Exercise 07

VU Performance-oriented Computing, Summer Semester 2024

Calvin Hoy

2024-05-14

As usual, all job scripts may be found in the `jobs/` directory, all raw data and figures in the `preload/` and `bumpmalloc/` directories.

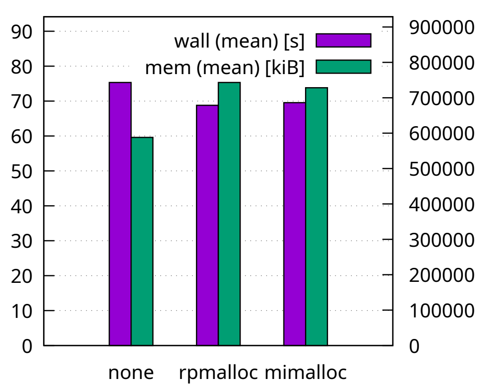
(A) Preloading General Allocators

To test the performance when compiling the `allscale_api` project at hand when (a) using the default memory allocator, (b) preloading RPMalloc, and (c) preloading MIMalloc, I created one job script each which first compiles the respective allocator (or does nothing) and sets the `LD_PRELOAD` flag appropriately.

Before each benchmark pass, `ninja clean` must be run as otherwise `ninja` will not build. To this end, the `benchmark.sh` script first used in exercise 2 was modified slightly to support a `prepare` command.

Further, RPMalloc in its current version uses a warning flag unknown to Clang v15, so its `build.ninja` had to be modified to ignore this.

Results



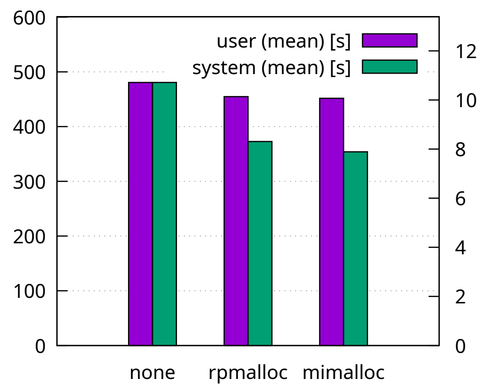


Figure 1: Wall time and memory usage/CPU user and system time using different allocators

`ninja` runs using the default allocator, `rpmalloc`, and `mimalloc` finish in 75.320, 68.792, and 69.548 seconds on average, respectively. `rpmalloc` is thus 8.7% faster than the default allocator, `mimalloc` 7.7%.

While `mimalloc` is technically slower, it does have a small edge in CPU time, spending 451.682 seconds in user space across all cores to `rpmalloc`'s 454.728 — 0.7% less. Compared to the default allocator, `rpmalloc` saves 5.1% user time, while `mimalloc` saves 5.7%.

Both custom allocators also spend significantly less time in kernel space than the default allocator does. The improvement is 22.4% with `rpmalloc` and 26.4% with `mimalloc`, meaning `mimalloc` has a slight edge (5.1%) here too.

(B) Implementing a special-purpose allocator

I chose to implement the arena bump allocator as a shared library, usable via `LD_PRELOAD`. It overrides the standard library's `malloc` and `free` functions. Please find the code in `bumpmalloc/bumpmalloc.c`. A `Makefile` is also provided.

There is one compile-time constant — `ARENA_SIZE` — that defines the size in bytes of the static `char arena[]` memory region used for memory allocation. By default it is set to $1000 \cdot 1000000$ bytes, accommodating one worst-case run (all 1,000,000 allocations use the maximum number of bytes, 1000) of the parameters we give to `tools/malloctest`.

The `malloc(size_t size)` function first checks if the requested size exceeds the arena size, and returns `NULL` if so. It then checks if there is enough space left in the arena to accomodate the requested allocation, and if not, resets the arena pointer to zero. Finally, it computes the memory address currently pointed to by `arena[arena_ptr]`, increases the arena pointer by `size` bytes, then returns the computed memory address.

The `free(void* ptr)` function does nothing; it only exists to override the standard library `free` function as it would crash the program.

Results

allocator	wall (mean) [s]	wall (variance)
default	230.172	1.246
bumpmalloc	12.147	0.000

The bump allocator improves benchmark performance by a factor of 18.95 over the default allocator. Further improvements could certainly be attained by directly integrating the allocator into the benchmark code.

allocator	system (mean) [s]	system (variance)
default	131.636	0.695
bumpmalloc	0.827	0.000

We see that using the default allocator, over half of the benchmark's runtime is spent in kernel space. This is to be expected, as there would be a lot of calls to `mmap` and `munmap` that are completely eliminated by the bump allocator.

allocator	mem (mean) [kiB]	mem (variance)
default	516894.4	1452.8
bumpmalloc	3909529.333	21.333

As expected, memory usage is higher due to the fixed, worst-case arena size — though, why it uses 4 gigabytes instead of 1 is not clear to me.