CS4414: Paging and Protection

cs.virginia.edu/~cr4bd/4414/S2022/paging-and-protection.html

Your Task

▼

1. (required for checkpoint) Add a new system call `int getpagetableentry(int pid, int address)` that returns the last-level page table entry for `pid` at virtual address `address`, or `0` if there is no such page table entry.

2. (required for checkpoint) Add a new system call `int isphysicalpagefree(int ppn)` that returns a true value if physical page number `ppn` is on the free list managed by `kalloc.c` and a false value (0) otherwise.

3. (required for checkpoint) Add a new system call `int dumppagetable(int pid)` that outputs the page table of the process with pid `pid` to the console (like with `cprintf()`) as follows:
   - Only last-level page table entries should be shown;
   - Only page table entries for the user part of memory (bytes 0 through `p->sz` where `p` is the corresponding `struct proc*`) should be output;
   - Output a line starting with `START PAGE TABLE` before the page table entry information, and one starting with `END PAGE TABLE` immediately after. You may, if you choose, put additional text afterwards on these lines;
   - Output a line for each page table entry, with fields seperated by whitespace, in order from lowest virtual address to highest (you may use any kind of ASCII whitespace, even if it (like tabs) doesn't display correctly in qemu's graphical console);
   - Optionally, output a header line describing the fields.
   - For each present page table entry, output a series of space-seperated fields in this order:
     - the v [illegible] entry in hexa [illegible] e first byte of that virtu [illegible] n)
     - the fi [illegible] d `ptetool.c` (who [illegible] `f()`):
       1. [illegible]
       2. [illegible] ccessible and '-' [illegible]
       3. [illegible] '-' otherwise
       4. [illegible] table entry in [illegible] yte of that [illegible])
     - any r [illegible]
   - For non-pr [illegible] om the output or output the [illegible]
     1. the v [illegible] hexadecimal (your choice which)
     2. the text '-' (instead of 'P')
     3. any other fields of your choice (for example, information that will help you debug the later steps)

   For example, one possible output might look like:

```
START PAGE TABLE (pid 54)
0 P U W 8a
1 P U W 8b
2 P - W 8c
3 P U W 8d
4 P U W 89
END PAGE TABLE
```

indicating that a process has five pages allocated, with virtual page numbers 0 through 4, to various physical page numbers in the range `0x89` through `0x8d`, and virtual page 2 is marked as non-user-accessible.

We do not care what your system call does if the `pid` supplied is invalid or if `pid` supplied has its page table modified or freed while the system call occurs.

When successful, your system call should return `0`.

4. (required for checkpoint) xv6 currently allocates stack and heap memory immediately. More commonly, OSs will allocate this memory on demand, saving memory when not all of the stack or heap is used immediately. Modify xv6 to allocate heap memory based on page faults rather than immediately. *It is okay if non-heap memory is still allocated statically.*

   Your automatic allocation scheme:

   - Should initialize all newly allocated memory to zeroes, like happens with a stock versi [illegible]
   - Should kill [illegible] tside of its allocation, [illegible] pt that heap allocations [illegible] llow processes can acces [illegible] f their heap is in the middle [illegible] tock version of xv6.
   - Should kill [illegible] quiring an allocation [illegible] d. Make sure a message i [illegible]
   - Should ne [illegible] e user stack accessible [illegible]
   - Must make [illegible] yet-allocated parts of the hea [illegible] ffer, then using `read()` to [illegible] appens if the process runs out of memory during such a system call and must be killed. (For the most part, this should just require treating page faults triggered by the kernel the same as page faults triggered from user mode.)
   - Should not leak memory.

   For suggestions on how to accomplish this, see the "allocating pages on demand" section of the hints below.

5. Add copy-on-write support for xv6’s `fork()` system call. xv6 currently makes a copy of each page of a process when it forks. Instead, you should not copy the page and instead mark each page as read-only. Then, when a protection fault happens, actually make a copy of the page, update the corresponding page table entry, and mark it as writeable. Your copy-on-write scheme:
    - Should kill a process when it attempts to write to memory, but there is not enough memory to allocate a copy-on-write page. Make sure a message is printed to the console when it happens.
    - Should not leak memory.
    - Should never make the “guard page” xv6 allocates below the user stack accessible.
    - Must make system calls that attempt to read or write to not-yet-allocated parts of the heap work. For example, `malloc()`ing a very large buffer, then using `read()` to fill it must work. We do not care, however, what happens if the process runs out of memory during such a system call. (For the most part, this should just require treating page faults triggered by the kernel the same as page faults

    For suggestions ... on-write support” section of the hi...

6. Run `make submi` ... e submission site.

Testing

system call test

> If you downloade ... ion may have a bug in how it proc

ptetool.c *[last updated* ... `pagetableentry`, `isphysicalpagefree`

If you run it with no arguments, it shows this usage message:

ptetool pte PID VA show what getpagetableentry() entries for the contents of the (last-level) page table entry for pid PID (in decimal) and virtual address VA (in hexadecimal) ptetool dump PID call dumppagetable(PID). PID is specified in decimal ptetool isfree PPN call isphysicalpagefree(PPN). PPN is specified in hexadecimal

To help determine a PID to specify, the xv6 kernel shows a list of active programs when you type control-P. Usually, pid 1 is `init`, and pid 2 is the shell (`sh`). Additional pids are assigned sequentially.

expected xv6 memory layout

Note that xv6 processes generally have a layout where the lowest addreses (starting with address 0) are assigned to code and program data (which will be marked as present and user accessible), then there is a guard page (which will be marked as present, and not user accessible), then there is a stack and one or more pages of heap (which will be marked as present and user accessible). You should be able to observe this using `pte` and `dump` if they are working correctly.

Also, the kernel part of xv6 memory will have virtual page 0x80000 (virtual addresses 0x80000000 through 0x80000FFF) mapped to physical page 0x0 0x80001 to physical page 0x1 and so on, which you should be able to observe using the `pte` command if it is working correctly.

In physical memory, the xv6 loads the kernel code and data at physical address 0x100000 (physical page number 0x100; constant `EXTMEM` in `memlayout.h`), and the memory between the kernel code and data and physical address 0xE000000 (physical page number 0xE000, constant `PHYSTOP` in `memlayout.h`) is managed by `kalloc()` and `kfree()`. (The memory managed by `kalloc()/kfree()` is what is used to allocate pages for user program data [illegible]

tests for allocate [illegible]

`pp_test` tool

> `pagingtestlib.h` [illegible] `pp_test` unintentionally di[illegible] to fix more minor issues (se[illegible]

pp_test.c, which requi[illegible]d 1 April 2022], will use the `getpageta`[illegible] to check that your allocate-on-demand a[illegible]ehaving correctly.

`pp_test -help` shows [illegible] and `pp_test TEST-TYPE -help` sho[illegible]rts.

One place you can start to look for specific commands to use is the list of commands done by `pp_suite`, described below.

assertion failure in mkfs?

If you get an error like `mkfs: mkfs.c:279: iappend: Assertion `fbn < (12 + (512 / sizeof(unit)))` failed` after adding `pp_test.c` or `pp_alloc.c` to the Makefile, this means that your compiler made `pp_test` or `pp_alloc` or some other executable too big for xv6's filesystem to support. Commonly this is because the compiler generated a lot of debug information. Try editing the Makefile rule that starts `_%: %.o $(ULIB)` to add the command `strip $@`; when this is done that Makefile rule should read:

```
_%: %.o $(ULIB)
  $(LD) $(LDFLAGS) -T program.ld --gc-sections -o $@ $^ $(LIBGCC_A)
  $(OBJDUMP) -S $@ > $*.asm
  $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym
  strip $@
```

if you see “error communicating … via pipe”

`pp_test` internally often forks child processes and checks things in both the parent and child processes. When a check fails, sometimes, the parent or child process will terminate abnormally after printing a message. In this case, you may also see some message like “communicating with child process via pipe” in addition to the test failure.

If you see such messages without a test failure, then probably something else happened that prevented the parent or child process from running properly.

running tests when the shell doesn’t work

> Make sure you ha … later; arguments
> are dropped in so …

Since the xv6 shell us … mentation of these when developing your … y, it may be helpful to run tests without the … being run as the `init` (first processs ru …

```
make qemu-nox
```

where `pp_test` match … our xv6 directory. (You can also use this …

This will create a virtu … of the xv6 filesystem, but with `init` program replaced by `pp_test`, then boot xv6 using this virtual disk instead of the normal `fs.img` virtual disk.

dumping page tables for debugging

The `pp_test`’s `alloc` and `cow` subcommands support the `-dump` option, which will call `dumppagetable()` at several points during the text. This can be helpful for debugging what your implementation is doing. We have some example outputs with brief explanations near the end of the hints section.

a test suite