Bahir Dar University

Bahir Dar Institute of Technology

Faculty of Computing

Department of Software Engineering

Operating system and System Programming

Individual Assignment on

int msync(void *addr, size_t length, int flags) system call

Made by: Dawit Zewdu Munie

ID: BDU1307571

Submitted to: Lecturer Wendimu Baye

Submission date: 17/11/2014 E.c

Contents

int msync(void *addr, size_t length, int flags) system call

Introduction

What is generally a system call?

A system call is a request that a program makes to the operating system. It enables a program to use features and controls available through the operating system's API. System calls carry out actions at the system level, such interacting with hardware and reading and writing files. Developers can avoid developing custom OS-supported functions by using pre-written ones by using system calls. This makes app creation easier, increases app stability, and increases the "portability" of programs between OS versions.

1. What /Why / How, this system call?

What is int msync(void *addr, size_t length, int flags) system call?

msync() system call especially int msync(void *addr, size_t length, int flags) system call is one of the most important system call that used to synchronize a file with a memory map. msync() is a system call used to force the contents of a mapped region to be flushed to disk.

The msync() function first appeared in 4.4BSD. It was modified to conform to IEEE Std 1003.1b-1993 (``POSIX.1'') in NetBSD 1.3.

NAME

msync - synchronize a file with a memory map

SYNOPSIS

#include <sys/mman.h>

int msync(void *start, size_t length, int flags);

Why int msync(void *addr, size_t length, int flags) system call?

We are compelled to include int msync(void *addr, size_t length, int flags) system call system call in an operating system because of the numerous advantages that a system can only receive from it. To name a few:

The kernel automatically carries modifications of the contents of a MAP_SHARED mapping through to the underlying file, but, by default, provides no guarantees about when such synchronization will occur. (SUSv3 doesn’t require an implementation to provide such guarantees.) The msync() system call gives an application explicit control over when a shared mapping is synchronized with the mapped file. Synchronizing a mapping with the underlying file is useful in various scenarios. For example, to ensure data integrity, a database application may call msync() to force data to be written to the disk. Calling msync() also allows an application to ensure that updates to a writable mapping are visible to some other process that performs a read()

on the file. A unified virtual memory system is not required by SUSv3 and is not employed on all UNIX implementations. On such systems, a call to msync() is required to make changes to the contents of a mapping visible to other processes that read() the file, and the MS_INVALIDATE flag is required to perform the converse action of making writes to the file by another process visible in the mapped region. Multiprocess applications that employ both mmap() and I/O system calls to operate on the same file should be designed to make appropriate use of msync() if they are to be portable to systems that don't have a unified virtual memory system.

How int msync(void *addr, size_t length, int flags) system call works?

The **msync**() system call writes all pages with shared modifications in the specified region starting from *addr* and continuing for *len* bytes. *addr* should be a multiple of the page size. Any required synchronization of memory caches will also take place at this time. Filesystem operations on a file that is mapped for shared modifications are unpredictable except after an **msync**().To be more precise, the part of the file that corresponds to the memory area starting at *start* and having length *length* is updated.
msync() flushes changes made to the in-core copy of a file that was mapped into memory using mmap(2) back to the filesystem. Upon successful completion, the value 0 is returned; otherwise the value -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

The following errors may be reported:

[EBUSY]
The MS_INVALIDATE flag was specified and a portion of the specified region was locked with mlock(2).

[EINVAL]
The specified *flags* argument was invalid.

[EINVAL]
The *addr* parameter was not page aligned or *addr* and *size* specify a region that would extend beyond the end of the address space.

[ENOMEM]
Addresses in the specified region are outside the range allowed for the address space of the process, or specify one or more pages which are unmapped.

[EIO]
An I/O error occurred while writing.

2. Briefly describe about the list of parameters and flags?

The list of parameters used in int msync(void *start, size_t length, int flags)

The list of parameters used in int msync(void *start, size_t length, int flags) are:

***1.* start**

The beginning of the range of addresses that you want to synchronize.

***2.* length**

The length of the range of addresses, in bytes.

3. **The *flags* argument**

It is the bitwise OR of zero or more of the following values:

- ✓ MS_ASYNC Perform asynchronous writes.
- ✓ MS_SYNC Perform synchronous writes.
- ✓ MS_INVALIDATE Invalidate cached data after writing.

3. List the flags, their purpose with code implementation (give Example source code with output)

list of flags and their purpose with detail implementations

MS_SYNC

Perform a synchronous file write. The call blocks until all modified pages of the memory region have been written to the disk.

IMPLEMENTATION

MS_SYNC is defined in header *sys/mman.h*.

Perform synchronous writes.

MS_SYNC can be used in the following way:

if (msync(data, file_size - file_offset, MS_SYNC)) {

The full source code is listed as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/unistd.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/mman.h>
#include <string.h>
int main(int argc, char **argv) {
  char buf[4096];
  char buf2[4096];
  int fd, i;
  char *path;
  unsigned int write_count;
  unsigned long file_size;
  unsigned long file_offset = 0;
  char* data;
  if (argc < 4) {
    fprintf(stderr, "Too few arguments\n");
    return EXIT_FAILURE;
  }
  path = argv[1];
  file_size = atol(argv[2]) * 1024 * 1024;
  write_count = atoi(argv[3]);
  if (argc > 4)
    file_offset = atol(argv[4]) * 1024 * 1024;
  if ((fd = open("/dev/urandom", O_RDONLY)) < 0) {
    perror("Error opening /dev/urandom");
    return EXIT_FAILURE;
  }
  if (read(fd, buf, sizeof(buf)) != 4096) {
    perror("Error initializing buffer");
    return EXIT_FAILURE;
  }
  close(fd);
  printf("Initializeing target...\nPath: %s, Size: %luMB, Offset:
%luMB, Count: %u\n",
      path, file_size / 1024 / 1024, file_offset / 1024 / 1024,
write_count);
  if ((fd = open(path, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
    perror("Error opening target");
    return EXIT_FAILURE;
  }

  if (ftruncate(fd, file_size)) {
    perror("Error resizing");
    return EXIT_FAILURE;
  }
```

```
  puts("Mapping file");
  if ((data = mmap(NULL, file_size - file_offset, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, file_offset)) == MAP_FAILED) {
    perror("Mmap failed");
    return EXIT_FAILURE;
  }
  puts("Writing data...");
  for (i = 0; i < write_count; ++i) {
    // first read the page
    memcpy(buf2, data + i * 4096, sizeof(buf2));
    // then write it
    memcpy(data + i * 4096, buf, sizeof(buf));
    if (msync(data, file_size - file_offset, MS_SYNC)) {
      perror("Msync failed");
      return EXIT_FAILURE;
    }
  }
  puts("Unmapping file...");
  if (munmap(data, file_size - file_offset)) {
    perror("Munmap failed");
    return EXIT_FAILURE;
  }
  close(fd);
  puts("Success!");
  return EXIT_SUCCESS;
}
```

OUTPUT IMAGE:

Note: The outputs are indicated by red line for ease of simplicity.

OUTPUT IN WORDS:

Too few arguments

MS_ASYNC

Perform an asynchronous file write. The modified pages of the memory region are written to the disk at some later point and are immediately made visible to other processes performing a read() on the corresponding file region.

Implementation

MS_ASYNC is defined in header *sys/mman.h*.

Perform asynchronous writes.

MS_ASYNC can be used in the following way:

The full source code is listed as follows:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>
typedef struct {
        int integer;
        char string[24];
} RECORD;

#define NRECORDS (10)
void display_record()
{
        int i;
        FILE *fp;
        RECORD record;
        fp = fopen("records.dat", "r+");
        for (i = 0; i < NRECORDS; i++) {
                fread(&record, sizeof(record), 1, fp);
                printf("%3d: %s\n", record.integer, record.string);
        }
        fclose(fp);
        printf("\n");
}
int main(void)
{
        int i, f;
        FILE *fp;
        RECORD record, *mapped;
        fp = fopen("records.dat", "w+");
        for (i = 0; i < NRECORDS; i++) {
                record.integer = i;
                sprintf(record.string, "RECORD-%d", i);
```

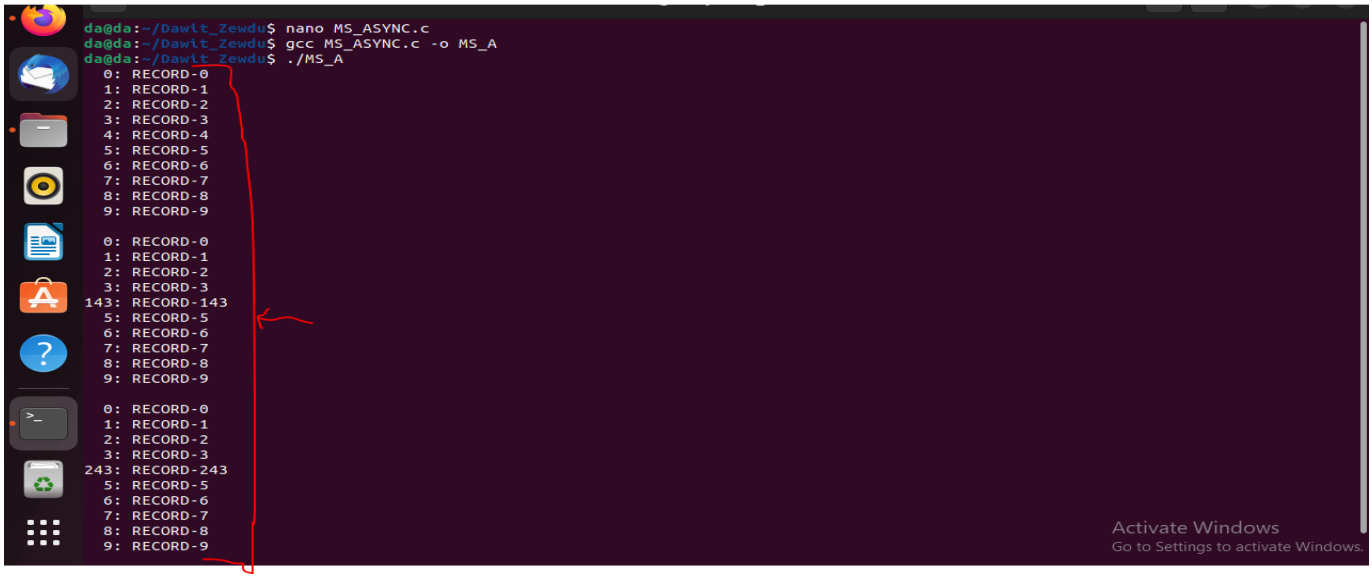
```
                fwrite(&record, sizeof(record), 1, fp);
        }
        fclose(fp);
        display_record();

        fp = fopen("records.dat", "r+");
        fseek(fp, 4*sizeof(record), SEEK_SET);
        fread(&record, sizeof(record), 1, fp);
        record.integer = 143;
        sprintf(record.string, "RECORD-%d", record.integer);
        fseek(fp, 4*sizeof(record), SEEK_SET);
        fwrite(&record, sizeof(record), 1, fp);
        fclose(fp);
        display_record();
        f = open("records.dat", O_RDWR);
        mapped = (RECORD *)mmap(0, NRECORDS*sizeof(record),
PROT_READ|PROT_WRITE,
                                MAP_SHARED, f, 0);
        mapped[4].integer = 243;
        sprintf(mapped[4].string, "RECORD-%d", mapped[4].integer);
        msync((void*)mapped, NRECORDS*sizeof(record), MS_ASYNC);
        munmap((void*)mapped, NRECORDS*sizeof(record));
        close(f);
        display_record();

        return 0;
}
```

OUTPUT IN IMAGE:

Note: The outputs are indicated by red line for ease of simplicity.



OUTPUT IN WORDS:

0: RECORD-0

1: RECORD-1

2: RECORD-2

3: RECORD-3

4: RECORD-4

5: RECORD-5

6: RECORD-6

7: RECORD-7

8: RECORD-8

9: RECORD-9

0: RECORD-0

1: RECORD-1

2: RECORD-2

3: RECORD-3

143: RECORD-143

5: RECORD-5

6: RECORD-6

7: RECORD-7

8: RECORD-8

9: RECORD-9

0: RECORD-0

1: RECORD-1

2: RECORD-2

3: RECORD-3

243: RECORD-243

5: RECORD-5

6: RECORD-6

7: RECORD-7

8: RECORD-8

9: RECORD-9

Another way of distinguishing these two values is to say that after an MS_SYNC operation, the memory region is synchronized with the disk, while after an MS_ASYNC operation, the memory region is merely synchronized with the kernel buffer cache.

Note: If we take no further action after an MS_ASYNC operation, then the modified pages in the memory region will eventually be flushed as part of the automatic buffer flushing performed by the pdflush kernel thread (kupdated in Linux 2.4 and earlier). On Linux, there are two (nonstandard) methods of initiating the output sooner. We can follow the call to msync() with a call to fsync() (or fdatasync()) on the file descriptor corresponding to the mapping. This call will block until the buffer cache is synchronized with the disk. Alternatively, we can initiate asynchronous write out of the pages using the posix_fadvise() POSIX_FADV_DONTNEED operation. (The Linux-specific details in these two cases are not specified by SUSv3.)

One other value can additionally be specified for flags:

MS_INVALIDATE

Invalidate cached copies of mapped data. After any modified pages in the memory region have been synchronized with the file, all pages of the memory region that are inconsistent with the underlying file data are marked as invalid. When next referenced, the contents of the pages will

be copied from the corresponding locations in the file. As a consequence, any updates that have been made to the file by another process are made visible in the memory region.

IMPLEMENTATION

MS_INVALIDATE is defined in header *sys/mman.h*.

Invalidate mappings.

MS_INVALIDATE can be used in the following way:

msync(count, sizeof(*count), MS_SYNC | MS_INVALIDATE);

The full sourec code is listed as follows:

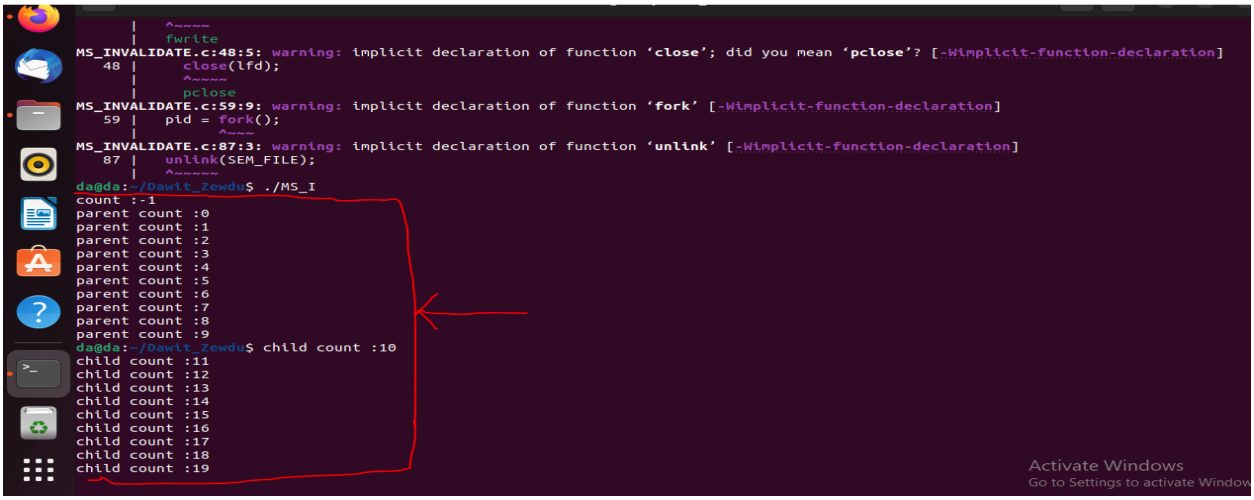
```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <sys/mman.h>
#define SEM_FILE  "lock.sem"
#define MMAP_FILE  "mmap.txt"
int main(int argc, char *argv[])
{
  sem_t *sem;//   w  w  w   . d e    mo  2   s   .  c  o  m
  pid_t pid;
  int i;
  int *count = NULL;
  int lfd = -1, mfd = -1;
  int value = -1;
  /* create file for mmap */
  mfd = open(MMAP_FILE, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  if (-1 == mfd) {
    fprintf(stderr, "create file for mmap error :%s\n",
        strerror(errno));
    goto failure;
  }
  write(mfd, &value, sizeof(value));
  /* mmap */
  count = mmap(NULL, sizeof(*count), PROT_READ | PROT_WRITE, MAP_SHARED, mfd,
0);
  if (MAP_FAILED == count) {
    fprintf(stderr, "mmap error :%s\n",
```

```
      strerror(errno));
  goto failure;
}
fprintf(stdout, "count :%d\n",
    *(count));
count[0] = 0;
/* create lock file for semaphore */
lfd = open(SEM_FILE, O_CREAT, S_IRUSR | S_IWUSR);
if (-1 == lfd) {
  fprintf(stderr, "create file for locked semaphore file :%s\n",
      strerror(errno));
  goto failure;
}
if (-1 != lfd)
  close(lfd);
/* sem_open */
sem = sem_open(SEM_FILE, O_CREAT, S_IRUSR | S_IWUSR, 1);
if (SEM_FAILED == sem) {
  fprintf(stderr, "sem_open error :%s\n",
      strerror(errno));
  goto failure;
}
/* setbuf for stdout */
setbuf(stdout, NULL);
/* fork */
pid = fork();
if (-1 == pid) {
  fprintf(stderr, "fork error :%s\n",
      strerror(errno));
  goto failure;
} else if (0 < pid) {
  /* parent */
  sem_wait(sem);
  for (i = 0; 10 > i; i++)
    fprintf(stdout, "parent count :%d\n",
        count[0]++);
  msync(count, sizeof(*count), MS_SYNC | MS_INVALIDATE);
  sem_post(sem);
} else {
  /* child */
  sem_wait(sem);
  for (i = 0; 10 > i; i++)
    fprintf(stdout, "child count :%d\n",
        count[0]++);
  sem_post(sem);
```

```
    exit(EXIT_SUCCESS);
  }
  if (-1 != lfd)
    close(lfd);
  if (-1 != mfd)
    close(mfd);
  /* sem_unlink */
  sem_unlink(SEM_FILE);
  unlink(SEM_FILE);
  /* munmap */
  munmap(count, sizeof(*count));
  /* unlink file for mmap */
  unlink(MMAP_FILE);
  return 0;
failure:
  if (-1 != lfd)
    close(lfd);
  if (-1 != mfd)
    close(mfd);
  sem_unlink(SEM_FILE);
  unlink(SEM_FILE);
  /* munmap */
  munmap(count, sizeof(*count));
  /* unlink file for mmap */
  unlink(MMAP_FILE);
  exit(EXIT_FAILURE);
}
```

OUTPUT IN IMAGE:

Note: The outputs are indicated by red line for ease of simplicity.



OUTPUT IN WORDS:

count :-1

parent count :0

parent count :1

parent count :2

parent count :3

parent count :4

parent count :5

parent count :6

parent count :7

parent count :8

parent count

References

1. *https://man7.org/linux/man-pages/man2/msync.2.html*

2. *https://www.demo2s.com/c/c-msync-void-mapped-nrecords-sizeof-record-ms-async-cuhw.html*

3. *https://www.demo2s.com/c/c-msync-count-sizeof-count-ms-sync-ms-invalidate.html*

4. *https://www.demo2s.com/c/c-if-msync-data-file-size-file-offset-ms-sync.html*