Pintos – Assignment 3

by bvk.chaitanya

IISc

October 30, 2010

What Needs to be Done

- Stack Growth
- Demand Paging
- Swapping
- File `mmap` support

Stack Growth

What is stack growth?

Stack that is setup by kernel **must** be allowed to grow as deep as possible (upto a limit), through function calls, parameters, local variables, etc.

Implementing Stack Growth

How to implement stack growth?

On every pagefault, you see if it *seems* like a stack fault, and map a new user page at that position.

Since, stack location is fixed and grows downwards from `0xC0000000`, we can easily see if a fault address is just below (within 32 bytes) the stack region.

Demand Paging

What does Demand Paging mean?

Any part of a process address space should loaded into memory only when it is really necessary.

Process Address Space

You already know what are the parts of a process address space, right?

`text`, `data`, `bss` and *stack*

- `text` and `data` section contents are present in the executable file.
- `bss` section is simply a data initialized to zero, and is not present in the executable.
- *stack* is completely created at run-time, and not present in the executable.

Implementing Demand Paging

So, what does it take to implement demand paging?

- Every process should have a map of its address space regions and how they can be populated on a *pagefault*.
- Pagefault handler should *lookup* current processes address space regions for a fault address and if it needs to be demand loaded, it should perform a disk read into the respective location.
- Notice that faults to invalid or unallocated regions still need to kill the process.

Demand Paging – Warning

Without *swapping* in place, we should ensure that there is always adequate user memory for a process to complete, *before* returning `exec` successfully.

Swapping

What is swapping?

Swapping is a mechanism to temporarily move some parts of process address space out to a swap-device to *create* more free memory.

Process Address Space

You already know what are the parts of a process address space, right?

`text`, `data`, `bss` and *stack*

- `text` section is executable code and is not modified, so as long as we ensure executable is unmodified during its execution, we can simply free text section pages, without writing to swap device.
- `data`, `bss` and *stack* section pages, once modified, **must** be saved somewhere, otherwise we loose the data.
- Notice that unmodified `data` section pages can be ignored, just like code until they are modified.

Implementing Swapping

How do we go about implementing swapping?

1. We need to pick an user page for swapping out.
2. We need to get a unique free sectors (a slot) from swap-device for this page.
3. We need mark that page as being swapped out and its swap-device location in the corresponding address space region of the process that owns that page.
4. We need to unmap the page from hardware page table and sync it to swap device.
5. Once synced to swap device, we need add the physical page frame to the free pages pool.

Swapping – Warning

You need to ensure that swap space + user memory is always adequate for handling process faults, before returning `exec` successfully.

You may need to ensure at least 6 pages (2 code, 2 data and 2 stack) in memory for every runnable process, otherwise you may not guarantee progress of that process.

File mmap Support

What is File mmap support?

Part of process address space is mapped from a file.

Implementing File `mmap` Support

How to implement File `mmap` support?

If we have demand paging and swapping in place, then `mmap` is simply a special case of it: use filesystem instead of swap-device.