Virtual Memory and Prefetching

2024 Fall ECE454: Computer Systems Programming — Lecture 10

Jon Eyolfson

1.1.0

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

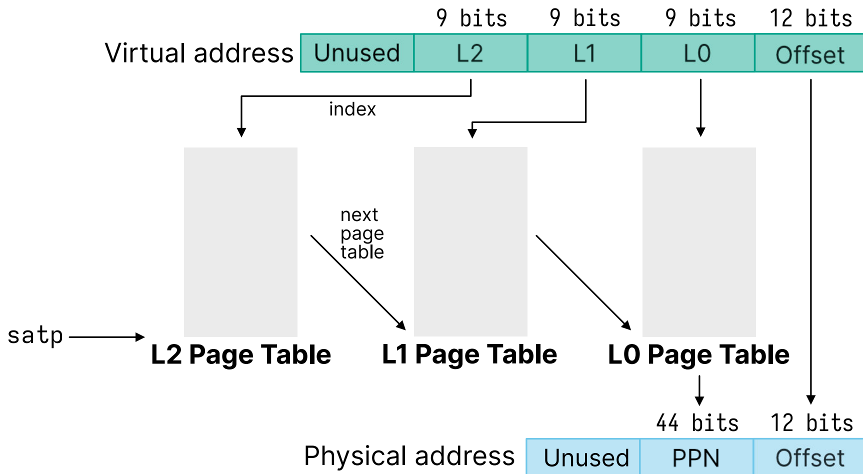
Each Process Uses Virtual Memory

They believe they have access to the entire address range
(no matter the amount of physical memory)

The kernel divides physical memory into pages
(typically 4 KiB aligned blocks of memory)

The kernel maps virtual pages to physical pages

The Kernel Manages Virtual Memory Through Page Tables



Benefits of Virtual Memory

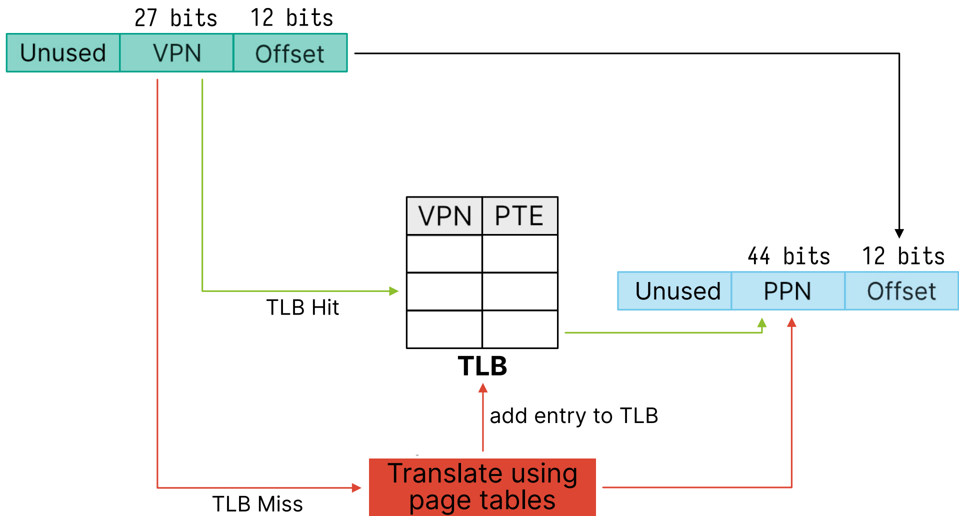
Decouples the memory a process uses from physical memory

The kernel can share memory implicitly (through copy-on-write)
or explicitly (through shared memory)

It’s lazy, the kernel allocates a physical page
when you first use a virtual page

If you run out of physical memory, the kernel
can move pages to swap space on disk

The TLB is a Cache for Virtual Address Lookups



The TLB is Typically Integrated with L1/L2 Cache

This cache significantly speeds up virtual memory access

Typically the TLB can store enough entries to translate everything in cache
(you will have a performance issue if every access is a new page)

≈ 64 L1 TLB entries, 2048 L2 TLB entries

The amount of memory you can access using TLB entries is called TLB reach

Programming with Virtual Memory in Mind

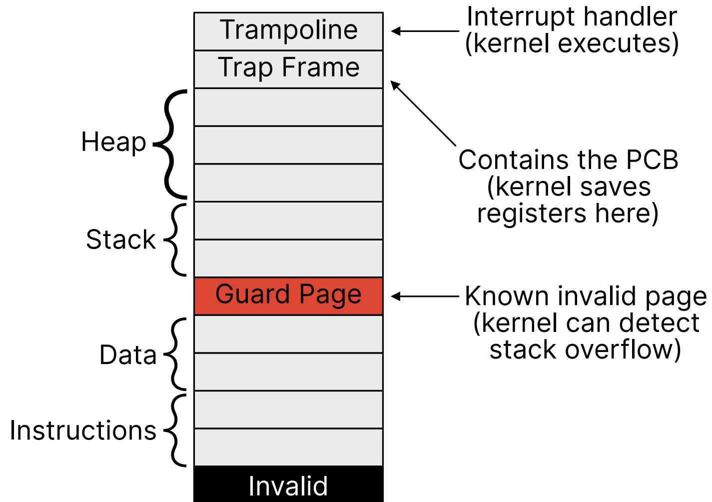
Your working set is the number of pages your program needs to execute a portion of instructions

If your working set is greater than your TLB reach

TLB misses occur, which may cause many memory accesses

If your working set is greater than the total amount of memory, pages swap to disk in and out continuously (thrashing)

The Typical Process Address Space Layout



The `sbrk` System Call Modifies the Heap

You'll be using a simulated version of this for Lab 3

This system call increments the heap by the specified number of bytes

This heap space is what `malloc` and friends use

You Can Also Map Memory Yourself

A more flexible system call is `mmap`

You can initialize virtual memory with the contents of a file

You can share memory across processes

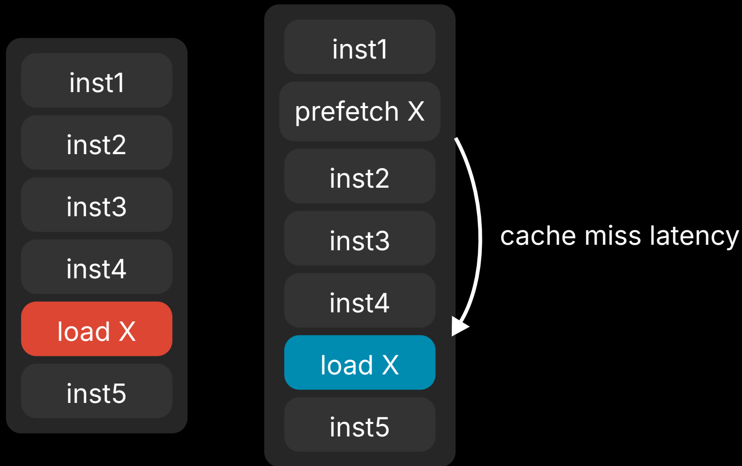
The Kernel Can Use Hugepages

A 39 bit virtual address space requires 3 levels of page tables,
most processors use 48 bits (4 levels)

One approach is to use the L1 page table to point to a 2 MiB page

You can also use the `madvise` system call to give hints about memory usage
`MADV_SEQUENTIAL` allows freeing pages after use
`MADV_HUGEPAGE` to enable huge pages for an address range

Prefetching Gives Hints on Memory Loads/Stores



If effective, it helps tolerate latency to memory

Prefetching is Difficult

It’s only effective if ALL the following is true:

There is spare memory bandwidth
- Otherwise prefetching can cause bandwidth bottleneck

Prefetching is accurate
- Only useful if the prefetched data will be used soon

Prefetching is timely
- i.e., prefetch the right data, but not enough in advance

Prefetched data doesn’t displace other in-use data
- e.g., prefetched data should not replace a cache block about to be used

Latency hidden by prefetching outweighs its cost
- Cost of lots of useless prefetched data can be significant

There’s Hardware Prefetching

A simple hardware prefetcher could fetch the adjacent block,
this makes it behave like cache blocks are twice as big
(this also helps with unaligned instructions)

A more complex prefetcher can recongize stride
stride is how many bytes are between addresses

For example, if you access a+0, a+128, a+256, the stride is 128
(prefetch addresses a+stride*j)

Real prefetchers are proprietary, you don’t know the details

You Can Add Prefetching Instructions with C Compilers

You can use: `__builtin_prefetch (const void *addr[, rw[, locality]])`

where,

- `addr` is the address to prefetch
- `rw` (optional)
  - `0` (default): read
  - `1`: write
- `locality` (optional)
  - `0` none
  - `1` low (leave in L3)
  - `2` medium (leave in L2)
  - `3` (default): high (leave in L1)

Example Using Binary Search

Check the `examples` directory and run
`./perf-wrapper.py all 1000000000 10000`

```
binsearch-base:
  Cycles per element: 4818.37
  L1d load miss rate: 30.38%
     Misses: 718_791
     Loads:  2_365_832
binsearch-prefetch:
  Cycles per element: 4132.07
  L1d load miss rate: 52.08%
     Misses: 763_769
     Loads:  1_466_656
```

Large Binary Search with Hugepages

Now enable hugepages and try running
`./perf-wrapper.py all 1000000000 10000`

```
    binsearch-base:
    Cycles per element: 2845.73
    L1d load miss rate: 50.29%
       Misses: 965_891
       Loads:  1_920_571
  binsearch-prefetch:
    Cycles per element: 2189.47
    L1d load miss rate: 90.98%
       Misses: 923_175
       Loads:  1_014_736
```

Takeaways

For cache, focus on either L1 or L2 depending on required working-set size
- try to make sure your working set can fit in cache

For virtual memory, keep your working set on as few pages as possible
- TLB misses will slow down your program

For prefetching, make sure your accesses are sequential/strided,
- if you know the next access that may seem random, use the compiler