CS354 Midterm 2

- Posix brk & unistd.h. — contains collection of Posix API functions

DIY Heap using Posix calls.

Portable OS interface: Standard for maintaining compatability among Unix OS's

brk: points to end of program (shifts up if need more space). points at top of Heap.

(1) int brk(void *addr) : sets top of heap to specified addr (0 if successful, else -1 & sets errno).
- not common to use.

(2) void *sbrk(intptr_t incr) intptr_t is size of long for ptr addr.
- attempts to change program's top of heap by incr bytes.
- returns old brk if successful, else -1 & sets errno.

(3) errno #include <errno.h>. Strerror(errno) to convert to string, only 1 error at a time, overwritten if new error.

(4) Best to use malloc/calloc/realloc/free, since well-tested, cannot use combination ⇒ UB undefined program behaviour.

stdlib.h: 25 commonly used functions. int 4 bytes long 8.

- conversion: atoi, stol (string to long)
- execution flow: skip to exit, exit(1)
- math funcs: absolute, floor, ceiling
- search: linear, binary
- sorting: qsort
- random number: rand.

Heap Allocator Functions

void * malloc(size_t size) - allocates size (in bytes), returns pointer to block of heap memory, NULL if fail.

void * calloc(size_t nItems, size_t size) - Allocates, clears to 0, returns block of memory of nItems * size bytes, return NULL if allocation fails.

void *realloc(void *ptr, size_t size) - reallocates to size bytes a prev allocated block of heap mem pointed to by ptr, return NULL if fail.
- if size smaller than current ⇒ no problem, else make block bigger w/o interfering w other blocks (uses mallo

void free(void *ptr) — free heap mem pointed to by ptr, does nothing if ptr is NULL.

Heap

- segment of Virtual Address Space used for dynamically allocated memory → requested while program is running.
- is a collection of various sized memory blocks.
  - ↳ block : contains payload, overhead, allocator.
  - ↳ payload : part requested & used by user
  - ↳ overhead : part used by allocator to manage allocation
  - ↳ allocator : code that allocates & frees.
- C uses explicit approach : tell how many bytes, explicitly call free to free up memory
  - vs Java "new" calculates bytes needed, "garbage collector", free mem when needed.

- Allocator Design Goals.
  1. Maximize Throughput – request bytes in certain order, just going to allocate the space
     - ↳ # of malloc & frees per time interval, higher better.
  2. Maximize Memory Utilization – % of memory used, that is used for payload.
     - ↳ memory requested (bytes) / heap that has been allocated (bytes).

  } Trade off: improve one, worsens other.

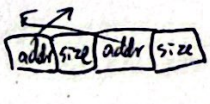
- Double Word Alignment
  ① Block sizes multiple of 8 ② Payload addr at multiple of 8.

- Fragmentation :
  ① External – enough free heap mem divided into blocks that are each too small.
  - happens due to sequence of calls.

  ② Internal – allocated memory used for overhead instead of payload.
  - block padding, block headers causes this.

  ③ False – when combined memory in adjacent free blocks sufficient to satisfy request but individually not.

- Explicit Free List (EFL) : allocator maintains data structure with free blocks.
  -ve : may need more memory for data structs.
  +ve : time to search only free block for size.



- Implicit Free List : look all blocks instead of just list of free blocks, allocator uses heap blocks as data structure.
  - code must track size & status of each block.
  -ve : time, more time to search both alloc'd & free blocks
  +ve : may be less memory.

- Placement Policies : Assuming heap pre-divided into various size free blocks smallest to largest
  ① First Fit : start from beginning, stop at first free block that is big enough, fail if reach END MARK.
  ~~mem~~ memory utilization : -ve, likely chooses desired size thruput : -ve, must step through many alloc'd blocks.
  ② Next Fit : start from block most recently alloc'd, stop at first free block that is big enough, fail if reach block started with (wrap around)
  mem util : not as good, may choose too big a block thruput : faster, skip many prev alloc'd blocks.
  ③ Best Fit : start from beginning, stop END MARK & choose best fit or early if EXACT FIT, fail if no block big enough.
  mem util : best, closest to size of request thruput : slowest in general.

- Free Block
  - If bigger than request
    - ① Use entire block
      - ↳ memutil : -ve, ↑ internal fragmentation — waste space.
      - ↳ thruput : tve, faster code.
    - ② Split into used + free blocks
      - ↳ memutil : tve, less wasted space.
      - ↳ thruput : -ve, more operations, slower to search, more blocks to search.
  - If request too large for any free block.
    - ① coalesce adjacent free blocks. (cannot move already alloc'd blocks to create larger free areas).
    - ② ask kernel for more.
    - ③ return NULL, alloc fail.

- Coalescing Free Blocks. (constant time).
  - immediate : coalesce after freeing block — check prev & next, coalesce if possible.
  - delayed : only if needed to satisfy request — in alloc func

- Free Block Footers
  - last word of free block (4 bytes) contain free block size, alloc'd blocks no need since cannot be freed.
  - from payload to prev block HDR : (ptr-4) ⇒ check p-bit, if 0, (ptr-4)-prev_block_size
    - * take note scaling (void*) or (char*) for 1.

- Explicit Free List Layout

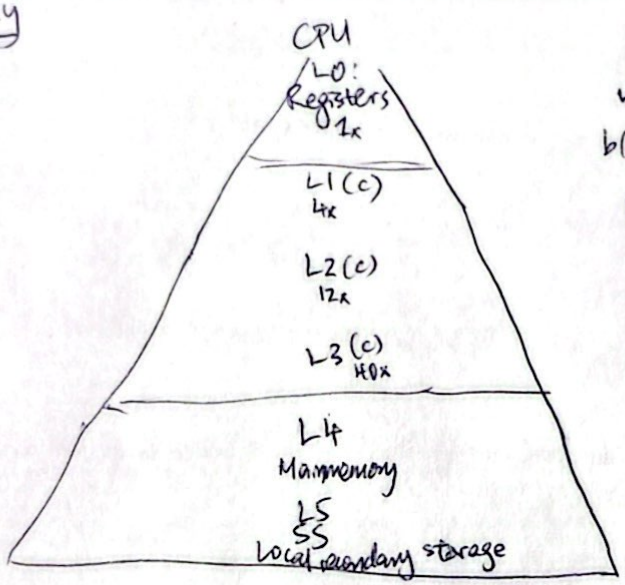
* Footer still useful for faster coalescing.

* order of free blocks in free list don't have to be in same order as found in addr space.

Improvements: — Free list Ordering
- ① address order : maintain list in order from low to higher addr.
  - malloc with FF : easier to find free block. tve
  - free : slower -ve
- ② last-in order : append most recently freed block at end of list.
  - malloc with FF : slower, must go through most recent
  - free : faster. not trying maintain order.

— Free List Segregation
- ① simple segregation : one EFL for each block size
- ② fitted segregation : one EFL for each size range — small, medium, large.
  - tve memutil — as good as best fit
  - tve thruput — search only part of the heap.

Memory Hierarchy



word : size btwn L1 & CPU (c)
block : transfer btwn C levels & MM
page : sized used by MM
trf between MM & SS.

- cpu cycles: used to measure time
- latency : memory access time (delay)

Temporal Locality : recently used location repeatedly used in future.
Spatial Locality : recently used location is followed by access of nearby location. (on the same pallet).
Stride : Step size in words between sequential access in the block. Smaller better.
Cache Block : unit of transfer between MM & C levels, typically 32 bytes.

Good or Bad Locality?.
Instruction Flow:
- sequencing : good spatial
- selection : if ... else ... bad spatial, bad temporal.
- repetition : good temporal, good spatial if stride is small.

Searching Algorithms:
- Linear search : good spatial if linear array, bad array if linked list/tree } some temporal.
- Binary search : bad spatial

Cache Misses: (block not in cache)
① Cold miss : cache location empty ② Capacity Miss : cache too small for working se
③ conflict miss : 2 or more blocks map to same location.

Cache Hit : Block in cache.
Placement Policies : ① unrestricted — can go anywhere (L1) ② Restricted — mapped to set of locations (L2).
usually block num % 16.
Replacement Policies : ① Choose any block location (L1) ② Restricted — block # % 16 (L2).
Victim Block : Block to be replaced
Working set : set of blocks accessed given some interval of time.

Bytes in Addr Space.

Let M = #bytes. (IA-32 4GB).

so $M = 2^m \Rightarrow m = \log_2 M$ = #bits in addr (32 in IA-32).

How Big is a block?

- Cache blocks should be big enough to capture spatial locality, small enough to minimize latency.

Let B = #bytes per block. (IA-32, 32 bytes per block).

$B = 2^b = 32$ bytes $\Rightarrow b = \log_2 B = 5$ (5 b bits).

b-bits: #bits of address to determine which byte of block.

word offset: identify which word of block.

byte offset: identify which byte of word (last 2 bits). since 4 byte/word:

b-bits on most right. If in most left side $\Rightarrow$ LOSE SPATIAL LOCALITY.

- all bytes near each other won't be in same block.

How many 32-byte blocks in 32-bit addr space?

$$VAS/B = 2^{32}/2^5 = 2^{27} = 2^7\,2^{20} = 128\,Mb.$$

For 32 byte blocks, if last 5 bits are 01101 use byte 1 of word 3.

(3rd word, 1 byte)

Sets & Tags

set: where a block of memory is uniquely mapped, maps to specific set in cache.

tag: uniquely identifies particular block in set.

S = # sets in cache $= 2^s \Rightarrow$ s = #bits for sets. s bits $\Rightarrow$ identifies which set block maps to.

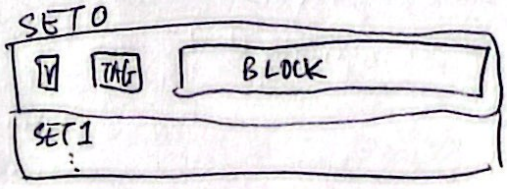
why s bits not most left $\Rightarrow$ don't want consecutive blocks in same set, might displace each other
will lose spatial locality.

How many blocks map to each set for 32-bit As and a cache with 1024 sets?

$$(VAS/S)/B = (2^{32}/2^{10})/2^5 = 2^{17} = 2^7\,2^{10} = 128\,KBc$$

t-bits: bits to identify which block in set. * when block is copied into cache, t bits stored as well

v-bit: indicate if cache line is in use or not



Request for word

1. Set selection: using s bits (gets set)
2. Line Matching: using t-bits match tag, v-bit must be valid too. (gets line).

For L1 cache only (highest level), must extract word from Block as well

Direct Mapped Cache: cache with s sets, 1 line per set.

mem blocks map to exactly 1 set.

cache operations: • no search, set selection $O(1)$, line matching $O(1)$
• simple circuitry.

when 2 mem blocks map to same set ⇒ CONFLICT MISS, same s bits, diff t bits so diff block num but since only 1 line/set, only 1 block can be stored.

* can cause thrashing: continuously exchange blocks.

Appropriate for larger caches.

Fully Associative Cache: cache with 1 set, E lines in that set.

mem blocks can be at any line in that set.

cache operations: • set selection $O(1)$, line matching $O(E)$ where E is # of lines.
• complex circuitry since every block matched to every line, many bits to match.

when 2 diff blocks map to same set ⇒ all map to same set ⇒ choose free line.

How many lines? $C = S \times E \times B$, $S=1$, so $E = \frac{C}{B}$.

Appropriate for smaller caches (L1).

Set Associative Cache: S sets with E lines per set.

mem blocks map to a single set with E lines (can be any line in that set).

cache operations: set selection $O(1)$, line matching $O(E)$ E smaller than Fully Assoc.
compute from s bits

+ve: ↓ conflict miss, less circuitry than fully assoc. -ve: more circuitry than direct mapped

$E=4$ ⇒ 4-way associative

$C = (S, E, B, m)$ (m = # bits in addr)

$C = S \times E \times B$. For $(1024, 4, 32, 32)$

$C = 2^{10} \times 2^2 \times 2^5 = 2^{17}$ with 15 t bits.

Replacement Policies

① Random Replacement: last block must be in cache, usually no duplicates & empty lines.

② Least Recently Used: use queue keep track.

③ Least Frequently Used: track how often each line is used, each line has a count.
• set to 0 if line gets new block.
• increment when line is accessed
• if tie, evict randomly chosen of ties.

Writing to a cache

- Reading data copies a block of memory into cache levels.
- Writing data requires these copies are consistent.

Write Hits: writing to a block that is in the cache (k)

1. Write Through: immediate update lower level (k+1)
   - -ve: need wait for lower level to write, ↑ time
   - -ve: more bus traffic, passing all writes through
   - +ve: or add bypass cache with write buffer (smaller side cache to let it wait for k+1 to do write).
2. Write Back: write to next level (k+1) when block evicted from K (line changes)
   - +ve: faster, no wait
   - +ve: less bus traffic, only update when done w block.
   - -ve: must track if line has been changed add "DIRTY" bit

Write Misses: when writing to a block that is <u>not</u> in cache (k).

1. NO Write Allocate: write directly to next lower level (k+1) bypass cache k.
   - -ve: must wait for next lower level to write (k+1)
   - +ve: less bus traffic since no wait to get block.
2. Write Allocate: Read into curr level K first, then write
   - -ve: must wait for block to be read in
   - -ve: more bus traffic, must pass block to current cache level.

Typical Designs: ① Write through with No Write Allocate "get to k+1"

② Write Back with Write Allocate "keep at K"

② exploits locality best, keeps in highest level before having to go lower level.

Cache Performance

Metrics: ① Hit rate: # hits / # mem access, higher better.

② Hit time: time to determine cache hit, set selection + line matching, lower better.

③ Miss Penalty: additional time to process a miss, lower better.

Larger Blocks (S & E unchanged) $C = S \times E \times B$

- hit rate: better, more spatial locality per block, eg more values from array in same block.
- hit time: same, since set selection & line matching same
- miss penalty: worse, ↑ bus traffic when transferring block.
- Thus block sizes small, usually 32/64 bytes.

More Sets (E & B unchanged)

- hit rate: better, ↑ temporal locality, more sets so diff blocks can be in cache at the same time.
- hit time: worse, more sets, more complexity (circuitry) to find correct matching set (still O(1))
- miss penalty: same, since block size same
- Thus, faster caches smaller (L1), slower caches larger.

More Lines E per set (S & B unchanged)

- hit rate: better, more temporal locality, more lines, more diff blocks in cache at same time, fewer conflict misses.
- hit time: worse, slower line matching O(E)
- miss penalty: ~~same~~ worse, more lines, longer to detect match. Therefore faster caches fewer lines (L