Embedded Operating Systems

Agenda

- Paging

Paging

Page Fault

- Each page table entry contains frame address, permissions, dirty bit, valid bit, etc.
- If page is present in main memory its page table entry is valid (valid bit = 1).
- If page is not present in main memory, its page table entry is not valid (valid bit = 0).
- This (Invalid entry) is possible due to one of the following reasons:
    - Page address is not valid (dangling pointer).
    - Page is on disk/swapped out.
    - Page is not yet allocated.
- If CPU requests a page that is not present in main memory (i.e. page table entry valid bit=0), then "page fault" occurs.
- Then OS's page fault exception handler is invoked, which handles page faults as follows:
    1. Check virtual address due to which page fault occured. If it is not valid (i.e. dangling pointer), terminate the process. (Validity fault).
    2. Check if read-write operation is permitted on the address. If not, terminate the process. (Protection fault).
    3. If virtual address is valid (i.e. page is swapped out), then locate one empty frame in the RAM.
    4. If page is on disk, load the page in that frame.
    5. Update page table entry i.e. add new frame address and valid bit = 1 into PTE.
    6. Restart the instruction for which page fault occurred.

mmap() Syscall

- Allocate a segment in virtual address space of the process.
    - Create a new VAD to keep information of segment (start, end, prot, flags, ...) and add it into process's VAD list (i.e. memory map of the process -> mm_struct -> mmap).

  - Create new pageastable entries for this segment.
- ptr = mmap(addr, length, prot, flags, file_descriptor, file_offset)
  - arg1: Start virtual addr of process's VAS to which new segment is to be mapped. NULL means any available address.
  - arg2: Length of virtual address segment
  - arg3: Segment protection flags: PROT_READ, PROT_WRITE, PROT_EXEC, or PROT_NONE.
  - arg4: Segment flags: MAP_PRIVATE, MAP_SHARED, MAP_ANONYMOUS, or ...
  - arg5: File descriptor of file to be mapped.
  - arg6: File offset of the file to be mapped.
  - returns: virtual base address of the process at which segment is allocated.
- ptr = mmap(NULL, file_size, PROT_READ, MAP_PRIVATE, fd, 0);
  - When read operation is performed on the segment, page fault will occur.
  - It will allocate an empty frame and load data from the file in it.
  - Now we can access data from the file.
- ptr = mmap(NULL, file_size, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
  - Creates anonymous segment -- not backed by any file on disk.

brk() and sbrk() Syscall

- brk() change program break (in task_struct) to the given virtual address and the end virtual address will be updated in VAD (of heap).
- If new program break is higher than the current, new page table entries will be created. If new program break is lower than the current, corresponding page table entries will be invalidated.
- ret = brk(addr);
  - arg1: New virtual address of program break.
  - returns: 0 on success.
- sbrk() increment/decrement program break by given value/delta. It change program break (in task_struct) and the end virtual address will be updated in VAD.
- If given value is +ve, program break is incremented. The new page table entries will be created. If given value is -ve, program break is decremented. Corresponding page table entries will be invalidated.
- ptr = sbrk(delta);
  - arg1: int by which program break is to be incremented (+ve) or decremented (-ve).
  - returns: new program break.

malloc()

- C library function that dynamically allocate contiguous memory in the heap section of the process.
- The malloc() internally calls brk()/sbrk() or mmap() syscall to allocate contiguous address range into virtual address space of the process.
- When malloc() allocating small segments, it will return base address from existing heap address range. If exceeds beyond heap address range, then it internally calls brk() syscall to increase the heap address range.
- When malloc() allocating larger segments, it will invoke mmap() syscall to allocate a new segment into process's address space.
- Note that brk()/mmap() are allocating contiguous virtual addresses for the process.
- When virtual addresses are allocated to the process their page table entries are created. All these PTEs are invalid (yet to allocate).
- The physical memory is allocated pagewise into the page fault handler, when the memory is used (read/write).

Virtual pages vs Logical pages

- By default all pages of user space process can be swapped out/in. This may change physical address of the page. All such pages whose physical address may change are referred as "Virtual pages".
- Few kernel pages are never swapped out. So their physical address remains same forever. All such pages whose physical address will not change are referred as "Logical pages".
- A process may call mlock() or mlockall() syscall that prevent swapping out the pages. This in turn make pages logical. Refer manual of mlock().

  ```
  mlockall(MCL_CURRENT | MCL_FUTURE);
  ```

- The page stealing process runs in background and swap out the non active pages (i.e. not in current active set) to make room for the new allocations by the running processes.

Page Replacement Algorithms

- While handling page fault if no empty frame found (step 3), then some page of any process need to be swapped out. This page is called as "victim" page.
- The algorithm used to decide the victim page is called as "page replacement algorithm".
- There are three important page replacement algorithms.
  - FIFO
  - Optimal

- LRU

FIFO

- The page brought in memory first, will be swapped out first.
- Sometimes in this algorithm, if number of frames are increased, number of page faults also increase. This abnormal behaviour is called as "Belady's Anomaly".

OPTIMAL

- The page not required in near future is swapped out.
- This algorithm gives minimum number of page faults.
- This algorithm is not practically implementable.

LRU

- The page which not used for longer duration will be swapped out.
- This algorithm is used in most OS like Linux, Windows, ...
- LRU mechanism is implemented using "stack based approach" or "counter based approach".
- This makes algorithm implementation slower i.e. time complexity O(n).
- Approximate LRU algorithm close to LRU, however is much faster.

Thrashing

- If number of programs are running in comparatively smaller RAM, a lot of system time will be spent into page swapping (paging) activity.
- Due to this overall system performance is reduced.
- The problem can be solved by increasing RAM size in the machine.

Global vs Local page replacement

- Local page replacement: If there is shortage of RAM, then swap-out a page of the current process itself.
- Global page replacement: If there is shortage of RAM, then swap-out a page from any process.

Paging optimization

- If page fault handling involves disk IO, it is referred as major fault; otherwise it is minor fault.
- To improve paging performance major page faults (disk io) should be minimized.
- terminal> ps -e -o pid,vsz,rsz,maj_flt,min_flt,cmd

Dirty Bit

- Each entry in page table has a dirty bit.
- When page is swapped in, dirty bit is set to 0.
- When write operation is performed on any page, its dirty bit is set to 1. It indicate that copy of the page in RAM differ from the copy in swap area/disk.
- When such page need to be swapped out again, OS check its dirty bit. If bit=0 (page is not modified) actual disk IO is skipped and improves performance of paging operation.
- If bit=1 (page is modified), page is physically overwritten on the swap area.

vfork()

- fork() create child process by duplicating calling process.
  - Allocates separate memory space for the child process.
  - Create a new thread to execute the child process.
- If exec() is called, it release allocated segments and reallocate as per need of new program to be loaded.
- To avoid this, BSD UNIX developed vfork() system call to create new process. This should be used only while calling exec() in child.
- vfork() creates child process virtually. It doesn't duplicate parent process; rather suspends execution of parent and contine execution of child process until exec() or exit() is called (under parent thread of execution and in parent's address space itself).
- When exec() is called, then actual memory is allocated for the child process and new thread of execution is created for the child process. Hereafter child process executes independent of the parent process.
- terminal> man 2 vfork

Copy On Write

- Modern fork() syscall creates a logical copy of the calling process.
- Initially, child and parent both processes share the same pages in the memory.

- When one of the process try to modify contents of a page, the page is copied first; so that parent and child both will have seperate physical copies of that page. This will avoid modification of a process by another process.
- This concept is known as "copy on write".
- The primary advantage of this mechanism is to speed up process creation (fork()).

Virtual memory

- Virtual memory is the memory that can be given to a process. Typically it includes RAM size (except kernel space) + Swap area.
- https://www.youtube.com/watch?v=qcBIvnQt0Bw&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX

Assignment

1. Implement fast file copy program (assume max file size = 1 GB).
    - step 1: open src file in rdonly mode.
    - step 2: get size of src file (fstat() syscall)
    - step 3: map src file contents to memory using mmap()
    - step 4: create dest file in rdwr mode.
    - step 5: make size of dest file, same as size of src file using ftruncate()
    - step 6: map dest file contents to memory using mmap() -- MAP_SHARED.
    - step 7: copy src file to dest file using memcpy()
    - step 8: close src and dest files.