1 Una implementación de memoria compartida para xv6

En este trabajo se describen modificaciones en el código fuente del sistema operativo xv6 para agregar una llamada al sistema con la que se tiene memoria compartida entre procesos de xv6. Las modificaciones de código fuente descritas en este documento están basadas en la información obtenida de la página

`https://github.com/jeffallen/xv6/commit/ca9cd826afb140a469a5dc9e9445a534add596b8`

consultada en fecha 27 de febrero de 2020. El autor de este documento, no pretende ninguna autoria de las modificaciones de código fuente aquí presentadas, y las incluye únicamente con propósitos académicos en el contexto de un curso de Sistemas operativos en tiempo real, impartido en la UPIITA del IPN, México, durante el semestre enero-julio de 2020.

1.1 Archivo `defs.h`

```
  defs.h
  @@ -116,6 +116,8 @@ void            userinit(void);
  int             wait(void);
  void            wakeup(void*);
  void            yield(void);
+ void            sharedinit(void);
+ struct shared * sharedalloc(void);
```

1.2 Archivo `main.c`

```
   main.c
    @@ -28,6 +28,7 @@ main(void)
    consoleinit();   // I/O devices & their interrupts
    uartinit();      // serial port
    pinit();         // process table
+   sharedinit();    // shared memory records
    tvinit();        // trap vectors
    binit();         // buffer cache
    fileinit();      // file table
```

1.3 Archivo proc.c

```
  proc.c
    @@ -12,6 +12,11 @@ struct {
      struct proc proc[NPROC];
    } ptable;

+   struct {
+   struct spinlock lock;
+   struct shared shared[NSHARED];
+   } sharedtable;
+
    static struct proc *initproc;

    int nextpid = 1;
    @@ -122,6 +127,49 @@ growproc(int n)
      return 0;
  }

+ void
+ sharedinit()
+ {
+   initlock(&sharedtable.lock, "shared");
+ }
+
+ struct shared *
+ sharedalloc()
+ {
+   int i;
+   void *mem;
+   struct shared *sh;
+
+ acquire(&sharedtable.lock);
+ for (i = 0; i < NSHARED; i++) {
```

```
+   if (sharedtable.shared[i].refcount == 0) {
+     // found a free one
+     break;
+   }
+ }
+
+   // no free shared records left
+   if (i == NSHARED) {
+     release(&sharedtable.lock);
+     return 0;
+   }
+
+   mem = kalloc();
+   if (!mem) {
+     release(&sharedtable.lock);
+     return 0;
+   }
+
+   sh = &sharedtable.shared[i];
+   sh->refcount = 1;
+   sh->page = mem;
+   release(&sharedtable.lock);
+
+   return sh;
+ }
+
+ extern void mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
+
  // Create a new process copying p as the parent.
  // Sets up stack to return as if from system call.
  // Caller must set state of returned proc to RUNNABLE.
  @@ -153,6 +201,15 @@ fork(void)
      if(proc->ofile[i])
        np->ofile[i] = filedup(proc->ofile[i]);
    np->cwd = idup(proc->cwd);
+
+   // shared mem
+   if (proc->shared) {
```

```
+     acquire(&sharedtable.lock);
+     proc->shared->refcount++;
+     mappages(np->pgdir, (char *)SHARED_V, PGSIZE, v2p(proc->shared->page),
+     PTE_W|PTE_U);
+     np->shared = proc->shared;
+     release(&sharedtable.lock);
+   }

    pid = np->pid;
    np->state = RUNNABLE;
  @@ -230,6 +287,15 @@ wait(void)
          p->parent = 0;
          p->name[0] = 0;
          p->killed = 0;
+         if (p->shared) {
+           acquire(&sharedtable.lock);
+           p->shared->refcount--;
+           if (p->shared->refcount == 0) {
+             kfree(p->shared->page);
+         }
+         release(&sharedtable.lock);
+         p->shared = 0;
+       }
        release(&ptable.lock);
        return pid;
      }
```

1.4 Archivo `proc.h`

```
   13  proc.h
  @@ -51,6 +51,17 @@ struct context {

  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

+ // The virtual address where shared memory appears, if requested
+ #define SHARED_V 0x70000000
+
+ // the maximum number of shared pages in the system
+ #define NSHARED 10
+
+ struct shared {
+   int refcount;
+   void *page;
+ };
+
  // Per-process state
  struct proc {
    uint sz;                     // Size of process memory (bytes)
  @@ -65,6 +76,7 @@ struct proc {
    int killed;                  // If non-zero, have been killed
    struct file *ofile[NOFILE];  // Open files
    struct inode *cwd;           // Current directory
+   struct shared *shared;       // Shared memory record (0 -> none)
    char name[16];               // Process name (debugging)
  };

  @@ -73,3 +85,4 @@ struct proc {
  //   original data and bss
  //   fixed-size stack
  //   expandable heap
+ //   (optionally) fixed-sized shared mem segment, 1 page @ 0x7000000
```

1.5 Archivo syscall.c

```
   2  syscall.c
  @@ -98,6 +98,7 @@ extern int sys_unlink(void);
  extern int sys_wait(void);
  extern int sys_write(void);
  extern int sys_uptime(void);
+ extern int sys_shared(void);

  static int (*syscalls[])(void) = {
  [SYS_fork]    sys_fork,
  @@ -121,6 +122,7 @@ static int (*syscalls[])(void) = {
  [SYS_link]    sys_link,
  [SYS_mkdir]   sys_mkdir,
  [SYS_close]   sys_close,
+ [SYS_shared]  sys_shared,
  };

  void
```

1.6 Archivo syscall.h

```
   2  syscall.h
  @@ -21,3 +21,5 @@
  #define SYS_link   19
  #define SYS_mkdir  20
  #define SYS_close  21
+
+ #define SYS_shared 22
```

1.7 Archivo sysproc.c

```
   24  sysproc.c
  @@ -88,3 +88,27 @@ sys_uptime(void)
    release(&tickslock);
    return xticks;
  }
+
+ extern void mappages(pde_t *pgdir,void *va,uint size,uint pa,int perm);
+
+ // arrange for this process to have a shared memory page
+ // that will be shared with all children
+ int
+ sys_shared(void)
+ {
+   struct shared *sh;
+
+   // if there's already a shared page, return now
+   if (proc->shared) {
+     return SHARED_V;
+   }
+
+   sh = sharedalloc();
+   if (sh) {
+     proc->shared = sh;
+     mappages(proc->pgdir,(char*)SHARED_V,PGSIZE,v2p(sh->page),
+     PTE_W|PTE_U);
+     return SHARED_V;
+   } else {
+     return 0;
+   }
+ }
```

1.8 Archivo user.h

```
   1  user.h
  @@ -22,6 +22,7 @@ int getpid(void);
  char* sbrk(int);
  int sleep(int);
  int uptime(void);
+ int shared(void);
```

1.9 Archivo usertests.c

```
   38  usertests.c
  @@ -1597,6 +1597,43 @@ fsfull()
    printf(1, "fsfull test finished\n");
  }

+ void
+ sharedtest()
+ {
+   int i;
+   char *sh;
+
+   sh = (char *)shared();
+   if (!sh) {
+     printf(2, "shared: fail\n");
+     exit();
+ }
+
+   if (fork() == 0) {
+     printf(1, "pid child: %d\n", getpid());
+     strcpy(sh, "hello world");
+     exit();
+   } else {
+     int pid;
+     printf(1, "pid parent: %d\n", getpid());
+     while (i < 10000) {
+       if (strcmp(sh, "hello world") == 0) {
+         printf(2, "shared: ok after %d checks\n", i);
+         break;
+       }
+       i++;
+       // yield and let the other guy run
+       sleep(0);
+     }
+     if (i == 10000) {
+       printf(2, "shared: not ok after %d checks\n", i);
+     }
+
+     pid = wait();
+     printf(1, "parent reaped pid %d\n", pid);
+   }
+ }
+
  unsigned long randstate = 1;
  unsigned int
  rand()
  @@ -1650,6 +1687,7 @@ main(int argc, char *argv[])
    bigdir(); // slow

    exectest();
+   sharedtest();

    exit();
  }
```

1.10 Archivo usys.S

```
  1  usys.S
 @@ -29,3 +29,4 @@ SYSCALL(getpid)
 SYSCALL(sbrk)
 SYSCALL(sleep)
 SYSCALL(uptime)
+ SYSCALL(shared)
```

1.11 Archivo vm.c

```
  6  vm.c
 @@ -67,7 +67,7 @@ walkpgdir(pde_t *pgdir, const void *va, int alloc)
 // Create PTEs for virtual addresses starting at va that refer to
 // physical addresses starting at pa. va and size might not
 // be page-aligned.
- static int
+ int
 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
 {
   char *a, *last;
 @@ -223,7 +223,7 @@ allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
   char *mem;
   uint a;

 -  if(newsz >= KERNBASE)
 +  if(newsz >= SHARED_V)
     return 0;
   if(newsz < oldsz)
     return oldsz;
 @@ -281,7 +281,7 @@ freevm(pde_t *pgdir)

   if(pgdir == 0)
     panic("freevm: no pgdir");
 -  deallocuvm(pgdir, KERNBASE, 0);
 +  deallocuvm(pgdir, SHARED_V, 0);
   for(i = 0; i < NPDENTRIES; i++){
     if(pgdir[i] & PTE_P){
        char * v = p2v(PTE_ADDR(pgdir[i]));
```

2 Modificaciones de los archivos fuente de xv6

Siguiendo las indicaciones anteriores, se modificaron los archivos
`defs.h`
`main.c`
`proc.c`
`proc.h`
`syscall.c`
`syscall.h`
`sysproc.c`
`user.h`
`usertests.c`
`usys.S`
`vm.c`
de la siguiente forma

2.1 Modificaciones del archivo `defs.h`

```
#define SOTR_20202_1
struct buf;
struct context;
struct file;
//...
void            wakeup(void*);
void            yield(void);
#ifdef SOTR_20202_1
void            sharedinit(void);
struct shared * sharedalloc(void);
#endif /*SOTR_20202_1*/

// swtch.S
void            swtch(struct context**, struct context*);
```

2.2 Modificaciones del archivo `main.c`

```
  uartinit();      // serial port
  pinit();         // process table
#ifdef SOTR_20202_1
  sharedinit();     // shared memory records
#endif /*SOTR_20202_1*/
  tvinit();        // trap vectors
  binit();         // buffer cache
```

2.3 Modificaciones del archivo `proc.c`

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

#ifdef SOTR_20202_1
struct {
  struct spinlock lock;
  struct shared shared[NSHARED];
} sharedtable;
#endif /*SOTR_20202_1*/

static struct proc *initproc;

int nextpid = 1;

//...
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}

#ifdef SOTR_20202_1
void
sharedinit()
{
  initlock(&sharedtable.lock,"shared");
}/*end void sharedinit()*/

struct shared *
sharedalloc()
{
  int i;
  void *mem;
  struct shared *sh;

  acquire(&sharedtable.lock);
  for(i=0;i<NSHARED;i++){
    if(sharedtable.shared[i].refcount==0){
      // found a free one
      break;
    }
  }

  // if no free shared records left
  if(i==NSHARED){
    release(&sharedtable.lock);
    return 0;
  }

  mem=kalloc();
  if(!mem){
    release(&sharedtable.lock);
    return 0;
  }

  sh=&sharedtable.shared[i];
  sh->refcount=1;
  sh->page=mem;
  release(&sharedtable.lock);
//cprintf("initialization of shared memory has completed\n");

  return sh;
}/*end struct shared *sharedalloc()*/
```

```
extern void mappages(pde_t *pgdir,void *va,uint size,uint pa,int perm);

#endif /*SOTR_20202_1*/

// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);
```

```
#ifdef SOTR_20202_1
  // shared mem
  if(curproc->shared){
    acquire(&sharedtable.lock);
    curproc->shared->refcount++;
    mappages(np->pgdir,(char*)SHARED_V,PGSIZE,V2P(curproc->shared->page),
    PTE_W|PTE_U);
    np->shared=curproc->shared;
    release(&sharedtable.lock);
  }
#endif /*SOTR_20202_1*/

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}

//...
// Wait for a child process to exit and return its pid.
// Return -1 if this process has no children.
int
wait(void)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        freevm(p->pgdir);
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
#ifdef SOTR_20202_1
        if(p->shared){
          acquire(&sharedtable.lock);
          p->shared->refcount--;
          if(p->shared->refcount==0){
            kfree(p->shared->page);
          }
          release(&sharedtable.lock);
          p->shared=0;
        }
#endif /*SOTR_20202_1*/
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }
    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

2.4 Modificaciones del archivo `proc.h`

```
#define SOTR_20202_1
// Per-CPU state
struct cpu {
  uchar apicid;                // Local APIC ID
//...
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

#ifdef SOTR_20202_1
// The virtual address where shared memory appears, if requested
#define SHARED_V 0x70000000

// The maximun number of shared pages in the system
#define NSHARED 10

struct shared {
  int refcount;
  void *page;
};
#endif /*SOTR_20202_1*/

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
#ifdef SOTR_20202_1
  struct shared *shared;       // Shared memory record (0 -> none)
#endif /*SOTR_20202_1*/
  char name[16];               // Process name (debugging)
};

// Process memory is laid out contiguously, low addresses first:
//   text
//   original data and bss
//   fixed-size stack
//   expandable heap
#ifdef SOTR_20202_1
//   (optionally) fixed-sized shared mem segment, 1 page @ 0x70000000
#endif /*SOTR_20202_1*/
```

2.5 Modificaciones del archivo `syscall.c`

```
extern int sys_uptime(void);
extern int sys_dummy(void);
#ifdef SOTR_20202_1
extern int sys_shared(void);
#endif /*SOTR_20202_1*/

static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_dummy]   sys_dummy,
#ifdef SOTR_20202_1
[SYS_shared]  sys_shared,
#endif /*SOTR_20202_1*/
};
```

2.6 Modificaciones del archivo `syscall.h`

```
#define SYS_close  21
#define SYS_dummy  22
#ifdef SOTR_20202_1
#define SYS_shared  23
#endif /*SOTR_20202_1*/
```

2.7 Modificaciones del archivo `sysproc.c`

```
// return how many clock tick interrupts have occurred
// since start.
int
sys_uptime(void)
{
  uint xticks;

  acquire(&tickslock);
  xticks = ticks;
  release(&tickslock);
  return xticks;
}

#ifdef SOTR_20202_1
extern void mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

// arrange for this process to have a shared memory page
// that will be shared with all children
int
sys_shared(void)
{
  struct shared *sh;
  struct proc *proc = myproc();

  // if there's already a shared page, return now
  if (proc->shared) {
    return SHARED_V;
  }

  sh = sharedalloc();
  if (sh) {
    proc->shared = sh;
    mappages(proc->pgdir, (char *)SHARED_V, PGSIZE, V2P(sh->page), PTE_W|PTE_U);
    return SHARED_V;
  } else {
    return 0;
  }
}
#endif /*SOTR_20202_1*/
```

2.8 Modificaciones del archivo `user.h`

```
#define SOTR_20202_1
struct stat;
struct rtcdate;
//...
int uptime(void);
```

```
char *dummy(int);
#ifdef SOTR_20202_1
int shared(void);
#endif /*SOTR_20202_1*/

// ulib.c
int stat(const char*, struct stat*);
```

2.9 Modificaciones del archivo `usertests.c`

```
    unlink(name);
    nfiles--;
  }

  printf(1, "fsfull test finished\n");
}

#ifdef SOTR_20202_1
void
sharedtest()
{
  int i=0;
  char *sh;

  sh = (char *)shared();
  if (!sh) {
    printf(2, "shared: fail\n");
    exit();
  }

  if (fork() == 0) {
    printf(1, "pid child: %d\n", getpid());
    strcpy(sh, "hello world");
    exit();
  } else {
    int pid;
    printf(1, "pid parent: %d\n", getpid());
    while (i < 10000) {
      if (strcmp(sh, "hello world") == 0) {
        printf(2, "shared: ok after %d checks\n", i);
        break;
      }
      i++;
      // yield and let the other guy run
      sleep(0);
    }
    if (i == 10000) {
      printf(2, "shared: not ok after %d checks\n", i);
    }

    pid = wait();
    printf(1, "parent reaped pid %d\n", pid);
  }
}
#endif /*SOTR_20202_1*/

void
uio()
{
  #define RTC_ADDR 0x70
  #define RTC_DATA 0x71

  ushort port = 0;
//...
int
main(int argc, char *argv[])
{
  printf(1, "usertests starting\n");

  if(open("usertests.ran", 0) >= 0){
    printf(1, "already ran user tests -- rebuild fs.img\n");
    exit();
  }
  close(open("usertests.ran", O_CREATE));

#ifndef SOTR_20202_1
```

```
argptest();
createdelete();
linkunlink();
concreate();
fourfiles();
sharedfd();

bigargtest();
bigwrite();
bigargtest();
bsstest();
sbrktest();
validatetest();

opentest();
writetest();
writetest1();
createtest();

openiputtest();
exitiputtest();
iputtest();

mem();
pipe1();
preempt();
exitwait();

rmdot();
fourteen();
bigfile();
subdir();
linktest();
unlinkread();
dirfile();
iref();
forktest();
bigdir(); // slow
```

```
  uio();

  exectest();
#else
  sharedtest();
#endif /*SOTR_20202_1*/

  exit();
}
```

2.10 Modificaciones del archivo usys.S

```
#define SOTR_20202_1
#include "syscall.h"
#include "traps.h"

#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret

SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(dummy)
#ifdef SOTR_20202_1
SYSCALL(shared)
#endif /*SOTR_20202_1*/
```

2.11 Modificaciones del archivo `vm.c`

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
#ifdef SOTR_20202_1
int
#else
static int
#endif /*SOTR_20202_1*/
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}

// There is one page table per process, plus one that's used when
// a CPU is not running any process (kpgdir). The kernel uses the
// current process's page table during system calls and interrupts;
// page protection bits prevent user code from using the kernel's
// mappings.
//
// setupkvm() and exec() set up every page table like this:
//
//   0..KERNBASE: user memory (text+data+stack+heap), mapped to
//                phys memory allocated by the kernel
//   KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
//   KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
//                for the kernel's instructions and r/o data
//   data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
//                                  rw data + free physical memory
//   0xfe000000..0: mapped direct (devices such as ioapic)
//
// The kernel allocates physical memory for its heap and for user memory
// between V2P(end) and the end of physical memory (PHYSTOP)
// (directly addressable from end..P2V(PHYSTOP)).

//...
  return 0;
}

// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;
```

```
#ifdef SOTR_20202_1
  if(newsz >= SHARED_V)
#else
  if(newsz >= KERNBASE)
#endif /*SOTR_20202_1*/
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}

//...
// Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
  uint i;

  if(pgdir == 0)
    panic("freevm: no pgdir");
```

```
#ifdef SOTR_20202_1
  deallocuvm(pgdir, SHARED_V, 0);
#else
  deallocuvm(pgdir, KERNBASE, 0);
#endif /*SOTR_20202_1*/
  for(i = 0; i < NPDENTRIES; i++){
    if(pgdir[i] & PTE_P){
      char * v = P2V(PTE_ADDR(pgdir[i]));
      kfree(v);
    }
  }
  kfree((char*)pgdir);
}
```

2.12 Un archivo de prueba

```
#include "types.h"
#include "user.h"

int *intA;

int main(int argc,char *argv[])
{
  int PID;

#ifdef SOTR_20202_1
  char *sh;
  sh = (char*)shared();  /*initialize shared memory*/
  if (!sh) {
    printf(2,"shared: fail\n");
    exit();
  }

  intA=(int*)sh;   /*define shared pointer*/
#else
  intA=(int*)malloc(sizeof(int));
#endif /*SOTR_20202_1*/
```

```
  if(argc<2){
    printf(1,"FORMA DE USO:%s <nummber>\n",argv[0]);
    exit();
  }

  *intA=atoi(argv[1]);  /*initialize shared integer*/

  PID=fork();  /*Crear un nuevo proceso (proceso hijo)*/
  if(0==PID){  /*Proceso hijo*/
    printf(1,"PID=%d -- Proceso hijo: %d\n",PID,getpid());
    //dummy1(getpid());
    (*intA)++;
  }else{       /*Proceso padre*/
    wait();
    printf(1,"PID=%d -- Proceso padre: %d\n",PID,getpid());
    //dummy1(getpid());
    (*intA)++;
    printf(1,"*intA=%d\n",*intA);
  }
  exit();
}/*end main()*/
```