UMAP Documentation

Release 0.0.3

Marty McFadden

Jan 07, 2021

BASICS

Umap is a library that provides an mmap()-like interface to a simple, user- space page fault handler based on the userfaultfd Linux feature (starting with 4.3 linux kernel). The use case is to have an application specific buffer of pages cached from a large file, i.e. out-of-core execution using memory map.

- Take a look at our Getting Started guide for all you need to get up and running with umap.
- If you are looking for developer documentation on a particular function, check out the code documentation.
- Want to contribute? Take a look at our developer and contribution guides.

Any questions? File an issue on GitHub.

CHAPTER

ONE

GETTING STARTED

This page provides information on how to quickly get up and running with umap.

1.1 Dependencies

At a minimum, cmake 3.5.1 or greater is required for building umap.

1.1.1 UMAP Build and Installation

The following lines should get you up and running:

```
$ git clone https://github.com/LLNL/umap.git
$ mkdir build && cd build
$ cmake -DCMAKE_INSTALL_PREFIX="<Place to install umap>" ../umap
$ make
$ make install
```

By default, umap will build a Release type build and will use the system defined directories for installation. To specify different build types or specify alternate installation paths, see the *Advanced Configuration*.

Umap install files to the `lib`, `include` and `bin` directories of the `CMAKE_INSTALL_PREFIX`.

1.1.2 Basic Usage

The interface to umap mirrors that of mmap(2) as shown:

```
  void* base_addr = umap(NULL, totalbytes, PROT_READ|PROT_WRITE, UMAP_PRIVATE, fd, 0);
  if ( base_addr == UMAP_FAILED ) {
    int eno = errno;
    std::cerr << "Failed to umap " << fname << ": " << strerror(eno) << std::endl;
    return;
  }
```

The following code is a simple example of how one may use umap:

```
//////////////////////////////////////////////////////////////////////////////
// Copyright 2017-2020 Lawrence Livermore National Security, LLC and other
// UMAP Project Developers. See the top-level LICENSE file for details.
//
// SPDX-License-Identifier: LGPL-2.1-only
```

(continues on next page)

(continued from previous page)

```
//////////////////////////////////////////////////////////////////////////////

/*
 * It is a simple example showing how an application may map to a a file,
 * Initialize the file with data, sort the data, then verify that sort worked
 * correctly.
 */
#include <iostream>
#include <parallel/algorithm>
#include <fcntl.h>
#include <omp.h>
#include <cstdio>
#include <cstring>
#include <vector>
#include "errno.h"
#include "umap/umap.h"

void
initialize_and_sort_file( const char* fname, uint64_t arraysize, uint64_t totalbytes,␣
↪uint64_t psize )
{
  if ( unlink(fname) ) {
    int eno = errno;
    if ( eno != ENOENT ) {
      std::cerr << "Failed to unlink " << fname << ": "
        << strerror(eno) << " Errno=" << eno << std::endl;
    }
  }

  int fd = open(fname, O_RDWR | O_LARGEFILE | O_DIRECT | O_CREAT, S_IRUSR | S_IWUSR);
  if ( fd == -1 ) {
    int eno = errno;
    std::cerr << "Failed to create " << fname << ": " << strerror(eno) << std::endl;
    return;
  }

  // If we are initializing, attempt to pre-allocate disk space for the file.
  try {
    int x;
    if ( ( x = posix_fallocate(fd, 0, totalbytes) != 0 ) ) {
      int eno = errno;
      std::cerr << "Failed to pre-allocate " << fname << ": " << strerror(eno) <<␣
↪std::endl;
      return;
    }
  } catch(const std::exception& e) {
    std::cerr << "posix_fallocate: " << e.what() << std::endl;
    return;
  } catch(...) {
    int eno = errno;
    std::cerr << "Failed to pre-allocate " << fname << ": " << strerror(eno) <<␣
↪std::endl;
    return;
  }

  void* base_addr = umap(NULL, totalbytes, PROT_READ|PROT_WRITE, UMAP_PRIVATE, fd, 0);
  if ( base_addr == UMAP_FAILED ) {
```

(continues on next page)

(continued from previous page)

```
    int eno = errno;
    std::cerr << "Failed to umap " << fname << ": " << strerror(eno) << std::endl;
    return;
  }

  std::vector<umap_prefetch_item> pfi;
  char* base = (char*)base_addr;
  uint64_t PagesInTest = totalbytes / psize;

  std::cout << "Prefetching Pages\n";
  for ( int i{0}; i < PagesInTest; ++i) {
    umap_prefetch_item x = { .page_base_addr = &base[i * psize] };
    pfi.push_back(x);
  };
  umap_prefetch(PagesInTest, &pfi[0]);

  uint64_t *arr = (uint64_t *) base_addr;

  std::cout << "Initializing Array\n";

#pragma omp parallel for
  for(uint64_t i=0; i < arraysize; ++i)
    arr[i] = (uint64_t) (arraysize - i);

  std::cout << "Sorting Data\n";
  __gnu_parallel::sort(arr, &arr[arraysize], std::less<uint64_t>(), __gnu_
↪parallel::quicksort_tag());


  if (uunmap(base_addr, totalbytes) < 0) {
    int eno = errno;
    std::cerr << "Failed to uumap " << fname << ": " << strerror(eno) << std::endl;
    return;
  }
  close(fd);
}

void
verify_sortfile( const char* fname, uint64_t arraysize, uint64_t totalbytes )
{
  int fd = open(fname, O_RDWR | O_LARGEFILE | O_DIRECT, S_IRUSR | S_IWUSR);
  if ( fd == -1 ) {
    std::cerr << "Failed to create " << fname << std::endl;
    return;
  }

  void* base_addr = umap(NULL, totalbytes, PROT_READ|PROT_WRITE, UMAP_PRIVATE, fd, 0);
  if ( base_addr == UMAP_FAILED ) {
    std::cerr << "umap failed\n";
    return;
  }
  uint64_t *arr = (uint64_t *) base_addr;

  std::cout << "Verifying Data\n";

#pragma omp parallel for
  for(uint64_t i = 0; i < arraysize; ++i)
```

(continues on next page)

(continued from previous page)

```
    if (arr[i] != (i+1)) {
      std::cerr << "Data miscompare\n";
      i = arraysize;
    }

  if (uunmap(base_addr, totalbytes) < 0) {
    std::cerr << "uunamp failed\n";
    return;
  }
  std::cout << "Data is verified. uunmap done.\n";

  close(fd);
}

int
main(int argc, char **argv)
{
  const char* filename = argv[1];

  // Optional: Make umap's pages size double the default system page size
  //
  // Use UMAP_PAGESIZE environment variable to set page size for umap
  //
  uint64_t psize = umapcfg_get_umap_page_size();

  const uint64_t pagesInTest = 64;
  const uint64_t elemPerPage = psize / sizeof(uint64_t);

  const uint64_t arraySize = elemPerPage * pagesInTest;
  const uint64_t totalBytes = arraySize * sizeof(uint64_t);

  // Optional: Set umap's buffer to half the number of pages we need so that
  //           we may simulate an out-of-core experience
  //
  // Use UMAP_BUFSIZE environment variable to set number of pages in buffer
  //
  initialize_and_sort_file(filename, arraySize, totalBytes, psize);
  verify_sortfile(filename, arraySize, totalBytes);
  return 0;
}
```

CHAPTER

TWO

ADVANCED CONFIGURATION

Listed below are the umap-specific options which may be used when configuring your build directory with cmake. Some CMake-specific options have also been added to show how to make additional changes to the build configuration.

```
cmake -DENABLE_LOGGING=Off
```

Here is a summary of the configuration options, their default value, and meaning:

These arguments are explained in more detail below:

- `ENABLE_LOGGING` This option enables usage of Logging services for umap. When this support is enabled, you may cause umap library to emit log files by setting the `UMAP_LOG_LEVEL` environment variable to "INFO" (for information-only logs), "WARNING" (for warning info logs), "ERROR" for (for errors only logs), and "DEBUG" for all debug messages to be emitted to a log file.
- `ENABLE_DISPLAY_STATS` When this option is turned on, the umap library will display its runtime statistics before unmap() completes.
- `ENABLE_TESTS` This option enables the compilation of the programs under the tests directory of the umap source code.
- `ENABLE_TESTS_LINK_STATIC_UMAP` This option enables the compilation of the programs under the tests directory of the umap source code against static umap library.

CHAPTER

THREE

RUNTIME ENVIRONMENT VARIABLES

The interface to the umap runtime library configuration is controlled by the following environment variables.

```
UMAP_PAGESIZE=$((2*4096)) your_program_that_uses_umap
```

The following environment varialbles may be set:

- `UMAP_PAGE_FILLERS` This is the number of worker threads that will perform read operations from the backing store (including read-ahead) for a specific umap region.

  Default: *std::thread::hardware_concurrency()*

- `UMAP_PAGE_EVICTORS` This is the number of worker threads that will perform evictions of pages. Eviction includes writing to the backing store if the page is dirty and telling the operating system that the page is no longer needed.

  Default: *std::thread::hardware_concurrency()*

- `UMAP_EVICT_HIGH_WATER_THRESHOLD` This is an integer percentage of present pages in the Umap Buffer that informs the Eviction workers that it is time to start evicting pages.

  Default: 90

- `UMAP_EVICT_LOW_WATER_THRESHOLD` This is an integer percentage of present pages in the Umap Buffer that informs the Eviction workers when to stop evicting.

  Default: 70

- `UMAP_PAGESIZE` This is the size of the umap pages. This must be a multiple of the system page size.

  Default: System Page Size

- `UMAP_BUFSIZE` This is the total number of umap pages that may be present within the Umap Buffer.

  Default: (90% of free memory)

- `UMAP_MONITOR_FREQ` This is the interval (in seconds) for the monitoring thread to print statistics, e.g., filled pages, free pages and processed events for debugging or tuning.

  Default: 0

CHAPTER

FOUR

SPARSE MULTI-FILES BACKING STORE INTERFACE

UMap provides an extensible design that supports multiple types of backing stores (e.g., local SSDs, network-interconnected SSDs, and HDDs).

An application that uses UMap can extend the abstract "Store" class to implement its specific backing store interface.

The default store object used by UMap is "StoreFile", which reads and writes to a single regular Linux file.

UMap also provides a sparse multi-files store object called "SparseStore", which creates multiple backing files dynamically and only when needed.

Below is an example of using UMap with a SparseStore object.

```
#include <umap/umap.h>
#include <umap/store/SparseStore.h>
#include <string>
#include <iostream>

void use_sparse_store(
  std::string root_path,
  uint64_t numbytes,
  void* start_addr,
  size_t page_size,
  size_t file_size){

 void * region = NULL;

 // Instantiating a SparseStore objct, the "file_size" parameter specifies the␣
↪granularity of each file.
 // An application that desires to create N files can calculate the file size using␣
↪totalbytes / num_files
 // Note that the file size is rounded to be a multiple of the page size,
 // which results in a number of files that close but not exactly equal to N
 Umap::SparseStore* sparse_store;
 sparse_store = new Umap::SparseStore(numbytes,page_size,root_path,file_size);

 // Check status to make sure that the store object was able to open the directory
 if (store->get_directory_creation_status() != 0){
   std::cerr << "Error: Failed to create directory at " << root_path << std::endl;
   return NULL;
 }

 // set umap flags
 int flags = UMAP_PRIVATE;

 if (start_addr != nullptr)
```

(continues on next page)

(continued from previous page)

```
   flags |= MAP_FIXED;

  const int prot = PROT_READ|PROT_WRITE;

  /* Map region using UMap, Here, the file descriptor passed to umap is -1, as we do␣
↪not start with mapping a file
     instead, file(s) will be created incrementally as needed using the "sparse_store
↪" object. */

  region = umap_ex(start_addr, numbytes, prot, flags, -1, 0, sparse_store);
  if ( region == UMAP_FAILED ) {
    std::ostringstream ss;
    ss << "umap_mf of " << numbytes
       << " bytes failed for " << root_path << ": ";
    perror(ss.str().c_str());
    exit(-1);
  }

  /*
   * some code that uses mapped region goes here.
   */

 // Unmap region
 if (uunmap(region, numbytes) < 0) {
   std::ostringstream ss;
   ss << "uunmap of failure: ";
   perror(ss.str().c_str());
   exit(-1);
 }
 // NOTE: the method "close_files" from SparseStore MUST be called explicitely before␣
↪deleting the object
 int sparse_store_close_files = store->close_files();
 if (sparse_store_close_files != 0 ){
   std::cerr << "Error closing SparseStore files" << std::endl;
   delete store;
   exit(-1);
 }
 delete store;
}
```

CHAPTER

FIVE

CONTRIBUTION GUIDE

This document is intented for developers who want to add new features or bugfixes to umap. It assumes you have some familiarity with git and GitHub. It will discuss what a good pull request (PR) looks like, and the tests that your PR must pass before it can be merged into umap.

5.1 Forking umap

If you aren't an umap deveolper at LLNL, then you won't have permission to push new branches to the repository. First, you should create a fork. This will create a copy of the umap repository that you own, and will ensure you can push your changes up to GitHub and create pull requests.

5.1.1 Developing a New Feature

New features should be based on the `develop` branch. When you want to create a new feature, first ensure you have an up-to-date copy of the `develop` branch:

```
$ git checkout develop
$ git pull origin develop
```

You can now create a new branch to develop your feature on:

```
$ git checkout -b feature/<name-of-feature>
```

Proceed to develop your feature on this branch, and add tests that will exercise your new code. If you are creating new methods or classes, please add Doxygen documentation.

Once your feature is complete and your tests are passing, you can push your branch to GitHub and create a PR.

5.1.2 Developing a Bug Fix

First, check if the change you want to make has been fixed in `develop`. If so, we suggest you either start using the `develop` branch, or temporarily apply the fix to whichever version of umap you are using.

If the bug is still unfixed, first make sure you have an up-to-date copy of the develop branch:

```
$ git checkout develop
$ git pull origin develop
```

Then create a new branch for your bugfix:

```
$ git checkout -b bugfix/<name-of-bug>
```

First, add a test that reproduces the bug you have found. Then develop your bugfix as normal, and ensure to `make test` to check your changes actually fix the bug.

Once you are finished, you can push your branch to GitHub, then create a PR.

5.1.3 Creating a Pull Request

You can create a new PR here. Ensure that your PR base is the `develop` branch of umap.

Add a descriptive title explaining the bug you fixed or the feature you have added, and put a longer description of the changes you have made in the comment box.

Once your PR has been created, it will be run through our automated tests and also be reviewed by umap team members. Providing the branch passes both the tests and reviews, it will be merged into umap.

5.1.4 Tests

Umap uses Bamboo for continuous integration tests. Our tests are automatically run against every new pull request, and passing all tests is a requirement for merging your PR. If you are developing a bugfix or a new feature, please add a test that checks the correctness of your new code.

Umap's tests are all in the `tests` directory.