DATASTRUCTURES:

PageInfo Structure

The PageInfo structure has been augmented with two new fields:

```
uint16_t flags;  /*  PG_SWAPPABLE  0x1 */
uint8_t env_bitmap[NENV/8] ;
```

All the pages allocated by the user are swappable. This includes page allocation/sharing via the following interfaces.

- sys_page_alloc()
- sys_page_map()
- sys_ipc_try_send()

The env_bitmap is necessitated by the lack of a kernel memory allocator. That would have been a project on its own. :).
Similar metadata

LRU List

A Least Recently Used(LRU) list of PageInfo pointers is maintained.

```
typedef struct lru_list {
   struct PageInfo *head;
   struct PageInfo *tail;
   int size;
} lru_list_t;
```

Whenever a page is being allocated by a process, the corresponding PageInfo structure is added to the LRU list at the tail using add_to_lru_list().
A page being shared by another process via sys_page_map() also moves the PageInfo from where it was on the LRU list to the end of the tail, the idea being that the page could be accessed soon by the new process and we don't want the overhead of swapping here. This is an optimization.

The page is now allocated using the flag PG_SWAPPABLE. The env_bitmap[] of the PageInfo is also updated with the environment Index. So, for shared pages, multiple bits may be set in env_bitmap[].

Kernel Thread

A Kernel Thread(swapper) is created at init time; it runs periodically every SWAP_INTERVAL_MS(1 sec by default).
In every run, it calls update_lru_page_list()

- it will walk through the Page Tables of each environment in the system.
- If a PTE is found to have been accessed recently(since the swapper last run), the PageInfo corresponding to the Physical page is moved to the end of the LRU list.
- The PTE_A flag is cleared for each entry which has the PTE_P bit set, so that the swapper can figure out if the page was accessed in the time interval between its next run.

Swapping Details:
When a new page alloc request is being handled in page_alloc():

- If the system is low on memory(i.e when the remaining RAM is less than a configured threshold of START_SWAP_AT_PERCENTAGE), a routine get_page_from_swap_ss() is called which does the following
    - Finds the first PageInfo in the LRU list(lru_list.head)
    - For the first process the corresponding physical page is present in the PTE
        - Find a free swap block. This may call the Out Of Memory Killer if the swap space is full.
        - Write the contents of KADDR(page2pa(pg)) to that block.
        - Save the env_bitmap field in the swapped_page_bitmap[] field corresponding to the block so that it can be referred to while swapping back in.
        - Set their PTE entry to contain the disk block number and the PTE_SWAP flag.
        - Remove the PTE_P flag from the PTE.
    - For the rest of the processes, just do the last two steps as was done for the first process.
    - Return the PageInfo structure.

Swap Space Full Scenario:
We have added a new field num_swapped_pages to the Env structure. This field keeps track of the number of swapped out pages for each environment. Whenever the swap space is full, we call an Out-Of-Memory Killer function “find_victim_env()” which will find a victim environment to destroy.
We don't randomly kill any environment, but the environment with the largest number of swapped out pages.
The reasoning is that

- The process has a large number of swapped out pages, so killing it will free max. swap space.
- The proc ess is not actively using these pages(since it is still present in the swap space), so it is an unfair process to keep allocating and not freeing pages for other processes in the system.

Note that in some cases, the environment which is trying to allocate a new page will itself get destroyed as per this algorithm if it's the largest consumer of swap Subsystem and this case is also handled.

Page Fault Handling:

Non-Shared Swapped Out Page Case:
The kernel page_fault_handler()

- checks if the PTE entry has the PTE_SWAP bit set
- If set, it allocates a new page using sys_page_alloc(). sys_page_alloc() will return a page no matter what.(even if it has to swap out another page to allocate a new page)
- it extracts the swap disk block number from the PTE entry, and reads in from the disk to the faulted va.
- Returns to user space.

Shared Swapped Out Page Case:

- Same as the above case.
- Additionally, it calls update_shared_page_ptes() which will
    - run through the Page tables of each environment present in the swapped_page_bitmap[] corresponding to the block, and fix up its page table entries.
    - Specifically we look for the disk block number in their Page tables and replace it with the PageInfo structure which we just allocated for the other process.
    - The PTE_P bit is also set so that they won't fault again.
    - The corresponding swap block is freed so that it can be used by the Swapping Sub System.
    - The PageInfo structure is moved to the end of the LRU list as it is being accessed.

Swapped out COW Page Case:

Same as the above 2 cases, additionally, it will now upcall the user space COW page fault handler to handled the duplication of pages.

Page Unmap of a Swapped Out Page:

When a swapped out page is unmapped, the block is released to the Swapping Subsystem if it was non-shared. If it was shared and we are the last reference, the block is released as well.
The same process is follwed when an environment with swapped out pages is destroyed.

OTHER FEATURES

1. As part of the swapping implementation, we needed a kernel thread(Env) which would periodically wake up and update the LRU list.
We created a new environment type ENV_TYPE_KERNEL_THREAD.
These kinds of environments can be initialized using
env_create((uint8_t *)func_ptr, ENV_TYPE_KERNEL_THREAD);
where func_ptr is the address of the entry function.

The kernel thread can run on any CPU, whenever it gets scheduled, it uses the kernel stack of the appropriate CPU to run.

```
 env_run()
..
   if(ENV_TYPE_KERNEL_THREAD == curenv->env_type){
       curenv->env_tf.tf_rsp = KSTACKTOP - (KSTKSIZE + KSTKGAP) * cpunum();
    }
    env_pop_tf(&curenv->env_tf);
..
```

2. We also needed a mechanism where an environment could put itself to sleep for a pre-defined time and the scheduler would make it RUNNABLE after the specified interval. We didn't want the overhead of being woken up before the time interval has elapsed and doing a sched_yield() if the time has not elapsed, so we created a timer infrastructure.
Any environment(be it a user or a kernel thread), when it needs to sleep for any number of

milliseconds, adds itself to the timer list and yields. The timer datastructure is looked up by the scheduler every tick and will make the environment RUNNABLE if the time has elpased.
This is done by the handle_timeouts() routine.

3. We also investigated using a regular large file as the swap space by interfacing with the FileSystem Server. To that end we implemented a writable file system as well. But we didn't go down that path as it was not efficient blocking the faulting process and then invoking the FS to write the LRU page to a swap file, then making the original faulting process runnable.

DEMO DETAILS:

Tests are run using the following command:

make run-user_swap_shared_page-nox-gdb

Please note that we have limited the JOS memory to 90MB in the GNUMakefile to test the results faster.

1) Handling out of memory issue:

In swap.h, minimize the swap disk space:
    #define SWAP_BLOCK_SZ 1
Keep a higher swap threshold for the swap sub system to be called
    #define START_SWAP_AT_PERCENTAGE 87

Output:

kernel Thread:[00001000] : swapper
Done updating LRU list
PHY MEMORY REMAINING = 86%
Child ENV[00001001] done sleeping.
Available RAM below Configured Threshold, SWAP SUB-SYSTEM CALLED
Walking 2th ENV[4097] 80047283a0
Out of Swap
Killing ENV[00000001] to free pages

2) Swapping out

In swap.h, minimize the swap disk space:
    #define SWAP_BLOCK_SZ (128*1024)
Keep a higher swap threshold for the swap sub system to be called
    #define START_SWAP_AT_PERCENTAGE 70

Output:

Available RAM below Configured Threshold, SWAP SUB-SYSTEM CALLED
Walking 2th ENV[4097] 80057ce240
Page 00000080057ce240 from ENV[00000001] swapped to disk

3) Swapping back in

Swap-out LRU page, on future access, handle fault in trap(), check block entry in PTE, reload it by doing a ide_read() to a newly alloced page.

In swap.h, minimize the swap disk space:

#define SWAP_BLOCK_SZ (128*1024)

Keep a higher swap threshold for the swap sub system to be called

#define START_SWAP_AT_PERCENTAGE 70

Output:

faulted for 0000000000a00000
Swapped page 0000000000a00000 faulted for ENV[00000001]
Available RAM below Configured Threshold, SWAP SUB-SYSTEM CALLED
Walking 2th ENV[4097] 80057ce740
Page 00000080057ce740 from ENV[00000001] swapped to disk
delete_from_lru_list: 00000080057ce740
SWAP SUB-SYSTEM PAGE = 00000080057ce740
Inserting 0000000001af2000 into ENV[00001001]'s Page Table
add_to_lru_list: 00000080057ce740
LRU Size = 67
Inserted swapped page 0000000000a00000 back into ENV[00000001]'s PT
Before Page Lookup
After Page Lookup
Moving 00000080057ce740 to the end of the LRU list
delete_from_lru_list: 00000080057ce740
add_to_lru_list: 00000080057ce740
LRU Size = 67
Data consistent across Swap

4) LRU list update

Enable printing of the LRU list by defining a macro DEBUG_SWAP in kern/swap.c.

Output:

ALLOC of 0000000000a3d000 by ENV[00001001] done.
ENV[00001001] Running...
ENV[00001001] done sleeping.
Inserting 0000000001afc000 into ENV[00001001]'s Page Table

add_to_lru_list: 00000080057ced80
LRU Size = 62
00000080057cc620 ->00000080057cc760 ->00000080057cc800 ->00000080057cc8a0 ->00000080057cc940 ->00000080057cc9e0 ->00000080057cca80 ->00000080057ccb20 ->00000080057ccbc0 ->00000080057ccc60 ->00000080057ccd00 ->00000080057ccda0 ->00000080057cce40 ->00000080057ccee0 ->00000080057ccf80 ->00000080057cd020 ->00000080057cd0c0 ->00000080057cd160 ->00000080057cd200 ->00000080057cd2a0 ->00000080057cd340 ->00000080057cd3e0 ->00000080057cd480 ->00000080057cd520 ->00000080057cd5c0 ->00000080057cd660 ->00000080057cd700 ->00000080057cd7a0 ->00000080057cd840 ->00000080057cd8e0 ->00000080057cd980 ->00000080057cda20 ->00000080057cdac0 ->00000080057cdb60 ->00000080057cdc00 ->00000080057cdca0 ->00000080057cdd40 ->00000080057cdde0 ->00000080057cde80 ->00000080057cdf20 ->00000080057cdfc0 ->00000080057ce060 ->00000080057ce100 ->00000080057ce1a0 ->00000080057ce240 ->00000080057ce2e0 ->00000080057ce380 ->00000080057ce420 ->00000080057ce
LRU Size = 62
Moving 00000080057cc620 to the end of the LRU list
delete_from_lru_list: 00000080057cc620
add_to_lru_list: 00000080057cc620
00000080057cc760 ->00000080057cc800 ->00000080057cc8a0 ->00000080057cc940 ->00000080057cc9e0 ->00000080057cca80 ->00000080057ccb20 ->00000080057ccbc0 ->00000080057ccc60 ->00000080057ccd00 ->00000080057ccda0 ->00000080057cce40 ->00000080057ccee0 ->00000080057ccf80 ->00000080057cd020 ->00000080057cd0c0 ->00000080057cd160 ->00000080057cd200 ->00000080057cd2a0 ->00000080057cd340 ->00000080057cd3e0 ->00000080057cd480 ->00000080057cd520 ->00000080057cd5c0 ->00000080057cd660 ->00000080057cd700 ->00000080057cd7a0 ->00000080057cd840 ->00000080057cd8e0 ->00000080057cd980 ->00000080057cda20 ->00000080057cdac0 ->00000080057cdb60 ->00000080057cdc00 ->00000080057cdca0 ->00000080057cdd40 ->00000080057cdde0 ->00000080057cde80 ->00000080057cdf20 ->00000080057cdfc0 ->00000080057ce060 ->00000080057ce100 ->00000080057ce1a0 ->00000080057ce240 ->00000080057ce2e0 ->00000080057ce380 ->00000080057ce420 ->00000080057ce->00000080057ced80

<u>What does not work</u>

1)We tried adding a 3rd disk(apart from the boot and fs) to QEMU by doing the following:

(1) Using fsformat to create a new blank image (fs/MakeFrag)

(2) Adding the image as hdc(disk 2) in GNUMakefile

(3) Creating a new ide.c to handle the secondary channel. (Ports 0x1F* changed to 0x17* as specified in the manual)

In the new ide.c, we have the following to check if the secondary controller exists:

```
outb(0x173, 0x88);
r = inb(0x173);
cprintf("SECONDARY DISK CONTROLLER%s\n", r==0x88?" present":"not present")
```

The controller does seem to exist, but the device always seems to be busy. We have posted it in QEMU forums and in Piazza, but have not received any ideas to solve it so far. We have switched the positions of the swap-disk and the fs-disk <u>temporarily</u>, since the secondary disk (hdb) in QEMU was working fine.

2) Shared pages (PTE_SHARE) do not handle swapping well in all cases since there is a race condition we have not been able to catch yet.