propag-4

MEMORY

Revision: 4.12 , April 24, 2009

When I was younger, I could remember anything, whether it had happened or not

—Mark Twain

1. Introduction. The functions defined in this document deal with several aspects of memory allocation. Services that they provide include checking for allocation failure, reporting to the logfile, memory protection, and gathering of statistics.

The `MALLOC` and `CALLOC` macros are the preferred interface for memory allocation. They take three arguments $(N, \mathbf{typ}, what)$, where N is the number of elements; **typ** is a literal giving the type of variable (such as **float**); and *what* is a string that will be used in the logfile message that is generated for each allocation. The messages can be suppressed (for an allocation in a long loop, for example) by giving Λ or an empty string (`""`) for the *what* argument. The *type* argument is used both to compute the amount of memory required and to typecast the pointer to the correct type, allowing the compiler to check for errors. Compilers cannot see anything wrong in **float** $*arr = calloc(4, \mathbf{sizeof}(\mathbf{short}))$, but they will bark at **float** $*arr =$ `CALLOC`$(4, \mathbf{short}, \texttt{"arr"})$.

The `TMALLOC` and `TCALLOC` macros do the same as `MALLOC` and `CALLOC`, but also touch the memory with the right processor. These are rarely used, because touching is often done implicitly by some useful operation.

The `SMALLOC` and `SCALLOC` macros do the same as `MALLOC` and `CALLOC`, but don't enter the pointer in the memory table. They do update the total memory counter, however. These are intended for large numbers of small allocations.

If a really large number of small allocations is used, say millions, even `SMALLOC` and `SCALLOC` should not be used. Using millions of allocations is not likely to be efficient on a multiprocessor machine. If it is necessary, it should be done directly with *malloc* or *calloc*.

The *Protect* function uses the memory table to enable memory protection for an array indicated only with its pointer, without the need to specify its size. Memory protection affects performance. It can be switched on or off with the *protect_memory* parameter.

When the above functions are used consistently, propag knows how much memory it uses. The *maxmem* parameter allows the user to set a maximum and prevent the program from over-allocating. This can be useful, for example, in an interactive PBS job, to make a graceful stop when the program accidentally allocates more then the limit imposed by PBS – instead of receiving a KILL signal.

2. This should normally be all that's needed in other modules. Remaining calls to the underlying functions should be replaced. The `*ALLOC` macros force a typecast to the return argument with the same type as used in **sizeof**, so the compiler will warn if we may be allocating with the wrong size, for example, when changing a typedef somewhere. Multiplications are done in **size_t** to rule out integer overflow.

⟨ declare interface 2 ⟩ ≡

```
#define MALLOC(N, typ, what)(typ *)Malloc ((size_t) (N) * sizeof(typ), what, 1)
#define CALLOC(N, typ, what)(typ *)Calloc ((size_t) (N), sizeof(typ), what, 1)
#define SMALLOC(N, typ, what) ((typ *) Malloc((size_t) (N) * sizeof(typ), what, 0))
#define SCALLOC(N, typ, what) ((typ *) Calloc((size_t) (N), sizeof(typ), what, 0))
#define TMALLOC(N, typ, what) ((typ *) Touch_Malloc((size_t) (N), sizeof(typ), what))
#define TCALLOC(N, typ, what) ((typ *) Touch_Calloc((size_t) (N), sizeof(typ), what))
  void Protect(void *addr, int mode);
  void report_memory( );
```

See also section 3.

This code is used in section 4.

3. These functions are also public, although they should be used only exceptionally.

⟨ declare interface 2 ⟩ +≡

```
  void *Malloc(size_t n, char *what, int tabit);
  void *Calloc(size_t n, size_t s, char *what, int tabit);
  void *Touch_Calloc(size_t n, size_t s, char *what);      /* see below */
  void *Touch_Malloc(size_t n, size_t s, char *what);      /* see below */
```

4. Here's the full header file. It includes `"mman.h"`, which applications will need for the values of the *mode* argument to *Protect*( ).

⟨ `propag_alloc.h` 4 ⟩ ≡

```
//␣$Id:␣memory.web,v␣4.12␣2009/04/24␣14:52:44␣potse␣Exp␣$␣
#include <sys/mman.h>      /* for mprotect-related things */
    ⟨ declare interface 2 ⟩
    float total_mem;      /* used in main program */
```

5. notes. ToDo: Try to use *valloc* to have all memory page-aligned, which is necessary for memory protection with *mprotect*(2). There is no *vcalloc*, so we implement memory cleaning here.

ToDo: Replace obsolete *valloc* with *posix_memalign*.

ToDo: Use *memtab* also to combine information with *dlook* output, to check memory placement for individual arrays.

Note: **size_t** is 8 bytes on large systems like the Altix, so the entire system memory can be allocated at once if desired. However, applications of *Malloc* should take care prepare their n argument in an 8-bytes type like **size_t** or **long** rather than a 4-byte **int** if chunks larger than 2 GB are required.

6. Housekeeping. This file implements the interface functions declared above, as well as some private functions.

```
⟨ propag_alloc.c 6 ⟩ ≡
  //␣$Id:␣memory.web,v␣4.12␣2009/04/24␣14:52:44␣potse␣Exp␣$␣
  ⟨ Preprocessor definitions ⟩  /* cweb defines are private */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  /* for sysconf */
#include <string.h>
#include "propag_alloc.h"
#include "propag.h"  /* get the protect_memory parameter and error functions */
  ⟨ function definitions 7 ⟩
```

7. This is a private function. For any expected memory usage, print at most four digits followed by a multiplier. It will print, for example, from "1G" upto "9999G," followed by "1T," etc. Presently, even "64-bits" machines have bus widths of only 48 bits, or other limitations on addressable memory, so we don't expect many exabytes for a while. Propag has run with upto 1.2T (exp268x11).

```
#define KILO 1024LL  /* 2^10 ≈ 10^3 */
#define MEGA (KILO * KILO)  /* 2^20 ≈ 10^6 */
#define GIGA (MEGA * KILO)  /* 2^30 ≈ 10^9 */
#define TERA (GIGA * KILO)  /* 2^40 ≈ 10^12 */
#define PETA (TERA * KILO)  /* 2^50 ≈ 10^15 */
#define EXA (PETA * KILO)  /* 2^60 ≈ 10^18 */
```

```
⟨ function definitions 7 ⟩ ≡
  static char *print_big_number(float x)
  {
    static char s[40];
    if (x ≥ 9995 * TERA) sprintf(s, "%.3g", x);
    else if (x ≥ 9995 * GIGA) sprintf(s, "%.0fT", x/TERA);
    else if (x ≥ 9995 * MEGA) sprintf(s, "%.0fG", x/GIGA);
    else if (x ≥ 9995 * KILO) sprintf(s, "%.0fM", x/MEGA);
    else if (x ≥ 9995) sprintf(s, "%.0fk", x/KILO);
    else sprintf(s, "%.0f", x);
    return s;
  }
```

See also sections 8, 9, 10, 13, 14, and 15.

This code is used in section 6.

8. This function is called a few times by the main program, e.g. at the end.

```
⟨ function definitions 7 ⟩ +≡
  void report_memory( )
  {
    logline("MEM", "allocated␣memory␣at␣this␣time␣is␣%s", print_big_number(total_mem));
  }
```

9. Memory-allocation functions. This is the main allocator. It checks whether the total memory will not surpass *maxmem* (if specified), checks for allocation failure, and stores information in the memory table. For memory protection, allocation areas must be aligned on memory pages (16 kB on the Altix 4700). This is done by *valloc*. Apparently the memory size for *valloc* must be a multiple of the page size too.

```
⟨ function definitions 7 ⟩ +≡
  ⟨ declare memory table 11 ⟩
  void *Malloc(size_t n, char *what, int tabit)
  {
    char *p;
    size_t pagesize, sz, n0, n1;
    if (maxmem > 0 ∧ total_mem + (float) n > maxmem * GIGA) {
      Error(1, "allocation␣of␣%ld␣bytes␣(%f␣MB)␣for␣%s␣will␣surpass␣maxmem=%fGB", (long)
          n, ((float) n)/MEGA, what ∧ strlen(what) ? what : "<anonymous>", maxmem);
    }
    n0 = (long) sbrk(0);
    if (protect_memory) {
      pagesize = sysconf(_SC_PAGESIZE);
      sz = pagesize * (n/pagesize + 1);
      p = valloc(sz);      /* code = posix_memalign(&p, pagesize, sz); */
      if (¬p) {
        Error(1, "allocation␣of␣%d␣bytes␣(%f␣MB)␣for␣%s␣failed", n, ((float) n)/MEGA,
            what ∧ strlen(what) ? what : "<anonymous>");
      }
    }
    else {
      sz = n;      /* need sz below */
      if (¬(p = malloc(n))) Error(1, "allocation␣of␣%ld␣bytes␣(%f␣MB)␣for␣%s␣failed", (long)
          n, ((float) n)/MEGA, what ∧ strlen(what) ? what : "<anonymous>");
    }
    n1 = (long) sbrk(0);
    if (n1 − (long) p < n) Error(1, "malloc␣fails:␣n=%ld␣space=%d\n", (long) n, n1 − (long) p);
    if (tabit) ⟨ store in memory table 12 ⟩
    total_mem += (float) n;
    if (what ∧ strlen(what))
      logline("MEM", "allocated␣%ld-->%ld␣bytes␣for␣%s␣at␣%p␣␣total=%s", (long) n, (long)
          sz, what, p, print_big_number(total_mem));
    return p;
  }
```

10. This is the equivalent of *calloc*. It uses *Malloc* to do the allocation itself.

```
⟨ function definitions 7 ⟩ +≡
  void *Calloc(size_t n, size_t s, char *what, int tabit)
  {
    void *p;
    char *q, *e;
    p = Malloc(n * s, what, tabit);
    e = (char *) p + n * s;
    for (q = (char *) p; q < e; q++) *q = 0;
    return p;
  }
```

11. The memory table. Information about each allocation is stored in the *memtab* array. This array is used by the *Protect* function to obtain the size of the area to be protected.

```
⟨ declare memory table 11 ⟩ ≡
  typedef struct {
    void *address;
    long size;
    char *name;
  } Memtab;
  Memtab *memtab = Λ;
  int Nalloc = 0, memtab_size = 0;
```

This code is used in section 9.

12. Reallocation of the memory table is worth a warning because it may indicate that *Malloc*( ) is called in a long loop somewhere. Such allocations should use the `SMALLOC` or `SCALLOC` macros, which call *Malloc* with *tabit* ≡ 0, instead of `MALLOC` and `CALLOC`.

```
⟨ store in memory table 12 ⟩ ≡
  {
    if (Nalloc ≡ 0) {
      memtab_size = 4096;
      memtab = malloc(memtab_size * sizeof(Memtab));
      logline("MEM", "Nalloc=%d␣;␣allocated␣memtab␣to␣%ld␣bytes,␣memtab=%p", Nalloc,
          memtab_size * sizeof(Memtab), memtab);
    }
    if (Nalloc ≥ memtab_size) {
      memtab_size *= 2;
      Warning(26, "Nalloc=%d␣;␣reallocating␣memtab␣to␣%d␣bytes\n", Nalloc,
          memtab_size * sizeof(Memtab));
      memtab = realloc((void *) memtab, (unsigned long) (memtab_size * sizeof(Memtab)));
    }
    memtab[Nalloc].address = p;
    memtab[Nalloc].size = n;
    if (what ∧ strlen(what)) memtab[Nalloc].name = strdup(what);
    else memtab[Nalloc].name = strdup("(anonymous)");
    Nalloc++;
  }
```

This code is used in section 9.

13. Memory protection. This is an easier alternative for the *mprotect* function; instead of requiring us to give the size of the area to be protected it obtains this information from the memory table.

If the address isn't found in the table, that probably means the memory was allocated through `SMALLOC` or `SCALLOC`. That is probably no catastrophe, but it merits a warning.

```
⟨ function definitions 7 ⟩ +≡
  void Protect(void *addr, int mode)
  {
    int i, code;
    if (protect_memory) {
      for (i = 0; i < Nalloc; i++) {
        if (memtab[i].address ≡ addr) {
          code = mprotect(addr, memtab[i].size, mode);
          if (code) Error(1, "memory␣protection␣failed␣for␣%s", memtab[i].name);
          if (0) logline("MEM", "set␣memory␣protection␣for␣%p␣to␣%d", addr, mode);
          return;
        }
      }
      Warning(24, "address␣%p␣not␣found␣in␣memtab:␣cannot␣protect␣memory", addr);
    }
  }
```

14. Memory touching. In order to make the program run efficiently on a multiprocessor machine with decentral memory but a single memory image, we must suggest to the machine that the memory will be located near to the processor that will use it most. On the SGI Origin system, this can be done by 'touching' the memory with the right processor. To make that possible, we try always to divide the model cells over the processors in the same way. Memory touching is then done in the same way. The *Touch_Calloc* function provides a shorthand for this operation. In some cases it is more efficient to separate allocation and touching, e.g. when several arrays of the same size must be touched, or if they must be initialized with special values. In those cases the normal *Malloc* will be used.

```
⟨ function definitions 7 ⟩ +≡
  void *Touch_Calloc(size_t n, size_t s, char *what)
  {
    char *p;
    long i, len = n * s;      /* OpenMP does not allow size_t for i */
    int tabit = 1;      /* hack */
    long t0, t1;
    float rate;
    p = (char *) Malloc(len, what, tabit);
    t0 = millitime( );
#pragma omp parallel for schedule(static) private(i)
    for (i = 0; i < len; i++) p[i] = 0;
    t1 = millitime( );      /* time in milliseconds */
    if (what ∧ strlen(what)) {
      rate = (float) len/(0.001 * (t1 - t0) * GIGA);
      logline("TCH", "Touch_Calloc touched %s at %.3f GB/s", what, rate);
    }
    return (void *) p;
  }
```

15. This function doesn't touch more than necessary. Since we don't know the size of the elements, we handle it as an array of **char** and touch one byte in each memory page.

However, A test on the Altix 4700 (with propag3) showed that this takes exactly the same amount of time as *Touch_Calloc* (0.2 seconds for 788 MB on 120 cores). Page size is 16 kB on this system. If the variable *pagesize* is **int** instead of **long**, it takes twice as long. It seems to take longer with more processors.

```
⟨ function definitions 7 ⟩ +≡
  void *Touch_Malloc(size_t n, size_t s, char *what)
  {
    char *p;
    long i, len = n * s;      /* OpenMP does not allow size_t for i */
    long pagesize, t0, t1;
    float rate;
    int tabit = 1;      /* always true here */
    pagesize = sysconf(_SC_PAGESIZE);
    logline("TCH", "pagesize = %dk", (int) (pagesize/1024));
    p = (char *) Malloc(len, what, tabit);
    t0 = millitime( );
#pragma omp parallel for schedule(static) private(i)
    for (i = 0; i < len; i += pagesize) p[i] = 0;
    t1 = millitime( );      /* time in milliseconds */
    if (what ∧ strlen(what)) {
      rate = (float) len/(0.001 * (t1 - t0) * GIGA);
      logline("TCH", "Touch_Malloc touched %s at %.3f GB/s", what, rate);
    }
    return (void *) p;
  }
```

16. Index.

⟨ declare interface 2, 3 ⟩ Used in section 4.
⟨ declare memory table 11 ⟩ Used in section 9.
⟨ function definitions 7, 8, 9, 10, 13, 14, 15 ⟩ Used in section 6.
⟨ `propag_alloc.c` 6 ⟩
⟨ `propag_alloc.h` 4 ⟩
⟨ store in memory table 12 ⟩ Used in section 9.