Memory Mapped I/O

- Basic idea: map a part of a file (or other object) into your virtual address space.
- Accesses to the mapped part of your address space converted into file reads and writes by the OS.
- Why?
  - May be faster than conventional file i/o.
  - May lead to better, less error-prone, programming style.

NAME

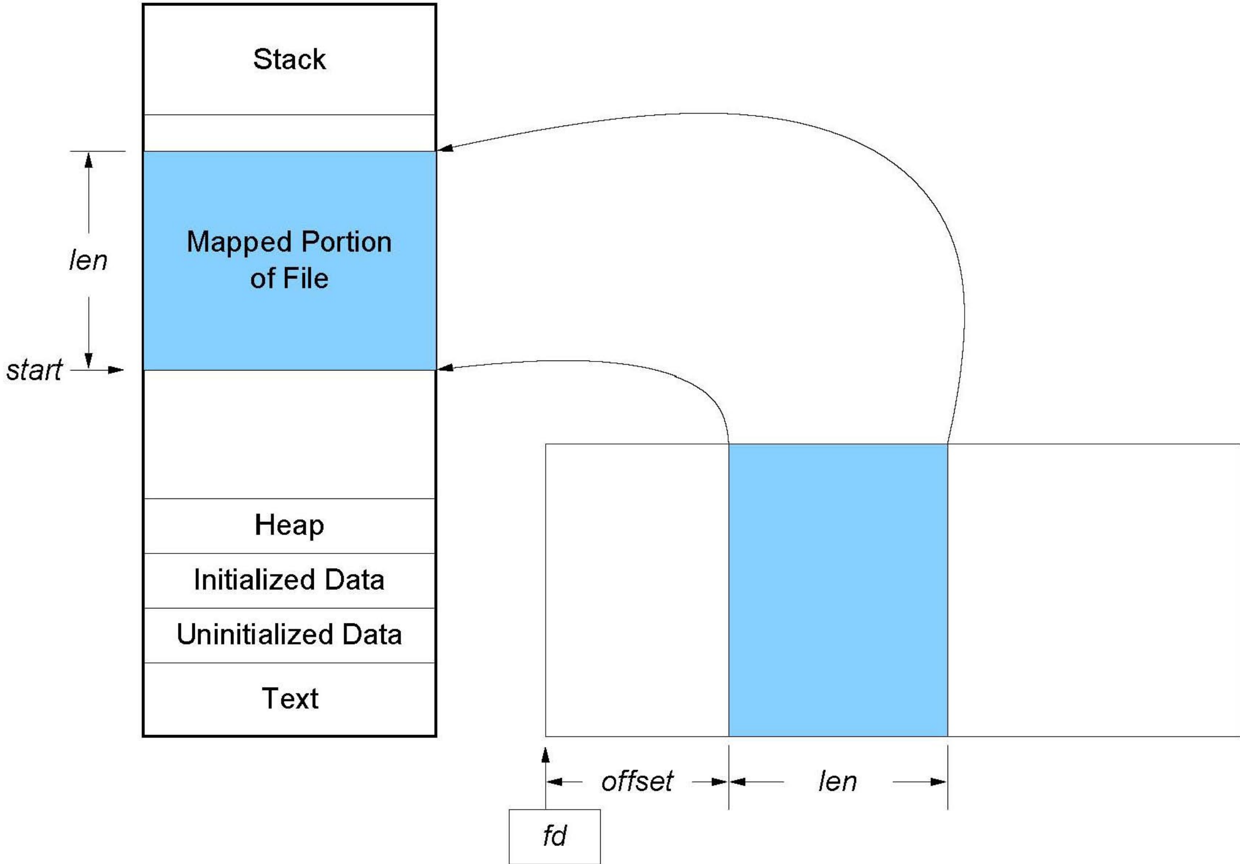
mmap, munmap - map or unmap files or devices into memory

SYNOPSIS

```
#include <sys/mman.h>

#ifdef _POSIX_MAPPED_FILES

void  *mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);

int munmap(void *start, size_t length);

#endif
```

DESCRIPTION

The mmap function asks to map length bytes starting at offset offset from the file (or other object) specified by the file descriptor fd into memory, preferably at address start. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by mmap, and is never 0.


Stack
len
Mapped Portion
of File
start
Heap
Initialized Data
Uninitialized Data
Text
offset
len
fd

```
The prot argument describes the desired memory protection (and must not
conflict  with the open mode of the file). It is either PROT_NONE or is
the bitwise OR of one or more of the other PROT_* flags.

PROT_EXEC  Pages may be executed.

PROT_READ  Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE  Pages may not be accessed.
```

PROT_NONE seems a bit strange. But basically means that if we attempt to reference the mapped region, we get a SIGSEGV. We can then change the protection dynamically (will see later).

The flags parameter specifies the type of the mapped object, mapping options and whether modifications made to the mapped copy of the page are private to the process or are to be shared with other references. It has bits

MAP_FIXED Do not select a different address than the one specified. If the specified address cannot be used, mmap will fail. If MAP_FIXED is specified, start must be a multiple of the pagesize. Use of this option is discouraged.

MAP_SHARED Share this mapping with all other processes that map this object. Storing to the region is equivalent to writing to the file. The file may not actually be updated until msync(2) or munmap(2) are called.

MAP_PRIVATE
Create a private copy-on-write mapping. Stores to the region do not affect the original file. It is unspecified whether changes made to the file after the mmap call are visible in the mapped region.

You must specify exactly one of MAP_SHARED and MAP_PRIVATE.

Basically forget about MAP_FIXED.

Also some non-standard flogs known to Linux. Most significant one:

```
MAP_ANONYMOUS
       The mapping is not backed by any file; the fd and  offset  argu-
       ments  are ignored.  This flag in conjunction with MAP_SHARED is
       implemented since Linux 2.4.
```

Allows for a block of virtual addresses to be mapped into your process' address space ... no file association (`fd` ignored in the call to `mmap()`).

Back to the manual page:

```
fd  should  be a valid file descriptor, unless MAP_ANONYMOUS is set, in
which case the argument is ignored.

offset should be a multiple of the page size as  returned  by  getpage-
size(2).

Memory  mapped  by  mmap  is  preserved  across  fork(2), with the same
attributes.
```

Note that the memory mapped region is not preserved across an `execve()`.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

Interpretation: don't change the size of a mmap-ed file.

The munmap system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address start must be a multiple of the page size. All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate SIGSEGV. It is not an error if the indicated range does not contain any mapped pages.

RETURN VALUE

On success, mmap returns a pointer to the mapped area. On error, MAP_FAILED (-1) is returned, and errno is set appropriately. On success, munmap returns 0, on failure -1, and errno is set (probably to EINVAL).

ERRORS

EBADF fd is not a valid file descriptor (and MAP_ANONYMOUS was not set).

EACCES A file descriptor refers to a non-regular file. Or MAP_PRIVATE was requested, but fd is not open for reading. Or MAP_SHARED was requested and PROT_WRITE is set, but fd is not open in read/write (O_RDWR) mode. Or PROT_WRITE is set, but the file is append-only.

EINVAL We don't like start or length or offset. (E.g., they are too large, or not aligned on a PAGESIZE boundary.)

ETXTBSY
MAP_DENYWRITE was set but the object specified by fd is open for

writing.

EAGAIN The file has been locked, or too much memory has been locked.

ENOMEM No memory is available, or the process's maximum number of mappings would have been exceeded.

ENODEV The underlying filesystem of the specified file does not support memory mapping.

Use of a mapped region can result in these signals:

SIGSEGV
Attempted write into a region specified to mmap as read-only.

SIGBUS Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

An Example: File Copying

```
// Memory-mapped file copy. Adapted from Stevens.
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>/* mmap() */
#include <fcntl.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
    int fdin, fdout;
    char *src, *dst;
    struct stat statbuf;

    if (argc != 3) {
      fprintf(stderr,"usage: a.out <fromfile> <tofile>");
      exit(1);
    }
    if ( (fdin = open(argv[1], O_RDONLY)) < 0) {
      fprintf(stderr,"can't open %s for reading", argv[1]);
```

```
  exit(1);
}
if ( (fdout = open(argv[2], O_RDWR | O_CREAT | O_TRUNC, 0644)) < 0) {
  fprintf(stderr, "can't creat %s for writing", argv[1]);
  exit(1);
}
if (fstat(fdin, &statbuf) < 0) { /* need size of input file */
  fprintf(stderr,"fstat error");
  exit(2);
}

//set size of output file
if (lseek(fdout, statbuf.st_size - 1, SEEK_SET) == -1) {
  fprintf(stderr,"lseek error");
  exit(2);
}
if (write(fdout, "", 1) != 1) {
  fprintf(stderr,"write error");
  exit(2);
}
if ( (src = mmap(0, statbuf.st_size, PROT_READ,
 MAP_FILE | MAP_SHARED, fdin, 0)) == (caddr_t) -1) {
```

```
    fprintf(stderr,"mmap error for input");
    exit(2);
  }
  if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
   MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1) {
    fprintf(stderr,"mmap error for output");
    exit(2);
  }

  memcpy(dst, src, statbuf.st_size); // does the file copy

  exit(0);
}
```

Compare performance against standard file copy which reads and writes 8k blocks of the file.

```
  while (1) {
    num = read(fdin, buffer, 8192);
    write(fdout, buffer, num);
    if (num < 8192) exit(0);
  }
```

```
dingo% pwd
~/tmp
dingo% perl -e 'print "a" x 40000000' > infile1
dingo% perl -e 'print "a" x 40000000' > infile2
dingo% ls -l infile?
-rw-r--r--    1 mayo   root     40000000 Nov  3 10:29 infile1
-rw-r--r--    1 mayo   root     40000000 Nov  3 10:29 infile2
dingo% time ./mcopy infile1 outfile1
0.030u 0.150s 0:02.40 7.5%      0+0k 0+0io 19618pf+0w
dingo% time ./copy infile2 outfile2
0.010u 0.160s 0:04.23 4.0%      0+0k 0+0io 83pf+0w
```

Not dramatic performance differences on ix86 Linux. Certainly dramatic differences in programming style.

Related Syscall: `mprotect`

```
NAME
  mprotect - control allowable accesses to a region of memory

SYNOPSIS
  #include <sys/mman.h>

  int mprotect(const void *addr, size_t len, int prot);

DESCRIPTION
  mprotect  controls  how  a  section  of  memory may be accessed.  If an
  access is disallowed by the protection given it, the program receives a
  SIGSEGV.

  prot is a bitwise-or of the following values:

    PROT_NONE  The memory cannot be accessed at all.

    PROT_READ  The memory can be read.

    PROT_WRITE The memory can be written to.
```

```
PROT_EXEC  The memory can contain executing code.

The  new  protection replaces any existing protection.  For example, if
the memory had previously been marked PROT_READ, and mprotect  is  then
called with prot PROT_WRITE, it will no longer be readable.
```

Example mased on `mprotect()` manual page:

```
int main(void)
{
  char *p; char c; int psize;

  psize = getpagesize(); printf("Pagesize is %d\n", psize); fflush(stdout);

  /* Allocate a buffer; it will have the default protection
     of PROT_READ|PROT_WRITE. */
  p = malloc(1024+psize-1);
  if (!p) { perror("Couldn't malloc(1024)"); exit(errno);}

  /* Align to a multiple of PAGESIZE, assumed to be a power of two */
  p = (char *)(((int) p + psize-1) & ~(psize-1));
```

```
    c = p[666];          /* Read; ok */
    p[666] = 42;         /* Write; ok */

    /* Mark the buffer read-only. */
    if (mprotect(p, 1024, PROT_READ)) {
        perror("Couldn't mprotect");
        exit(errno);
    }

    c = p[666];          /* Read; ok */
    printf("READ OK\n"); fflush(stdout);
    p[666] = 42;         /* Write; program dies on SIGSEGV */
    printf("WRITE OK\n"); fflush(stdout);
    exit(0);
}
```

Using `mmap()` to the same effect (but easier):

```
/* Allocate a buffer; it will have the default
   protection of PROT_READ|PROT_WRITE. */
//  p = malloc(1024+psize-1);
//if (!p) {
//    perror("Couldn't malloc(1024)");
//    exit(errno);
//}


///* Align to a multiple of PAGESIZE, assumed to be a power of two */
//p = (char *)(((int) p + psize-1) & ~(psize-1));

p = (char *) mmap(0, 1024, PROT_READ|PROT_WRITE,
                  MAP_ANONYMOUS|MAP_PRIVATE, 0, 0);
printf("p is %x\n", (unsigned) p); fflush(stdout);
```

Mapping `/dev/zero`

- Can use special properties of `/dev/zero` to create shared memory between <u>related</u> processes
- /dev/zero
  - On read, infinite source of 0
  - On write, data ignored
- When mapped:
  1. Unnamed region created with size given by second argument, rounded to nearest page size
  2. Region initialized to zero
  3. Processes can share this region if common ancestor specifies MAP_SHARED flag to mmap

Example*

- Transmit values between parent and child
- In a loop:
  1. Parent reads current value at addr; increments value
  2. Child reads current value at addr; increments value
- <u>With some synchronization</u>

* *From "Advanced Programming in the UNIX Environment", W. Richard Stevens*

- First, the synchronization:
  - Parent/child set up handlers for USR1, USR2
    - Handler sets value of global variable to 1
    - Block USR1, USR2 (no premature reception)
  - Child:
    - While signal var is zero wait; handler sets signal var to one; reset signal var to zero; update shared var; signal parent;
  - Parent:
    - Update shared var; signal child; unblock all; while signal var is zero, wait; update signal var

```
static void
sig_usr(int signo)
{
        sigflag = 1;
        return;
}
```

```
void
TELL_WAIT()
{
        if (signal(SIGUSR1, sig_usr) == SIG_ERR)
                err_sys("signal(SIGINT) error");
        if (signal(SIGUSR2, sig_usr) == SIG_ERR)
                err_sys("signal(SIGQUIT) error");

        sigemptyset(&zeromask);

        sigemptyset(&newmask);
        sigaddset(&newmask, SIGUSR1);
        sigaddset(&newmask, SIGUSR2);

/* block SIGUSR1 and SIGUSR2, and save current signal mask */

        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
                err_sys("SIG_BLOCK error");
}
```

```
void
TELL_PARENT(pid_t pid)
{
        kill(pid, SIGUSR2);        /* tell parent we're done */
}

void
WAIT_PARENT(void)
{
        while (sigflag == 0)
              sigsuspend(&zeromask);  /* and wait for parent */

        sigflag = 0;
                      /* reset signal mask to original value */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
}
```

```
void
TELL_CHILD(pid_t pid)
{
        kill(pid, SIGUSR1);          /* tell child we're done */
}

void
WAIT_CHILD(void)
{
        while (sigflag == 0)
              sigsuspend(&zeromask);  /* and wait for child */

        sigflag = 0;
                     /* reset signal mask to original value */
        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0)
                err_sys("SIG_SETMASK error");
}
```

- Now the shared value update
  - Map `sizeof(long)` bytes of `/dev/zero`
  - Parent prints current value; increments
  - Child prints current value; increments

```
 int main()
{
        int      fd, i, counter;
        pid_t   pid;
caddr_t area;

   if ( (fd = open("/dev/zero", O_RDWR)) < 0)
          err_sys("open error");
   if ( (area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
                     MAP_SHARED, fd, 0)) == (caddr_t) -1)
          err_sys("mmap error");
   close(fd);       /* can close /dev/zero now that it's mapped */

   TELL_WAIT();     /* Initialize synchronization */
```

```
if ( (pid = fork()) < 0) {
        err_sys("fork error");
} else if (pid > 0) {                          /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
                if ( (counter = update((long *) area)) != i)
                  err_quit("parent: expected %d got %d\n",i,counter);
                printf("<P> prior to update <%d>\n",counter);
                fflush(stdout);
                TELL_CHILD(pid);
                WAIT_CHILD();
        }
```

```
} else {                              /* child */
    for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();
            if ( (counter = update((long *) area)) != i)
                err_quit("child: expected %d, got %d\n",i,counter);
            printf("<C> prior to update <%d>\n",counter);
            fflush(stdout);
            TELL_PARENT(getppid());
    }
 }
 exit(0);

} /* end main */


static int
update(long *ptr)
{
        return( (*ptr)++ );     /* return value before increment
}
```

```
[jmayo@asimov advio]$ ./devzero
<P> prior to update <0>
<C> prior to update <1>
<P> prior to update <2>
<C> prior to update <3>
<P> prior to update <4>
<C> prior to update <5>
<P> prior to update <6>
<C> prior to update <7>
<P> prior to update <8>
<C> prior to update <9>
```

Final Example, mmap and mprotect

```
unsigned pagebase;
int i;

void
segvhandler()
{
  fprintf(stderr, "Segfaulted at %d\n", i);
  mprotect((void *)pagebase, 4096, PROT_READ | PROT_WRITE);
}

int
main(int argc, char *argv[])
{
    int fdin, fdout;
    char *src, *dst;
    struct stat statbuf;

    signal(SIGSEGV, segvhandler); // Get ready for the segfault.

. . .
```

```
    if ( (dst = mmap(0, statbuf.st_size, PROT_READ | PROT_WRITE,
     MAP_FILE | MAP_SHARED, fdout, 0)) == (caddr_t) -1) {
      fprintf(stderr,"mmap error for output");
      exit(2);
    }

    // Want confirmation before clobbering byte 13271 of destination.
    // Note granularity.
    printf("dst is %x\n", (unsigned) dst); fflush(stdout);
    pagebase = (((unsigned)dst + 13271) / 4096) * 4096;
    printf("pagebase is %x\n", pagebase); fflush(stdout);

    if (mprotect((void *) pagebase, 4096, PROT_NONE) < 0) {
      fprintf(stderr,"protect failure\n");
      exit(2);
    }
    for (i=0; i<statbuf.st_size; i++)
      memcpy(dst++, src++, 1); // does the file copy

    exit(0);
}
```

Locking Pages in Memory

- With virtual memory, pages may be swapped to disk
  - Time consuming
  - Remnants of process data left on disk
- Secure data may be revealed
  - Unencrypted keys, passwords
  - Will not save key from laptops with “suspend”, “hibernate”, etc.
- Real-time processes may miss deadlines
  - Real-time processes have time constraints
  - Commonly interact with physical world
    - E.g., controllers, simulations

NAME

mlock - disable paging for some parts of memory

SYNOPSIS

```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
```

DESCRIPTION

mlock disables paging for the memory in the range starting at addr with length len bytes. All pages which contain a part of the specified memory range are guaranteed be resident in RAM when the mlock system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked by munlock or munlockall, until the pages are unmapped via munmap, or until the process terminates or starts another program with exec. Child processes do not inherit page locks across a fork.

Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlock for the corresponding range or by munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

On POSIX systems on which mlock and munlock are available, _POSIX_MEM-LOCK_RANGE is defined in <unistd.h> and the value PAGESIZE from <limits.h> indicates the number of bytes per page.

NOTES

With the Linux system call, addr is automatically rounded down to the nearest page boundary. However, POSIX 1003.1-2001 allows an implementation to require that addr is page aligned, so portable applications should ensure this.

RETURN VALUE

On success, mlock returns zero. On error, -1 is returned, errno is set appropriately, and no changes are made to any locks in the address space of the process.

NAME

munlock - reenable paging for some parts of memory

SYNOPSIS

```
#include <sys/mman.h>

int munlock(const void *addr, size_t len);
```

DESCRIPTION

munlock reenables paging for the memory in the range starting at addr with length len bytes. All pages which contain a part of the specified memory range can after calling munlock be moved to external swap space again by the kernel.

Memory locks do not stack, i.e., pages which have been locked several times by calls to mlock or mlockall will be unlocked by a single call to munlock for the corresponding range or by munlockall. Pages which are mapped to several locations or by several processes stay locked into RAM as long as they are locked at least at one location or by at least one process.

On POSIX systems on which mlock and munlock are available, _POSIX_MEMLOCK_RANGE is defined in <unistd.h> and the value PAGESIZE from <limits.h> indicates the number of bytes per page.

RETURN VALUE

On success, munlock returns zero. On error, -1 is returned, errno is set appropriately, and no changes are made to any locks in the address space of the process.

NAME

sysconf - Get configuration information at runtime

SYNOPSIS

```
#include <unistd.h>

long sysconf(int name);
```

DESCRIPTION

POSIX allows an application to test at compile- or run-time whether certain options are supported, or what the value is of certain configurable constants or limits.

At compile time this is done by including <unistd.h> and/or <limits.h> and testing the value of certain macros.

At run time, one can ask for numerical values using the present function sysconf(). On can ask for numerical values that may depend on the filesystem a file is in using the calls fpathconf(3) and pathconf(3). One can ask for string values using confstr(3).

The values obtained from these functions are system configuration constants. They do not change during the lifetime of a process.

For options, typically, there is a constant _POSIX_FOO that may be defined in <unistd.h>. If it is undefined, one should ask at run-time. If it is defined to -1, then the option is not supported. If it is defined to 0, then relevant functions and headers exist, but one has to ask at runtime what degree of support is available. If it is defined to a value other than -1 or 0, then the option is supported. Usually the value (such as 200112L) indicates the year and month of the POSIX revision describing the option. Glibc uses the value 1 to indicate support as long as the POSIX revision has not been published yet. The sysconf() argument will be _SC_FOO. For a list of options, see posixoptions(7).

For variables or limits, typically, there is a constant _FOO, maybe defined in <limits.h>, or _POSIX_FOO, maybe defined in <unistd.h>. The constant will not be defined if the limit is unspecified. If the constant is defined, it gives a guaranteed value, and more might actually be supported. If an application wants to take advantage of values which may change between systems, a call to sysconf() can be made. The sysconf() argument will be _SC_FOO.

POSIX.1 VARIABLES

We give the name of the variable, the name of the sysconf() parameter used to inquire about its value, and a short description.

First, the POSIX.1 compatible values.

ARG_MAX - _SC_ARG_MAX

The maximum length of the arguments to the exec() family of functions. Must not be less than _POSIX_ARG_MAX (4096).

CHILD_MAX - _SC_CHILD_MAX

The max number of simultaneous processes per user id. Must not be less than _POSIX_CHILD_MAX (25).

HOST_NAME_MAX - _SC_HOST_NAME_MAX

Max length of a hostname, not including the final NUL, as returned by gethostname(2). Must not be less than _POSIX_HOST_NAME_MAX (255).

LOGIN_NAME_MAX - _SC_LOGIN_NAME_MAX

Maximum length of a login name, including the final NUL. Must not be less than _POSIX_LOGIN_NAME_MAX (9).

clock ticks - _SC_CLK_TCK

The number of clock ticks per second. The corresponding variable is obsolete. It was of course called CLK_TCK. (Note: the macro CLOCKS_PER_SEC does not give information: it must equal 1000000.)

OPEN_MAX - _SC_OPEN_MAX

The maximum number of files that a process can have open at any time. Must not be less than _POSIX_OPEN_MAX (20).

PAGESIZE - _SC_PAGESIZE

Size of a page in bytes. Must not be less than 1. (Some systems use PAGE_SIZE instead.)

…. Many more …

Example

- Lock page associated with character array into memory, then unlock
  - Compute address of start of page containing variable
  - Compute length of segment to lock
  - Lock

```c
#include <unistd.h>
#include <sys/mman.h>
#define DATA_SIZE 2048

lock_memory(char *addr, size_t size) {

    unsigned long page_offset, page_size;

    page_size = sysconf(_SC_PAGE_SIZE);

    page_offset = (unsigned long) addr % page_size;

    addr -= page_offset; /*Adjust addr to pg boundary */

    size += page_offset; /*Adjust size w/page_offset */

    return ( mlock(addr, size) ); /* Lock the memory */
}
```

```
unlock_memory(char *addr, size_t size) {

  unsigned long page_offset, page_size;

  page_size = sysconf(_SC_PAGE_SIZE);

  page_offset = (unsigned long) addr % page_size;

  addr -= page_offset;  /* Adjust addr to page boundary */

  size += page_offset;  /* Adjust size with page_offset */

  return ( munlock(addr, size) ); /* Unlock the memory */
}
```

```
main() {

    char data[DATA_SIZE];

    if ( lock_memory(data, DATA_SIZE) == -1 )

    perror("lock_memory"); /* Do work here */

    if ( unlock_memory(data, DATA_SIZE) == -1 )

    perror("unlock_memory");

}
```

Example

- Read password
  - Set terminal attributes so echo is off
  - Lock page (containing password) in memory
  - Read password
  - Use password
  - Clear password
  - Unlock page

```
int main(void) {
   struct termios ts, ots;
   char passbuf[1024];

   /* get and save current termios settings */
   tcgetattr(STDIN_FILENO, &ts);
   ots = ts;

   /* change and set new termios settings */
   ts.c_lflag &= ~ECHO;
   ts.c_lflag |= ECHONL;
   tcsetattr(STDIN_FILENO, TCSAFLUSH, &ts);

   /* paranoia: check that the settings took effect */
   tcgetattr(STDIN_FILENO, &ts);
   if (ts.c_lflag & ECHO) {
      fprintf(stderr, "Failed to turn off echo\n");
      tcsetattr(STDIN_FILENO, TCSANOW, &ots);
      exit(1);
   }
```

```
if ( lock_memory(passbuf, 1024) == -1 )
    perror("lock_memory");

   /* get and print the password */
   printf("enter password: ");
   fflush(stdout);
   fgets(passbuf, 1024, stdin);
   printf("read password: %s", passbuf);
   /* there was a terminal \n in passbuf */

  bzero(passbuf,1024);
  if ( unlock_memory(passbuf, 1024) == -1 )
    perror("unlock_memory");

   /* restore old termios settings */
   tcsetattr(STDIN_FILENO, TCSANOW, &ots);

   exit(0);
}
```