MAP_PRIVATE anonymous mappings

MAP_PRIVATE anonymous mappings are used to allocate blocks of process-private memory initialized to 0. We can use the /dev/zero technique to create a MAP_PRIVATE anonymous mapping as follows:

```
fd = open("/dev/zero", O_RDWR);
if (fd == -1)
errExit("open");
addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (addr == MAP_FAILED)
errExit("mmap");
```

The glibc implementation of malloc() uses MAP_PRIVATE anonymous mappings to allocate blocks of memory larger than MMAP_THRESHOLD bytes. This makes it possible to efficiently deallocate such blocks (via munmap()) if they are later given to free(). (It also reduces the possibility of memory fragmentation when repeatedly allocating and deallocating large blocks of memory.) MMAP_THRESHOLD is 128 kB by default, but this parameter is adjustable via the mallopt() library function.

MAP_SHARED anonymous mappings

A MAP_SHARED anonymous mapping allows related processes (e.g., parent and child) to share a region of memory without needing a corresponding mapped file.

MAP_SHARED anonymous mappings are available only with Linux 2.4 and later.

We can use the MAP_ANONYMOUS technique to create a MAP_SHARED anonymous mapping as follows:

```
addr = mmap(NULL, length, PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

```
if (addr == MAP_FAILED)
errExit("mmap");
```

If the above code is followed by a call to fork(), then, because the child produced by fork() inherits the mapping, both processes share the memory region.

Example program

The program in Listing 49-3 demonstrates the use of either MAP_ANONYMOUS or /dev/zero to share a mapped region between parent and child processes. The choice of technique is determined by whether USE_MAP_ANON is defined when compiling the program. The parent initializes an integer in the shared region to 1 prior to calling fork(). The child then increments the shared integer and exits, while the parent waits for the child to exit and then prints the value of the integer. When we run this program, we see the following:

```
$ ./anon_mmap
Child started, value = 1
In parent, value = 2
```


Listing 49-3: Sharing an anonymous mapping between parent and child processes

———————————————————————————— mmap/anon_mmap.c

```
#ifdef USE_MAP_ANON
#define _BSD_SOURCE /* Get MAP_ANONYMOUS definition */
#endif
#include <sys/wait.h>
#include <sys/mman.h>
#include <fcntl.h>
#include "tlpi_hdr.h"
int
```

```
main(int argc, char *argv[])
{
int *addr; /* Pointer to shared memory region */
#ifdef USE_MAP_ANON /* Use MAP_ANONYMOUS */
addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANONYMOUS, -1, 0);
if (addr == MAP_FAILED)
errExit("mmap");
#else /* Map /dev/zero */
int fd;
fd = open("/dev/zero", O_RDWR);
if (fd == -1)
errExit("open");
addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (addr == MAP_FAILED)
errExit("mmap");
if (close(fd) == -1) /* No longer needed */
errExit("close");
#endif
*addr = 1; /* Initialize integer in mapped region */
switch (fork()) { /* Parent and child share mapping */
case -1:
errExit("fork");
case 0: /* Child: increment shared integer and exit */
printf("Child started, value = %d\n", *addr);
(*addr)++;
```

```
if (munmap(addr, sizeof(int)) == -1)
errExit("munmap");
exit(EXIT_SUCCESS);
default: /* Parent: wait for child to terminate */
if (wait(NULL) == -1)
errExit("wait");
printf("In parent, value = %d\n", *addr);
```



```
if (munmap(addr, sizeof(int)) == -1)
errExit("munmap");
exit(EXIT_SUCCESS);
}
}
```

———————————————————————————— mmap/anon_mmap.c

49.8 Remapping a Mapped Region: mremap()

On most UNIX implementations, once a mapping has been created, its location and size can't be changed. However, Linux provides the (nonportable) mremap() system call, which permits such changes.

The old_address and old_size arguments specify the location and size of an existing mapping that we wish to expand or shrink. The address specified in old_address must be page-aligned, and is normally a value returned by a previous call to mmap(). The desired new size of the mapping is specified in new_size. The values specified in old_size and new_size are both rounded up to the next multiple of the system page size.

While carrying out the remapping, the kernel may relocate the mapping within the process's virtual address space. Whether or not this is permitted is controlled by the flags argument, which is a bit mask that may either be 0 or include the follow-

ing values:

MREMAP_MAYMOVE

If this flag is specified, then, as space requirements dictate, the kernel may relocate the mapping within the process's virtual address space. If this flag is not specified, and there is insufficient space to expand the mapping at the current location, then the error ENOMEM results.

MREMAP_FIXED (since Linux 2.4)

This flag can be used only in conjunction with MREMAP_MAYMOVE. It serves a purpose for mremap() that is analogous to that served by MAP_FIXED for mmap() (Section 49.10). If this flag is specified, then mremap() takes an additional argument, void *new_address, that specifies a page-aligned address to which the mapping should be moved. Any previous mapping in the address range specified by new_address and new_size is unmapped.

On success, mremap() returns the starting address of the mapping. Since (if the MREMAP_MAYMOVE flag is specified) this address may be different from the previous starting address, pointers into the region may cease to be valid. Therefore, applications that use mremap() should use only offsets (not absolute pointers) when referring to addresses in the mapped region (see Section 48.6).

```
#define _GNU_SOURCE
#include <sys/mman.h>
void *mremap(void *old_address, size_t old_size, size_t new_size, int flags, ...);
Returns starting address of remapped region on success,
or MAP_FAILED on error
```


On Linux, the realloc() function uses mremap() to efficiently reallocate large blocks of memory that malloc() previously allocated using mmap() MAP_ANONYMOUS.

(We mentioned this feature of the glibc malloc() implementation in Section 49.7.)
Using mremap() for this task makes it possible to avoid copying of bytes during
the reallocation

FLAG_PART_3: PDF_Expert_123} <!-- Flag hidden in body text -->