UMAP Documentation

Release 0.0.3

Marty McFadden

Sep 25, 2021

BASICS

MP-Umap is a collection of libraries that allows multiple processes to share a single file-backed Umap buffer. Just like Umap, it uses user-space page fault handler based on the userfaultfd Linux feature (starting with 4.3 linux kernel). This feature requires a Umap Service to be started and the client applications bind/interact with this service through MP-Umap client API. Presently, this enables sharing of buffer in read-only capacity. The use case is to have a multi-process application accessing a large file through cached pages, 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 mp-umap.

1.1 Dependencies

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

1.1.1 MP-UMAP Build and Installation

MP-Umap provides service and client libraries. The former enables developers to launch mp-umap service that transparently manages the shared UMAP buffer and the client API allows multi-processes to bind/interact with the service thereby enabling shared access to UMAP buffer. The following lines should get you up and running:

```
$ git clone https://github.com/LLNL/umap.git
$ cd umap
$ git checkout mp_umap_pre_release
$ cmake3 -DCMAKE_BUILD_PREFIX=. -DENABLE_TESTS_LINK_STATIC_UMAP=ON -DCMAKE_INSTALL_
↪PREFIX=../install ..;
$ cmake3 --build . --target install
```

MP-Umap is an experimental branch of Umap that enables sharing of file-backed Umap buffers to multiple processes. At the moment, this feature is only available in read-only mode. By default, mpumap builds with 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*.

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

1.1.2 Basic Usage

MP-Umap library provides seperate interfaces for service and client processes.

Service communicate with the client processes through a Unix Domain socket. This can be accomplished by calling the following mpumapd API with UNIX domain socket path as an argument. When this argument is specified NULL, default option of UMAP_SERVER_PATH, which is set to '/tmp/umap-server'

```
void umap_server(
std::string filename
);
```

A simple mpumap Service looks like the following:

```
#include "umap/mpumapd.h"
#include <string>
#include <iostream>

int main(int argc, char **argv){
  std::string filename;
  if(argc < 2){
        filename = std::string(UMAP_SERVER_PATH);
        std::cout<<"Using default mp-umapd socket path";
        std::cout<<"Usage: umap-server <unix socket path>";
  }else{
          filename = std::string(argv[1]);
  }
  umap_server(filename.c_str());
  return 0;
}
```

In order for a process to interact with the umap-service client applications interact with target service through the following client API calls defined in mpumapclient.h

```
/*!
 * @function    init_umap_client
 * @abstract    initializes connection with an already running mpumap service
 * @param       sock_path         Path to Unix Domain socket hosted by target
 *                                 mpumap service. When a NULL value is passed,
 *                                 a default UMAP_SERVER_PATH is used
 * @result      Exits if it fails to connect to the service.
*/
void init_umap_client(const char *sock_path);

/*!
 * @function    close_umap_client
 * @abstract    Closes an already established mpumap service connection
 * @result      Exits if no service is found to be connected with
*/
void close_umap_client();


/*!
 * @function    client_umap
 * @abstract    Functionally equivalent to mmaping a file. Should be called only
 *                 after connection with target umap-service has been established using␣
↪init_umap_client
 * @param       filename         Full path to the file to be memory mapped
 * @param        prot                At present it only accepts PROT_READ as we serve␣
↪read-only buffers
 *                                 This parameter is to catch instances where users␣
↪intend to use the buffer
 *                                 other than read-only purposes.
 * @param        flags                At present it only accepts MAP_SHARED as these␣
↪buffers are supposed to
 *                                 be shared between multiple processes, which include␣
↪the mpumap service.
```

(continues on next page)

(continued from previous page)

```
 *                                This is to catch cases where user intends to use it␣
↪otherwise.
 * @param         addr               non-NULL value is intended for fixed address.␣
↪Floating address otherwise.
 * @result      Return NULL on failure. Else return the userspace mapped address
*/
void* client_umap(
    const char *filename
  , int prot
  , int flags
  , void *addr
);


/*!
 * @function    init_umap_client
 * @abstract    Removes the mapping from the client process' address space.
 * @param       filename         Path to file that has previously been mapped
 *                                by client_umap call.
 * @result      Return -1 on error, 0 On Success
*/
int client_uunmap(
    const char *filename
);

/*
 * The following mpumap client API calls provide visibility to mpumap
 * service's settings to the client. This allows clients to use these
 * values as they deem necessary. These calls need to be called after
 * establishing connection with a mpumap service through init_umap_client
 * API call.
 */
long umapcfg_get_system_page_size( void );
uint64_t umapcfg_get_max_pages_in_buffer( void );
uint64_t umapcfg_get_umap_page_size( void );
uint64_t umapcfg_get_num_fillers( void );
uint64_t umapcfg_get_num_evictors( void );
int umapcfg_get_evict_low_water_threshold( void );
int umapcfg_get_evict_high_water_threshold( void );
uint64_t umapcfg_get_max_fault_events( void );
```

Here is a simple mpumap Client application that shows the use of API defined in mpumapclient.h

```
#include "umap/mpumapclient.h"
#include <unistd.h>
#include <sys/mman.h>
using namespace std;
#include <iostream>

void disp_umap_env_variables() {
  std::cout
      << "Environment Variable Configuration (command line arguments obsolete):\n"
```

(continues on next page)

(continued from previous page)

```
      << "UMAP_PAGESIZE                  - currently: " << umapcfg_get_umap_page_size()
↪<< " bytes\n"
      << "UMAP_PAGE_FILLERS              - currently: " << umapcfg_get_num_fillers() <<
↪" fillers\n"
      << "UMAP_PAGE_EVICTORS             - currently: " << umapcfg_get_num_evictors() <
↪< " evictors\n"
      << "UMAP_BUFSIZE                   - currently: " << umapcfg_get_max_pages_in_
↪buffer() << " pages\n"
      << "UMAP_EVICT_LOW_WATER_THRESHOLD  - currently: " << umapcfg_get_evict_low_water_
↪threshold() << " percent full\n"
      << "UMAP_EVICT_HIGH_WATER_THRESHOLD - currently: " << umapcfg_get_evict_high_water_
↪threshold()
      << " percent full\n"
      << std::endl;
}

int main(int argc, char *argv[]){
  void *mapped_addr, *mapped_addr2;;
  char *read_addr, *read_addr2;
  bool diff = false;
  int i=0;
  std::string sock_path, filename1, filename2;
  int opt;
  char opt_string[] = "c:s:f:";
  while((opt = getopt(argc, argv, opt_string)) != -1){
    switch (opt) {
      case 'c':
        sock_path = std::string(optarg);
        break;
      case 'f':
        filename1 = std::string(optarg);
        break;
      case 's':
        filename2 = std::string(optarg);
        break;
    }
  }
  if(filename1.empty() || filename2.empty()){
    std::cerr<<"Usage: ./umap-client -c <socket_path> -f <file1_name> -s <file2_name>";
    exit(-1);
  }

  if(sock_path.empty()){
    sock_path = std::string(UMAP_SERVER_PATH);
  }
  init_umap_client(sock_path.c_str());
  disp_umap_env_variables();
  mapped_addr = client_umap(filename1.c_str(), PROT_READ, MAP_SHARED, NULL);
  mapped_addr2 = client_umap(filename2.c_str(), PROT_READ, MAP_SHARED, NULL);
  char *end_addr = (char *)mapped_addr + 4096*1024;
  for(read_addr = (char *)mapped_addr, read_addr2 = (char *)mapped_addr2; read_addr <=␣
↪end_addr ; read_addr++, read_addr2++ ){
```

(continues on next page)

(continued from previous page)

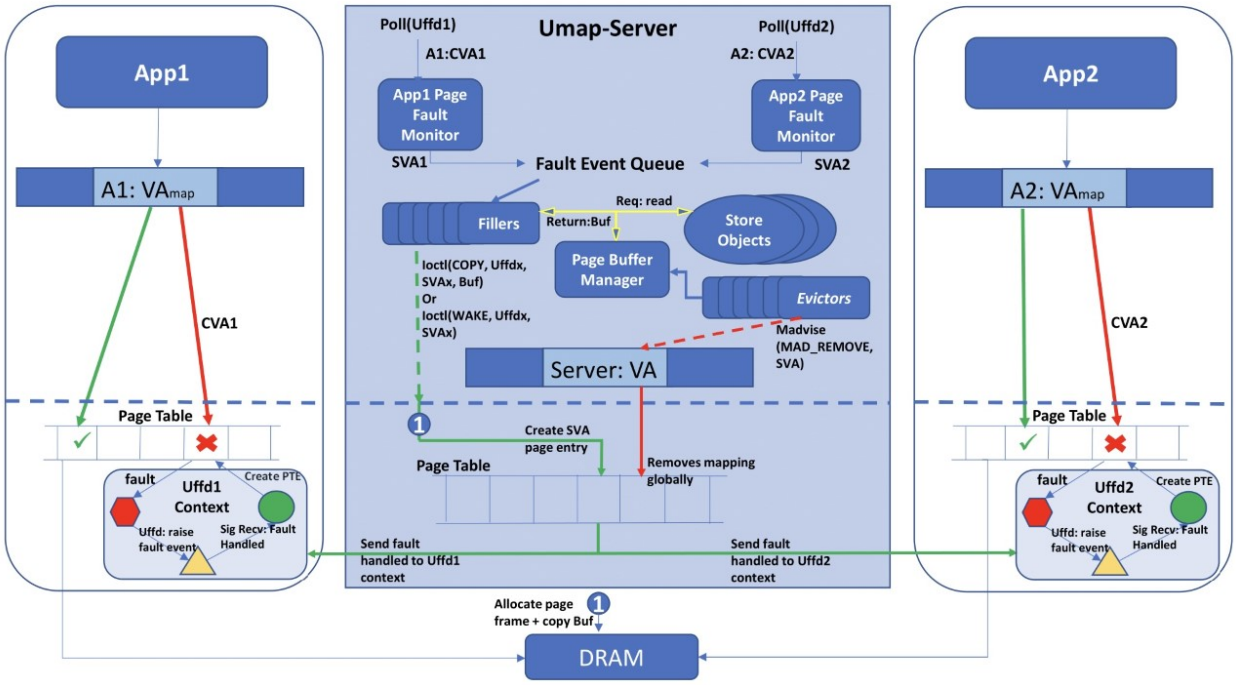
```
    if(*read_addr != *read_addr2){
        diff = true;
        std::cout<<"Files differ at offset "<<(unsigned long)read_addr - (unsigned␣
↪long)mapped_addr<<std::endl;
        break;
    }
  }
  if(!diff){
    std::cout<<"No difference detected at page boundaries"<<std::endl;
  }
  client_uunmap(filename1.c_str());
  client_uunmap(filename2.c_str());
  return 0;
}
```

Note: use -lmpumapd and -lmpumapclient to link service and client apps, respectively.

CHAPTER

TWO

MP-UMAP PAGE FAULT HANDLING



This page provides design details on MP-UMap. For this description, we call an application that uses mpumapd library to instantiate a MP-UMap service as a UMap-Server. The Applications App1 and App2 use mpumapclient library to communicate and interface with UMap-Server. In contrast to UMap, as shown in Figure, UMap-Server runs independently of the application and it launches one fault monitor thread per client application. Absence of a valid page table entry, causes application thread to switch to kernel mode and it creates a uffd_msg holding page fault information before blocking itself. The fault monitor polling on UFFD_fd file handler receives the faulting address Client Virtual Address (CVA) as part of the message. In the figure, we illustrate a scenario where two applications App1 and App2 page fault on addresses A1:CVA1 and A2:CVA2, respectively. Corresponding fault monitoring threads translate these application addresses to server's address space as Server Virtual Address (SVA) SVA1 and SVA2. These addresses are then placed on a fault event queue to a group of worker threads called filler threads (FTs). The job of a filler in UMap-server, is to perform the following:

1. Read the appropriate page from the Store Objects
2. Update status of the page in Page Buffer Manager
3. Issue Ioctls on target Uffd_fd to copy the contents of the read buffer into the faulting page
4. Signal the blocked application about completion of the faults handling

Additionally, the fillers need to handle the case when SVA1 and SVA2 are the same. This scenario is different from contending requests arising from different threads of a single process. There, the UMap-server receives the page faults on the same fault monitor, which allows these requests to be grouped together. Also, a single UFFDIO_IOCOPY is capable of unblocking any number of threads of a process blocked on a single memory region. In case of requests arriving from multiple clients, the Page Buffer Manager, which is responsible for maintaining a record of all mapped regions, informs fillers of a region's current state. In case the memory region is in an actively filling state, the contending requests wait for the region's state to change. On completion of the initial request, the UFFD_IOCOPY ioctl is issued for the original requestor, while for all other blocked processes a UFFDIO_WAKE ioctl is issued. A UFFDIO_COPY allocates pages for the SVA region, copies the contents of a buffer and unblocks associated context. UFFDIO_WAKE only unblocks a target context. The unblocked context assigns CVAs to physical page frames thereby handling the page fault.

Another key aspect of Page buffer management is to limit the size of the page buffer while retaining useful pages. This is accomplished by Evictor Threads (ETs) that operate on the same eviction policies of UMap. It is to be noted that the use of memfd to create a shared region, creates a volatile backing store in memory. This means that madvice of MADV_DONTNEED may mark pages ready to be removed but due to memory based backing store, the kernel still retains pages in memory. To resolve this, ETs issue MADV_REMOVE as shown in the Figure.

During our testing, it was observed that having a fixed sized Page cache buffer potentially adds pressure on system memory, which in turn prevents the UMap-server from scaling the number of concurrent processes it can support. This drawback was eliminated through the adaptive page cache size management optimization previously incorporated into UMap and ported to MP-UMap.

CHAPTER

THREE

ADVANCED CONFIGURATION

Listed below are the MP-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 for mp-umap service apps. When this support is enabled, you may cause mpumapd 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 mpumapd library will display its runtime statistics before client_uunmap() completes.
- `ENABLE_TESTS` This option enables the compilation of the programs under the tests directory of the mp-umap source code.
- `ENABLE_TESTS_LINK_STATIC_UMAP` This option enables the compilation of the programs under the tests directory of the mp-umap source code against static mp-umap libraries.

CHAPTER

FOUR

RUNTIME ENVIRONMENT VARIABLES

The interface to the mp-umap service configuration is controlled by the following environment variables. Note: these environment variables only control mp-umap service's configuration and does not effect applications using mpumap-client library. Also, client applications do not control and cannot modify service application's configuration.

```
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 mp-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 MP-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 MP-Umap Buffer that informs the Eviction workers when to stop evicting.

  Default: 70

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

  Default: System Page Size

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

  Default: (90% of free memory)

- `UMAP_BUFADAPT_FREQ` This sets the frequency with which the system memory pressure is evaluated for memory pressure. MP-Umap Buffer is adapted depending upon the memory pressure experienced by the system. This parameter is over-ridden when UMAP_BUFSIZE is set.

  Default: (3 seconds)

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.