Feature Description

The new feature is a new syscall called ‘mmap’. The purpose of this syscall is to allow a user program to map a file directly into its virtual address space. It is primarily used for fast I/O to large files, or when sharing physical pages with another user program, could be used to achieve interprocess communication (IPC).

The plan is to attempt to implement the following:

- Direct mapping of file contents into memory, with a user defined file offset
- Lazy allocation of PTEs - pages further than the initial mapped page are not loaded initially, and the implementation will load them in from disk when requested via page faults. The purpose of this is to accommodate files larger than physical memory available, and for speed.
- Shared mapping - pages should be written to disk as they are unmapped. Additionally when another process ‘mmap’s the same file, share physical pages if possible.
- Private mapping - pages should not be written to disk as they are unmapped.
- User defined page protection - user programs can define the R,W,X protection flags, **but not** the other PTE flags.

Design Considerations

Storing VMAs

Xv6 does not come with a dynamic kernel memory allocator. Therefore we will have a static table of VMAs for each mapping. We will store this in a static table in the mmap module, instead of per process. The reasoning behind this is to easily search through mappings to find already allocated physical pages for MAP_SHARED mappings.

Synchronisation

Xv6 is quite primitive, and it would be hard to sync changes to a file to all of its MAP_SHARED mappings. Considering this, mappings are not guaranteed to have the latest version of a file, although when that page is first loaded, it will have fresh contents directly read from the file.

Sharing physical pages

There are multiple ways to do this, but I’ve opted to not use reference counting for physical pages and instead store the process owning each mapping in the VMA entries. This way, a simple walk through the mappings table can reveal the physical addresses of each mapping, should another process map the same file. The drawback of this is that it adds overhead when mapping and unmapping, as we need to look through the mappings table each time to see if we can free the physical page / share a page. Since the maximum number of VMAs are limited anyway due to a static table, this shouldn’t be too much of an issue.

Preservation of mappings on fork

In the current implementation of ‘fork’, a forked process will only copy the heap and it will not contain the mappings created by mmap. To solve this issue, additional code will be added to fork() to duplicate the mappings of the old process (and increment filesystem object reference counts), so that when accessed in the new process it will work.

Memory leak prevention on exit

A program may exit without unmapping all it’s previously mapped mmap regions. This will result in physical memory being blocked off (memory leak) everytime this happens, until no more memory is left. To resolve this, the code in ‘exit()’ in proc.c will be modified to force unmap all mapped regions. This has the added bonus of syncing changes to shared mappings back to their respective files.

Implementation Details

Structures, Types, and Defines

We need to define a few data structures and some constants for both the kernel and user programs to use. Firstly we will create a struct called ‘mmap_vma’, which contains metadata of an mmap virtual memory area. The struct will contain a pointer to the process owning it, the start virtual address, mapping length, page protection flags, mapping type, pointer to the file (and underlying inode), the inum of the file and the file offset for the mapping (see A.2 for code). We store the inum because you need to hold the inode lock to access it, so we access it once in the beginning and save it.

Next, we need to define some constants and flags for various uses. For user program use, we define MAP_SHARED, MAP_PRIVATE, PROT_READ, PROT_WRITE, PROT_EXEC (see A.2). These are just single bits in a uint64 type. The first two defines are the mapping types, shared and private mapping respectively. The last 3 defines are the page protection flags that will be used to set the PTE access protection flags. For kernel use, we define FAIL, PROT_MASK, PTE_DIRTY (see A.1). FAIL simply resolves to 0xFFFFFFFF, which is -1 as an integer and max unsigned int, this value will be used as return values to userspace to indicate failure. PROT_MASK is 0xE, or 0b1110. We use this mask for two purposes later, as a bitmask for the user set page prot flags (to avoid malicious setting of other PTE bits), and to check the validity of a page (since if R,W,X bits are not set, the page has not yet been loaded into memory.

Function Prototypes and System Call Entries

We need to update ‘defs.h’ with some of our future mmap functions (see B.2). There will mainly be 6 functions available to the rest of the OS. ‘mmapinit’ will initialise the mmap vma table lock, which will be called in main.c (see B.1). We also need a page fault handler as we are doing a lazy approach to mmap, which we define as ‘mmap_handlepgfault’, and takes a single parameter - the virtual address of the page that caused the fault. We have our main functionality of mmap ‘mmap_map’, this will take a file pointer, the file offset, map length, virtual address, page protection, mapping flags. This will later be called by the ‘mmap’ syscall. Next we have ‘mmap_unmap’, a function that unmaps a previously mapped mmap region, it takes the virtual address and length as a parameter. Finally we have ‘mmap_fork’ and ‘mmap_forceunmap’, which are called by fork and exit respectively (see B.4). The former will copy

mappings from an old proc to a new proc, and the latter will unmap all mmap regions on an exiting process.

We need to add syscalls as well for the user, so we create a new file, sysmmap.c (see A.3). We have our ‘sys_mmap’ and ‘sys_munmap’ system calls here, which take the params from userspace and call our mmap functions. We need to register these syscalls, so firstly we head to syscall.h and add system call numbers for each one (see B.6). We add the functions to the function call table in syscall.c (see B.5). Finally we define the syscalls in userspace by adding the function prototypes in user.h (see B.8) and then adding the entries to usys.pl to generate the proper ecall instructions and function signatures (see B.9).

Page Faults

Lastly, we need to prepare to catch page faults from userspace. We edit ‘usertrap’ to add an extra check before killing the process due to an unexpected scause. We pass the virtual address that caused the fault to our handler ‘mmap_handlepgfault’ and if it succeeds, we do not kill the process (see B.7).

Implementing mmap/munmap

Now that we have done the modifications to xv6 to lay out the foundations for ‘mmap’ and ‘munmap’, we can begin implementing mmap.

‘Mmap’ table locks - VMA entry allocation / deallocation

Our ‘mmapinit’ function (see A.1) simply calls ‘initlock’ for the lock of our global vma_table. We need a lock as this is a global table of VMAs, similar to kalloc’s global table of physical pages. When a process wants to allocate a table entry, we will perform checks to ensure it is free and mark it as taken. There will be an issue if two processes running in parallel passes the checks at the same time, and tries to both claim the same entry. This is why we need a lock for allocating an entry.

However we do not need to use the lock for freeing the entry, as we have guaranteed only one process will be able to use an allocated entry at a time. But it is critical that we do not mark the entry as being free before we have finished doing our unmapping tasks (like calling fileclose). This is why we explicitly define a function called ‘vma_free’ (see A.1) to ensure invalidating the entry is the last thing we do when freeing a VMA.

During VMA entry allocation, we acquire the table lock, look for the first entry in which the file pointer is null, set the file pointer and then release the lock. During deallocation, we set the process pointer to NULL (future support when we use that for entry valid check for anonymous mappings), call fileclose, and then set the file pointer to null (see vma_alloc and vma_free in A.1).

‘mmap()’ and ‘munmap()’

We start by doing some initial checks in ‘mmap_map’, the function called by the ‘mmap’ syscall. Firstly we check the f isn’t NULL, because currently we don’t support anonymous mappings and the file pointer is what we use to check for a VMA entry being valid. The file has to be an inode, not an unseekable device file, etc. Finally, we expect the starting virtual address to be null for now.

Then we find our process and allocate a vma entry for it from the global table. We set all the metadata other than the virtual address, which we have to find ourselves. An important step here is to perform a

bitwise and of the user provided prot flags with our mask PROT_MASK, to avoid malicious / accidental setting of other PTE bits other than R,W,X.

We find our VA from the top of our virtual address space, MAXVA minus 2 pages. The top 2 pages are used for the trampoline and trapframe so we have to start below them. We start from the top as the heap grows up, so using this method we delay intersecting with the heap for the longest time. We find a contiguous block large enough (rounding up the requested length to the next page aligned size), then set vma->va to starting address, or return FAIL if we intersected with the heap (see mmap_map in A.1 for full details). We read and save the file's unique inum, and then load the first page if it's a private mapping. If however it is a shared mapping, it will walk the table to find a similar VMA and try to find the first page from that, or NULL otherwise.

Finally, we call 'mappages' and reserve the length by setting all pages' PTE_V and PTE_U flags.

For unmapping, we first check that the given va is page aligned, fail otherwise. Then we free all the pages included in the given va range. If the page wasn't loaded yet, we just 'uvunmap' the pages with do_free param set to 0 (as the physical address isn't valid). If it was shared, we check if there were other mappings of that file. If there are, we can clear the PTE but not free the physical page, if not we can clear and free the physical page references. We can also use this opportunity to sync back to the disk. If it is private we clear and free the physical page since private mappings are not written back to disk (see mmap_unmap in A.1 for full details).

Page faults

Handling page faults is trivial (see mmap_handlepgfault in A.1). If it is a shared page, we check other similar mappings and attempt to get the physical address of their mapping for the requested offset. If we cannot find one or it is a private mapping, we just simply allocate a new physical page, read the file from the offset and set the PTE accordingly. Additionally if it is shared we update all similar mappings.

Process fork() and exit() helpers - (see mmap_fork and mmap_forceunmap in A.1)

Evaluation

This implementation of mmap and munmap is quite complete, other than the missing feature of anonymous mappings and virtual address start hints.

It shares physical pages on shared mappings, which is the most notable feature. Although it has the limitation of not being able to share physical pages if two mappings are of a different offset, something which I believe the original posix implementation can do. Mappings are also preserved across process forks, and properly unmapped on process exits.

A big limitation however is the global table, which means one process can saturate the global table, currently at 500 mappings per system. Using physical page reference counting and a per process mappings table would have definitely been better. Another limitation is not being able to split a VMA by freeing a region in the middle of an existing one.

Although slightly out of scope, an additional syscall to modify the page protection flags after mmap is called could have been useful. It would allow a process to modify it without unmapping and mapping the entire region again.

Appendix A - Source of mmap.c, mmap.h, sysmmap.c

A.1 mmap.c

```c
#include "types.h"
#include "param.h"
#include "riscv.h"
#include "spinlock.h"
#include "mmap.h"
#include "proc.h"
#include "defs.h"
#include "fs.h"
#include "sleeplock.h"
#include "file.h"

#define NULL 0
#define FAIL 0xFFFFFFFF

// 0b1110
#define PROT_MASK 0xE
#define PTE_DIRTY (1L << 7)

struct
{
  struct spinlock lk;       // VMA table lock
  struct mmap_vma tbl[NMMAP]; // array of VMAs
} vma_table;

// Why do we need a lock?
// Because if we get interrupted just before line 44 (vma->f = f),
// another process may try to use the same slot in the tbl.
//
// Editing the VMA once allocated without lock is not a problem
// since then we are guaranteed to be the only process that is using that slot

void mmapinit(void)
{
  initlock(&vma_table.lk, "vma_table");
}
```

```
// page fault handler
// returns:
// 0 - success
// 1 - mmap mapping not found
// 2 - error
int mmap_handlepgfault(uint64 va)
{
  // function must be called with a process context
  struct proc *p = myproc();
  if (p == NULL)
    return 2;

  // find corresponding mapping
  struct mmap_vma *vma;
  for (vma = vma_table.tbl; vma < vma_table.tbl + NMMAP; ++vma)
    if (p == vma->p && va >= vma->va && va < (vma->va + vma->len))
    { // mapping found!
      // find the corresponding page and offset
      uint64 pg_va = PGROUNDDOWN(va);
      uint offset = (pg_va - vma->va) + vma->f_offset;
      pte_t *pg_pte = walk(p->pagetable, pg_va, 0);

      // check other similar mappings
      struct mmap_vma *o_vma;
      if (vma->flags & MAP_SHARED)
        for (o_vma = vma_table.tbl; o_vma < vma_table.tbl + NMMAP; ++o_vma)
          if (o_vma->f && o_vma != vma && o_vma->f_inum == vma->f_inum && o_vma->f_offset
== vma->f_offset && o_vma->len >= vma->len)
          { // found another mapping, get the physical page from it and return
            pte_t *o_pte = walk(o_vma->p->pagetable, o_vma->va + (pg_va - vma->va), 0);
            if ((*o_pte & PROT_MASK) == 0) // nevermind, invalid mapping
              continue;
            uint64 pg_pa = PTE2PA(*o_pte);
            *pg_pte = PA2PTE(pg_pa) | vma->prot | PTE_V | PTE_U;

            // success
            return 0;
          }
```

```
        // get new physical page, rewrite pa on PTE
        uint64 pg_pa = (uint64)kalloc();
        *pg_pte = PA2PTE(pg_pa) | PTE_V | PTE_U;

        // read from file
        ilock(vma->f->ip);
        if (readi(vma->f->ip, 1, pg_va, offset, PGSIZE) < 0)
            return 2;
        iunlock(vma->f->ip);

        // set PTE prot flags in this mapping and all other similar mappings
        *pg_pte |= vma->prot;

        if (vma->flags & MAP_SHARED)
            for (o_vma = vma_table.tbl; o_vma < vma_table.tbl + NMMAP; ++o_vma)
                if (o_vma->f && o_vma != vma && o_vma->f_inum == vma->f_inum && o_vma->f_offset
== vma->f_offset && o_vma->len >= vma->len)
                    (*walk(o_vma->p->pagetable, o_vma->va + (pg_va - vma->va), 0)) = PA2PTE(pg_pa)
| o_vma->prot | PTE_U | PTE_V;

        // success
        return 0;
    }

  return 1;
}

// loop through the VMA table, find a free slot
static struct mmap_vma *vma_alloc(struct file *f)
{
  acquire(&vma_table.lk);
  struct mmap_vma *vma;
  for (vma = vma_table.tbl; vma < vma_table.tbl + NMMAP; ++vma) // search the table
    if (vma->f == NULL)
    { // found an empty slot!
      vma->f = f;
      release(&vma_table.lk);
      return vma;
    }
```

```
  release(&vma_table.lk);
  return NULL;
}

// free a VMA entry
//
// explicitly defined as a function as to emphasize
// the entry should no longer be used after setting file
// to NULL, as it is up for grabs by other mmap operations
static inline void vma_free(struct mmap_vma *vma)
{
  vma->p = NULL;
  fileclose(vma->f);
  vma->f = NULL;
}

// get the first page of a MAP_PRIVATE mapping
// returns physical address of page
static void *get_priv_firstpg(struct file *f, uint offset, uint length)
{
  void *firstpg = kalloc();
  ilock(f->ip);
  readi(f->ip, 0, (uint64)firstpg, offset, PGSIZE);
  iunlock(f->ip);
  return firstpg;
}

// find a similar mapping for MAP_SHARED, otherwise NULL
static void *find_shared_firstpg(struct file *f, struct mmap_vma *vma)
{
  struct mmap_vma *othervma;
  for (othervma = vma_table.tbl; othervma < vma_table.tbl + NMMAP; ++othervma)
    if (othervma->f && vma != othervma && othervma->f_inum == vma->f_inum &&
othervma->f_offset == vma->f_offset && othervma->len >= vma->len) // same file & offset
    {
      // check valid page
      if ((*walk(othervma->p->pagetable, othervma->va, 0) & PROT_MASK) == 0)
        continue;
      return (void *)walkaddr(othervma->p->pagetable, othervma->va);
```

```
    }
  return NULL;
}

// maps a file to memory
// returns the virtual address of where it is mapped
uint64 mmap_map(struct file *f, uint offset, uint length, uint64 start_va, int prot, int flags)
{
  if (f == NULL) // anonymous mappings not yet supported
    return FAIL;

  if (f->type != FD_INODE) // hello user seekable inodes only pls
    return FAIL;

  if (start_va != NULL) // start addr hints not yet supported
    return FAIL;

  // start by allocating a vma struct, fail if none free
  struct mmap_vma *vma;
  if ((vma = vma_alloc(f)) == NULL)
    return FAIL;

  // find our proc, fail if no proc?
  struct proc *p = myproc();
  if (p == NULL)
    return FAIL;

  vma->p = p;
  vma->f_offset = offset;
  vma->len = length;
  vma->prot = (prot & PROT_MASK); // prevent user from setting PTE flags other than R,W,X
  if (vma->prot == 0)
    vma->prot = PTE_R; // default readonly;
  vma->flags = flags;

  // find next available address
  //
  // From Figure 2.3 in the xv6-riscv book, the heap grows up...
  // ...all the way until it hits the trapframe. Using this...
```

```
// ...information, we can just go down from the trapframe,...
// ...growing downwards.

uint64 top_va = MAXVA - (PGSIZE * 2); // top - trampoline - trapframe
uint64 bottom_va = top_va;
const uint64 len = PGROUNDUP(length); // round up, can't lose bytes

// loop until we find enough pages
while (top_va - bottom_va != len)
{
    // find free top_va (check valid flag)
    pte_t *top_pte = walk(p->pagetable, top_va, 0);
    if ((*top_pte & PTE_V) != 0)
    {
        top_va -= PGSIZE;
        bottom_va = top_va;
        continue;
    }

    // keep moving bottom_va down until we reach the required length
    bottom_va -= PGSIZE;

    // have we intersected with heap?
    const uint64 brk = p->sz;
    if (bottom_va <= brk)
        return FAIL;

    // is bottom_va free? if not we reset the search
    pte_t *bottom_pte = walk(p->pagetable, bottom_va, 0);
    if ((*bottom_pte & PTE_V) != 0)
    {
        bottom_va -= PGSIZE;
        top_va = bottom_va;
    }
}

// we've found a valid va range
vma->va = bottom_va;
```

```
  // dup file ref and set inode num
  filedup(f);
  ilock(f->ip);
  vma->f_inum = f->ip->inum;
  iunlock(f->ip);

  void *firstpg = NULL;

  // get private page
  if (flags & MAP_PRIVATE)
    firstpg = get_priv_firstpg(f, offset, length);

  // find a similar mapping, otherwise leave as NULL
  if (flags & MAP_SHARED)
    firstpg = find_shared_firstpg(f, vma);

  // leave R,W,X unset for lazy mmap
  if (mappages(p->pagetable, bottom_va, len, (uint64)firstpg, PTE_V | PTE_U) != 0)
    return FAIL;

  // set the first PTE to user perms, if firstpg is valid
  if (firstpg != NULL)
    (*walk(p->pagetable, bottom_va, 0)) |= vma->prot;

  // return the starting virtual address
  return bottom_va;
}

// frees a shared page, sync content to disk
// frees physical page if no other references to it
//
// va must be page aligned
static void free_shared_page(struct mmap_vma *vma, uint64 va)
{
  // we can use this opportunity to sync changes back to disk
  pte_t *pte = walk(vma->p->pagetable, va, 0);
  if ((*pte & PTE_DIRTY))
  { // only write back to disk if the page is dirty (has been written to)
    ilock(vma->f->ip);
```

```
      writei(vma->f->ip, 1, va, (va - vma->va) + vma->f_offset, PGSIZE);
      iunlock(vma->f->ip);
    }

    // now we check other similar mappings
    // starting to regret not using physical page ref counting
    // this looks a little slow...
    struct mmap_vma *o_vma;
    for (o_vma = vma_table.tbl; o_vma < vma_table.tbl + NMMAP; ++o_vma)
      if (o_vma->f && vma != o_vma && o_vma->f_inum == vma->f_inum && o_vma->f_offset ==
vma->f_offset && o_vma->len >= vma->len) // same file & offset
      {
        uint64 va_offset = va - vma->va;
        if ((*walk(o_vma->p->pagetable, o_vma->va + va_offset, 0) & PROT_MASK) != 0)
        { // do not free the physical page, there is another ref to it
          uvmunmap(vma->p->pagetable, va, 1, 0);
          return;
        }
      }

    // if we reach here, it means there are no other referenes
    // to the physical page, unmap the VMA and free the PA
    uvmunmap(vma->p->pagetable, va, 1, 1);
}

int mmap_unmap(uint64 va, uint length)
{
    if (va % PGSIZE != 0) // addr must be a multiple of PGSIZE
      return -1;

    // find mapping
    struct mmap_vma *vma;
    for (vma = vma_table.tbl; vma < vma_table.tbl + NMMAP; ++vma)
      if (vma->p == myproc() && va >= vma->va && va < (vma->va + vma->len))
      { // found mapping
        if (va != vma->va && PGROUNDUP((va + length)) != PGROUNDUP(vma->va + vma->len))
          return -1; // cannot punch a hole in mapping / end va out of range

        const uint pages = PGROUNDUP(length) / PGSIZE;
```

```
        uint pg;
        for (pg = 0; pg < pages; ++pg)
        {
            uint64 c_va = va + (PGSIZE * pg);
            pte_t *c_pte = walk(vma->p->pagetable, c_va, 0);

            if ((*c_pte & PROT_MASK) == 0)
            { // page not yet mapped anyway
                uvmunmap(vma->p->pagetable, c_va, 1, 0);
                continue;
            }

            if (vma->flags & MAP_SHARED)
            { // special free for shared pages
                free_shared_page(vma, c_va);
                continue;
            }

            // if we reach here, it's a private mapped page
            // just free the physical page and clear the PTE
            uvmunmap(vma->p->pagetable, c_va, 1, 1);
        }

        if (pages * PGSIZE == PGROUNDUP(vma->len))
        {
            vma_free(vma);
            return 0;
        }

        // regardless, we freed pages * PGSIZE bytes
        vma->len -= pages * PGSIZE;
        if (va == vma->va) // beginning freed, move start va
            vma->va += pages * PGSIZE;

        return 0;
    }

  return -1;
}
```

```
// force unmap all mappings for a process
void mmap_forceunmap(struct proc *p)
{
  struct mmap_vma *vma;
  for (vma = vma_table.tbl; vma < vma_table.tbl + NMMAP; ++vma)
    if (vma->p == p && vma->f)
      mmap_unmap(vma->va, vma->len);
}

static void vma_copy(struct mmap_vma *src, struct mmap_vma *dst)
{
  dst->f = src->f;
  dst->f_inum = src->f_inum;
  dst->f_offset = src->f_offset;
  dst->flags = src->flags;
  dst->len = src->len;
  dst->prot = src->prot;
  dst->va = src->va;
}

// copies mappings from p to np
int mmap_fork(struct proc *p, struct proc *np)
{
  struct mmap_vma *vma;
  for (vma = vma_table.tbl; vma < vma_table.tbl + NMMAP; ++vma)
    if (vma->p == p && vma->f)
    { // found a mapping belonging to old proc
      struct mmap_vma *n_vma = vma_alloc(vma->f);
      if (n_vma == NULL)
        return -1;
      n_vma->p = np;

      // copy VMA
      vma_copy(vma, n_vma);

      // new reference to file
      filedup(n_vma->f);
```

```
        // map our pages, leave all pages R,W,X unset for
        // lazy mmap for the new file as well :D
        if (mappages(np->pagetable, n_vma->va, n_vma->len, (uint64)0, PTE_V | PTE_U) != 0)
            return -1;
      }
  return 0;
}
```

A.2 mmap.h

```
#define MAP_SHARED  (1L << 0)
#define MAP_PRIVATE (1L << 1)

#define PROT_READ   (1L << 1)
#define PROT_WRITE  (1L << 2)
#define PROT_EXEC   (1L << 3)

struct mmap_vma
{
  struct proc *p;         // process owning this VMA
  uint64 va;              // virtual addr
  uint64 len;             // vma length
  int prot;               // page prot flags
  int flags;              // mapping flags
  struct file *f;         // file
  uint f_inum;            // file inode num
  uint f_offset;          // file offset
};
```

A.3 sysmmap.c

```
#include "types.h"
#include "riscv.h"
#include "defs.h"
#include "date.h"
#include "param.h"
#include "memlayout.h"
#include "spinlock.h"
#include "proc.h"
```

```
uint64 sys_mmap(void)
{
  // we really be maxing out max syscall args
  int fd, prot, flags, off, len;
  uint64 start_va;

  if (argint(0, &fd) < 0 ||
      argint(1, &off) < 0 ||
      argint(2, &len) < 0 ||
      argaddr(3, &start_va) < 0 ||
      argint(4, &prot) < 0 ||
      argint(5, &flags) < 0)
      return -1; // same as 0xffffffff

  return mmap_map(myproc()->ofile[fd], off, len, start_va, prot, flags);
}

uint64 sys_munmap(void)
{
  uint64 va;
  int len;

  if (argaddr(0, &va) < 0 || argint(1, &len) < 0)
      return -1;

  // writing to fs, must do it in a transaction
  // or else kernel panic :shrug:
  begin_op();
  int retval = mmap_unmap(va, len);
  end_op();
  return retval;
}
```

Appendix B - Source changes (diffs) from xv6 origin as of commit 077323a8f0b3440fcc3d082096a2d83fe5461d70

B.1 main.c

```
@@ -27,6 +27,7 @@ main()
   binit();         // buffer cache
   iinit();         // inode cache
   fileinit();      // file table
+  mmapinit();      // mmap VMA table
   virtio_disk_init(); // emulated hard disk
   userinit();      // first user process
   __sync_synchronize();
```

B.2 defs.h

```
@@ -54,6 +54,14 @@ void            stati(struct inode*, struct stat*);
int             writei(struct inode*, int, uint64, uint, uint);
void            itrunc(struct inode*);
+// mmap.c
+void            mmapinit(void);
+int             mmap_handlepgfault(uint64);
+uint64          mmap_map(struct file*, uint, uint, uint64, int, int);
+int             mmap_unmap(uint64, uint);
+int             mmap_fork(struct proc*, struct proc*);
+void            mmap_forceunmap(struct proc*);
+
// ramdisk.c
void            ramdiskinit(void);
void            ramdiskintr(void);
```

B.3 param.h

```
@@ -2,6 +2,7 @@
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
#define NFILE       100  // open files per system
+#define NMMAP       500  // number of mmap VMAs per system
#define NINODE       50  // maximum number of active i-nodes
#define NDEV         10  // maximum major device number
```

```
#define ROOTDEV       1  // device number of file system root disk
```

B.4 proc.c

```
@@ -289,6 +289,12 @@ fork(void)
  }
  np->sz = p->sz;
+  if(mmap_fork(p, np) < 0){
+    freeproc(np);
+    release(&np->lock);
+    return -1;
+  }
+
  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);
@@ -344,6 +350,9 @@ exit(int status)
  if(p == initproc)
    panic("init exiting");
+  // unmap all mmap mappings
+  mmap_forceunmap(p);
+
  // Close all open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
```

B.5 syscall.c

```
@@ -95,6 +95,8 @@ extern uint64 sys_kill(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_mknod(void);
+extern uint64 sys_mmap(void);
+extern uint64 sys_munmap(void);
extern uint64 sys_open(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
@@ -126,6 +128,8 @@ static uint64 (*syscalls[])(void) = {
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
+[SYS_mmap]    sys_mmap,
+[SYS_munmap]  sys_munmap,
```

```
[SYS_close]   sys_close,
};
```

B.6 syscall.h

```
@@ -19,4 +19,6 @@
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
-#define SYS_close  21
+#define SYS_mmap   21
+#define SYS_munmap 22
+#define SYS_close  23
```

B.7 trap.c

```
@@ -67,6 +67,10 @@ usertrap(void)
    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
+  } else if (mmap_handlepgfault(r_stval()) == 0){
+    // ok
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
```

B.8 user.h

```
@@ -13,6 +13,8 @@ int kill(int);
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, short, short);
+void* mmap(int, int, int, void*, int, int);
+int munmap(void*, int);
int unlink(const char*);
int fstat(int fd, struct stat*);
int link(const char*, const char*);
```

B.9 usys.pl

```
@@ -26,6 +26,8 @@ entry("kill");
entry("exec");
entry("open");
entry("mknod");
+entry("mmap");
+entry("munmap");
entry("unlink");
entry("fstat");
entry("link");
```