```
6.S081 2020 Lecture 17: User-level virtual memory

Reading: VM primitives for user programs by Appel & Li (1991)

Plan:
  OS kernel uses virtual memory in creative ways
  Paper argues user-level application can also benefit from VM
    Concurrent garbage collector
    Generation garbage collector
    Concurrent check-pointing
    Data-compression paging
    Persistent stores
  Most OSes have mmap() and user-level pagefaults

What VM user-level VM primitives?
  Trap: handle page-fault traps in user mode
  Prot1: decrease the accessibility of a page
  ProtN: decrease the accessibility of N pages
  Unprot: increase the accessibility of a page
  Dirty: return a list of dirtied pages since previous calls
  Map2: map the same physical page at two different VAs with
    different prot levels.

Xv6 supports none of them
  One way to see the paper is that a good OS should support them
  What about Unix today?

Unix today: mmap()
  Maps memory into the address space (many flags and options)
  Example: mapping file
    mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, fd, offset)
  Examples: anonymous memory
    mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
    (preferred over sbrk)

Unix today: mprotect()
  Changes the permission of a mapping
  Examples:
    mprotect(addr, len, PROT_READ)
    mprotect(addr, len, PROT_NONE)
      results in a trap on access

Unix today: munmap()
  Removes a mapping
  Example:
    munmap(addr, len)

Unix today: sigaction()
  Configures a signal handler
    (Think sigalarm() lab)
  act.sa_sigaction = handle_sigsegv;
  act.sa_flags = SA_SIGINFO;
  segemptyset(&act.sa_mask)
  sigaction(SIGSEGV, &act, NULL);

Additional Linux VM calls
  Madvise(), Mincore(), Mremap(), Msync(), Mlock(), Mbind(), Shmat(),..

VM Implementation
  Address spaces consists of VMAs and page tables
  VMA (virtual memory area)
    contiguous range of virtual addresses
    same permissions
    backed by the same object (file, anonymous memory)
```

```
    VMA helps the kernel to decide how to handle page faults

  Trap/sigaction implementation
    PTE (or TLB entry) marked "protected"
    CPU saves user state, jumps into kernel.
    Kernel asks VM system what to do?
      I.e. page in from disk? Core dump?
      VM system looks at VMA
    Generate signal -- upcall into user process.
      Lower on user stack, or on separate stack...
    Run user handler, can do anything.
      Probably must call UNPROT for referenced page.
      That is, must avoid repeated fault.
    User handler returns to kernel.
    Kernel returns to user program.
    Continue or re-start instruction that trapped.

  Can we support user-level VM primitives?
    Trap: sigaction and SIGSEGV
    Prot1: mprotect()
    ProtN: mprotect()
    Unprot: mprotect()
    Dirty: No, but workaround exists
    Map2: Not directly, but shm_open/mmap/mmap

  Demo: large sqrt table backed by a single page
    Application code thinks there is a pre-computed big table
      n -> sqrt(n)
    Application can lookup sqrt:  sqrts[n]
      If present, very fast!
    Table is bigger than physical memory
    User-level VM primitives allow it to occupy only a *single* page

  Use case: Concurrent GC
    Application allocates memory and computes with it
      Application is called the "mutator"
      Application doesn't have to call free
      (which is error prone, in particular in concurrent programs)
    Collector concurrently finds free memory
      Memory that isn't in use by application anymore
      Not reachable from root pointer or registers of any thread
    Traditional implementation requires language/compiler support
      Instrument loads/stores
    User-level virtual memory can avoid instrumenting load/stores
      Faster

  Example: Baker's real-time GC algorithm  (see toy version in baker.c)
    https://dl.acm.org/citation.cfm?id=359460.359470
    Divide heap into 2 regions: from-space and to-space
      to-space is further divided in: scanned, unscanned, and new
    At start of collection: all objects are in from-space
      Copy roots to to-space (register and stack)
      Put forwarding pointers in from-space to the to-space copy
    At the end of scan, from-space is free memory
    Don't have to stop the world for GC
      Do a little of scanning on each allocation
      Or when dereferencing pointer in from-space

  Observations
    Why attractive?
      Alloc is cheap. Compacts, so no free list.
      Incremental: every allocation scan/copy a little
      This is the "real-time" aspect
    What are the cost?
      Does pointer reside in the from space? (If so, it needs to be copied)
```

```
      Requires test and branch for every dereference
    Difficult to run collector and program at same time
      Race condition between collector tracing heap and program threads
      Risk: two copies of the same object

Solution: let application use VM
  https://dl.acm.org/doi/10.1145/53990.53992
  Avoid explicit checks for references in application threads
    After copying roots, unmap the unscanned part of to-space
      Initially a page with roots etc.
    Page fault when thread accesses unscanned region
      handlers scans just the page and inspects all objects, then UNPROT
      at most one fault per page
      no compiler changes
  Easy to make concurrent
    A collector thread can run concurrently with application thread
      A collector thread can UNPROT a page after scanning
    Only synchronization needed is for which thread is scanning which page

Are existing VM primitives good enough for concurrent GC?
  MAP2 is the only functionality issue -- but not really.
    We never have to make the same page accessible twice!
  Are traps &c fast enough?
    They say no: 500 us to scan a page, 1200 us to take the trap.
    Why not scan 3 pages?
    How much slower to run Baker's actual algorithm, w/ checks?
      VM version might be faster! Even w/ slow traps.
    What about time saved by 2nd CPU scanning? They don't count this.
  Is it an issue how often faults occur for concurrent GC?
    Not really -- more faults means more scanning.
    I.e. we'll get <= one fault per page, at most.

Is user-level VM a good idea?
  Most of the use cases can be implemented by adding instructions
    E.g., check in application thread on reference
  Pro:
    Avoid compiler changes
    CPU provides VM support
  Con:
    Requires OS support, and efficient support
    Most OS kernel can't expose the raw hardware performance of paging
      OS imposes much abstraction

What has changed between 1991 and 2020?
  Tons of changes to VM API and implementations
    VM system evolves continuously
    Continued research too (e.g., see OSDI 2020)
  Switching address space is free (tagged TLBs)
  Extended addressibility doesn't matter
    2^52 bytes of virtual address space
```