

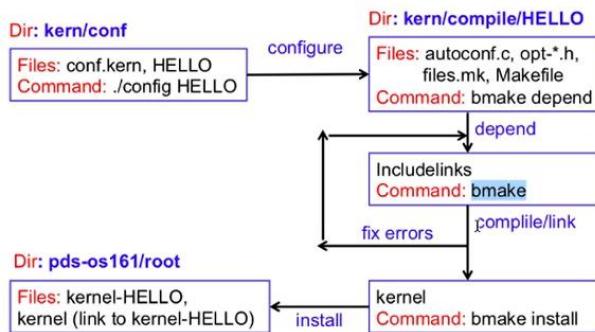
CODICI OS161

1) LAB1 (Intro)

Nella VM abbiamo **USER** e **PASSWORD** uguali (**os161user**). La cartella **\$HOME** è **/usr/os161**. Dentro VSCode devo aprire **os161/root** (mentre i tools sono **os161/tools**).

Il **bootstrap** di os161 si fa dalla cartella **os161/root** con il terminale con il comando **sys161 kernel** (esecuzione normale).

⚠ Diciamo che nella cartella **OS161/os161-base** troviamo le 2 cartelle **/kern** (ovvero il kernel) e **/userland** (programmi/processi utente); mentre in **OS161/root** troviamo l'ambiente per runnare OS161



Apri **VSCode** e apri la cartella OS161. Compilare con **DUMBVM** (non **DUMBVM-OPT** se voglio debug): **Run task -> Run Config -> DUMBVM**, **Run task -> Make depend -> DUMBVM** (equivalente a `bmake depend`), **Run and Build Task -> DUMBVM** (equivalente a `bmake` seguito da `bmake install`); **Run OS161 (debug)** se voglio il debug oppure direttamente **Run -> Start debugging** (perché Run è come Run OS161) e si apre una finestra gdb per scrivere comandi gdb, infatti come in bash se voglio fare un comando devo fare nella finestra gialla sotto **-exec b kmain**. Gli errori sono in mezzo al testo (**fai attenzione!!!**)

⚠ **Build** = costruisco il kernel

Run task -> Run Config -> NOME_CONFIGURAZIONE e vedremo per ogni configurazione runnata dentro **compile** la cartella della configurazione con un **file.h** per ogni opzione scritta nella configurazione. Prima devo creare un opzione con **defoption nome_opzione** dentro **conf.kern**, poi la aggiungo con **option nome_opzione** dentro la configurazione in cui voglio aggiungerla

⚠ **conf.kern** = definisce dispositivi e file per il kernel; contiene anche la definizione a **conf.arch** (che contiene le varie configurazioni a seconda dell'hardware)

Se apro i vari **opt-nome_opzione.h** dentro la cartella della configurazione dentro **compile**, vedo che le opzioni abilitate sono segnate con **1**, mentre quella disabilitate con **0**. Per **disattivare un'opzione** basta mettere un **#** davanti all'opzione nel file di configurazione. Per includere la singola opzione dentro il main.c, devo fare **#include "opt-nome_opzione.h"**.

⚠ Per rendere opzionali alcune istruzioni in base alle opzioni →

```

#ifndef _HELLO_H_
#define _HELLO_H_

#include "opt-hello.h"
#if OPT_HELLO
void hello(void);
#endif

#endif

```

⚠ Distinguere errori di compilazione e errori di link: negli errori di link, le varie compilazioni dei singoli file vanno bene, ma dopo di esse si verifica errore; negli errori di compilazione, il singolo file non viene compilato e da errore.

Nel **conf.kern**, se voglio mettere un nuovo **file.c** (e non un file.h, perché questo viene automaticamente aggiunto col file.c) che sia “assoluto” devo fare **file path_file**; se voglio che sia un file presente solo con una specifica opzione attiva, devo fare **optfile nome_opzione path_file**

⚠ Useremo **VSCode** anche per quanto è **comodo e dettagliato il suo debugger**: quando facciamo Run OS161 (debug) si apre il menù a sinistra di debug dove possiamo aggiungere **breakpoint**, **variabili al watch** e vedere **call stacks** e variabili correnti; ma possiamo comunque usare il terminale di sistema o di VSCode se preferiamo

⚠ Si può usare il File Reference o il “Cerca” di VSCode per trovare tutte le presenze di funzioni/variabili nei vari file di sistema di os161

⚠ Si può anche **DEBUGGARE PIU' THREADS**; inoltre si può **modificare RAM e CPU allocate** (virtualmente) al mio MIPS su cui gira os161 con **gedit sys161**

2) LAB2 (Rendere eseguibile programma utente + MemVirtuale)

I **programmi utente OS161** sono richiamabili mediante il comando **p path_programma**, ma **non sono eseguibili in modo corretto** perché nella versione base di OS161 manca il supporto per:

- syscall **read**, **write** e **_exit** (le **read/write** non sono necessarie per ogni processo user, ma lo diventano per quelli che fanno I/O [di solito i programmi che troviamo per i test fanno **uso di output**, quindi di **write**]) (la **exit** è chiamata sempre in modo implicito/esplicito a fine main)
- gestione della **memoria virtuale** con restituzione della memoria allocata ai processi
- argomenti al main (**argc**, **argv**)

In questo LAIB ci è stato proprio chiesto di **implementare il supporto per queste features mancanti** (eccetto gli argomenti al main per ora) per consentire l'esecuzione di semplici programmi utente

⚠ Abbiamo già detto che i programmi utente sono in **os161/os161-base-2.0.3/userland** (e sottodirettori); questi non possono essere debuggati con **mips-harvard-os161-gdb** come per il kernel, ma va usato **gcc** (**gcc -o file Oggetto file_sorgente.c**) o semplice **gdb**

⚠ Data la nostra configurazione sulla vm di os161, in risposta al comando **p path_programma** avviene la chiamata di **cmd_dispatch()** con l'**attivazione di un nuovo thread/processo** da parte della **menu_execute()**; viene attivato un **nuovo kernel thread** (con **thread_fork()**) nella **common_prog()**. Il **thread padre/principale esce SENZA fare wait sul thread generato**; questo nuovo thread esegue la **cmd_progthread()** (che dentro di sé chiama la **runprogram()**). La **common_prog()** crea anche un **descrittore di processo** (con **proc_create_runprogram()**). Il **thread principale nel frattempo è di nuovo nel menù in attesa di un nuovo input**

Quindi per:

- **write/read** → si consiglia di fare 2 funzioni **sys_write** e **sys_read** con firme uguali alle syscall di **read** e **write**, in un file come **kern/syscall/file_syscalls.c** (qui solo su **stdout** e **stdin**, poi nei prossimi laib su file)
- **_exit** → si consiglia lo stesso approccio usato sopra; la **_exit** (in **userland/lib/libc/libc/exit.c**) ha 1 parametro **int** (codice di errore/successo) e deve:
 - o **mettere questo codice nel campo previsto** (va aggiunto) del descrittore di processo/thread (**struct thread** [in **kern/include/thread.h**] o **struct proc** [in **kern/include/proc.h**])
 - o **far terminare il thread** (e il processo) che chiama la **exit**, chiamando **thread_exit** (in **thread.c**) che pulisce la struct del thread chiamante e lo rende **zombie** (il descrittore è ancora leggibile da altri threads; la distruzione vera verrà fatta da **thread_destroy** chiamata dalla **exorcise** alla prossima **thread_switch**) [ora è solo richiesta una **exit** che attivi la **as_destroy** e che chiuda il thread, ma non che salvi lo stato; anche questa è consigliata come **sys_exit(int status)** da mettere in **kern/syscall/proc_syscalls.c** e che chiami la **as_destroy** e la **thread_exit**]

L'ultimo punto del laboratorio è sulla **GESTIONE DELLA MEMORIA VIRTUALE**: OS161 ha un gestore che fa solo allocazione contigua di memoria reale (ad ogni chiamata di **getppages** o **ram_stealmem**), **senza mai rilasciarla**. Va fatto un nuovo allocatore di memoria contigua che però **mantiene traccia delle pagine (o frame) di RAM occupate e libere**: si consiglia di usare una **bitmap** o un **vettore di flag** → bisogna modificare la **ricerca di RAM**

libera, da parte di `as_prepare_load` e `alloc_kpages` (per inizializzare le strutture dati) e la restituzione di memoria da `is_destroy` e `free_kpages`.

Si consiglia di modificare la `getppages` e/o `ram_stealmem` (per allocare memoria fisica contigua)

⚠ `getppages()` viene chiamata sia `kmalloc()` -> `alloc_kpages()` (per allocazione dinamica al kernel) che da `as_prepare_load()` per allocare memoria ai processi user

SOLUZIONE PROPOSTA: va aggiunta l'opzione `syscalls` in `conf.kern`, che rende opzionali `proc_syscalls.c`

e `file_syscalls.c`, ovvero:

```
defoption syscalls  
optfile syscalls syscall/file_syscalls.c  
optfile syscalls syscall/proc_syscalls.c
```

[mettere i file `proc_syscalls.c` e `file_syscalls.c` in `kern/syscall`, mentre `syscall.c` va messo in `kern/arch/mips/syscall`; infine mettere `syscall.h` in `kern/include` e non in `kern/include/kern` (qui troviamo la versione con i codici delle syscalls); `dumbvm.c` va in `kern/arch/mips/vm`]

Vediamo i file proposti (`dumbvm.c` è al fondo perché si riferisce alla 2^ parte del LAB, quella del deallocator):

syscall.c:

```
#include ...  
  
/*  
 * System call dispatcher.  
 *  
 * A pointer to the trapframe created during exception entry (in  
 * exception-*.S) is passed in.  
 *  
 * The calling conventions for syscalls are as follows: Like ordinary  
 * function calls, the first 4 32-bit arguments are passed in the 4  
 * argument registers a0-a3. 64-bit arguments are passed in *aligned*  
 * pairs of registers, that is, either a0/a1 or a2/a3. This means that  
 * if the first argument is 32-bit and the second is 64-bit, a1 is  
 * unused.  
 *  
 * This much is the same as the calling conventions for ordinary  
 * function calls. In addition, the system call number is passed in  
 * the v0 register.  
 *  
 * On successful return, the return value is passed back in the v0  
 * register, or v0 and v1 if 64-bit. This is also like an ordinary  
 * function call, and additionally the a3 register is also set to 0 to  
 * indicate success.  
 *  
 * On an error return, the error code is passed back in the v0  
 * register, and the a3 register is set to 1 to indicate failure.  
 * (Userlevel code takes care of storing the error code in errno and  
 * returning the value -1 from the actual userlevel syscall function.  
 * See src/user/lib/libc/arch/mips/syscalls-mips.S and related files.)  
 *  
 * Upon syscall return the program counter stored in the trapframe  
 * must be incremented by one instruction; otherwise the exception  
 * return code will restart the "syscall" instruction and the system  
 * call will repeat forever.  
 *  
 * If you run out of registers (which happens quickly with 64-bit  
 * values) further arguments must be fetched from the user-level  
 * stack, starting at sp+16 to skip over the slots for the  
 * registerized values, with copyin().  
 */  
  
void syscall(struct trapframe *tf) {  
    int callno;  
    int32_t retval;  
    int err;
```

```

KASSERT(curthread != NULL);
KASSERT(curthread->t_curspl == 0);
KASSERT(curthread->t_iplhigh_count == 0);

callno = tf->tf_v0;

/*
 * Initialize retval to 0. Many of the system calls don't
 * really return a value, just 0 for success and -1 on
 * error. Since retval is the value returned on success,
 * initialize it to 0 by default; thus it's not necessary to
 * deal with it except for calls that return other values,
 * like write.
 */
retval = 0;

switch (callno) {
    case SYS_reboot:
        err = sys_reboot(tf->tf_a0);
        break;

    case SYS_time:
        err = sys_time((userptr_t)tf->tf_a0, (userptr_t)tf->tf_a1);
        break;

#if OPT_SYSCALLS
    case SYS_write:
        retval = sys_write((int)tf->tf_a0, (userptr_t)tf->tf_a1, (size_t)tf->tf_a2);
        if (retval<0) err = ENOSYS; /* error: function not implemented */
        else err = 0;
        break;

    case SYS_read:
        retval = sys_read((int)tf->tf_a0, (userptr_t)tf->tf_a1, (size_t)tf->tf_a2);
        if (retval<0) err = ENOSYS; /* error: function not implemented */
        else err = 0;
        break;

    case SYS_exit:
        sys_exit((int)tf->tf_a0);
        break;
#endif

    default:
        kprintf("Unknown syscall %d\n", callno);
        err = ENOSYS;
        break;
}

if (err) {
    /*
     * Return the error code. This gets converted at
     * userlevel to a return value of -1 and the error
     * code in errno.
     */
    tf->tf_v0 = err;
    tf->tf_a3 = 1;      /* signal an error */
}
else {
    /* Success. */
    tf->tf_v0 = retval;
    tf->tf_a3 = 0;      /* signal no error */
}

```

```

/*
 * Now, advance the program counter, to avoid restarting
 * the syscall over and over again.
 */

tf->tf_epc += 4;

/* Make sure the syscall code didn't forget to lower spl */
KASSERT(curthread->t_curspl == 0);
/* ...or leak any spinlocks */
KASSERT(curthread->t_iplhigh_count == 0);
}

/*
* Enter user mode for a newly forked process.
* This function is provided as a reminder. You need to write
* both it and the code that calls it.
* Thus, you can trash it and do things another way if you prefer.
*/
void enter_forked_process(struct trapframe *tf) {
    (void)tf;
}

```

file_syscalls.c:

```

#include ...

int sys_write(int fd, userptr_t buf_ptr, size_t size) {
    int i;
    char *p = (char *)buf_ptr;

    if (fd!=STDOUT_FILENO && fd!=STDERR_FILENO) {
        kprintf("sys_write supported only to stdout\n");
        return -1;
    }

    for (i=0; i<(int)size; i++) {
        putch(p[i]);
    }

    return (int)size;
}

```

```

int sys_read(int fd, userptr_t buf_ptr, size_t size) {
    int i;
    char *p = (char *)buf_ptr;

    if (fd!=STDIN_FILENO) {
        kprintf("sys_read supported only to stdin\n");
        return -1;
    }

    for (i=0; i<(int)size; i++) {
        p[i] = getch();
        if (p[i] < 0)
            return i;
    }

    return (int)size;
}

```

proc_syscalls.c:

```

#include ...
void sys_exit(int status) {

```

```

/* get address space of current process and destroy */
struct addrspace *as = proc_getas();
as_destroy(as);
/* thread exits. proc data structure will be lost */
thread_exit();

panic("thread_exit returned (should not happen)\n");
(void) status; // TODO: status handling
}

syscall.h:
#ifndef _SYSCALL_H_
#define _SYSCALL_H_

#include <cdefs.h> /* for __DEAD */
#include "opt-syscalls.h"

struct trapframe; /* from <machine/trapframe.h> */
void syscall(struct trapframe *tf);
void enter_forked_process(struct trapframe *tf); /* Helper for fork() */
__DEAD void enter_new_process(int argc, userptr_t argv, userptr_t env,
    vaddr_t stackptr, vaddr_t entrypoint); /* Enter user mode. Does not return. */

int sys_reboot(int code);
int sys_time(userptr_t user_seconds, userptr_t user_nanoseconds);
#if OPT_SYSCALS
int sys_write(int fd, userptr_t buf_ptr, size_t size);
int sys_read(int fd, userptr_t buf_ptr, size_t size);
void sys_exit(int status);
#endif

#endif /* _SYSCALL_H_ */

```

dumbvm.c [QUESTO è IL DEALLOCATORE CHE ABBIAMO CITATO ANCHE IN TEORIA DI OS161]

```

#include ...
/* under dumbvm, always have 72k of user stack */
/* (this must be > 64K so argument blocks of size ARG_MAX will fit) */
#define DUMBVM_STACKPAGES 18

/* G.Cabodi: set DUMBVM_WITH_FREE
 * - 0: original dumbvm
 * - 1: support for alloc/free
 */
#define DUMBVM_WITH_FREE 1

/* Wrap ram_stealmem in a spinlock */
static struct spinlock stealmem_lock = SPINLOCK_INITIALIZER;

#if DUMBVM_WITH_FREE
/* G.Cabodi - support for free/alloc */

static struct spinlock freemem_lock = SPINLOCK_INITIALIZER;
static unsigned char *freeRamFrames = NULL;
static unsigned long *allocSize = NULL;
static int nRamFrames = 0;
static int allocTableActive = 0;

static int isTableActive () {
    int active;
    spinlock_acquire(&freemem_lock);
    active = allocTableActive;
}
```

```

    spinlock_release(&freemem_lock);
    return active;
}

void vm_bootstrap(void) {
    int i;
    nRamFrames = ((int)ram_getsize()) / PAGE_SIZE;
    /* alloc freeRamFrame and allocSize */
    freeRamFrames = kmalloc(sizeof(unsigned char) * nRamFrames);
    if (freeRamFrames == NULL) return;
    allocSize = kmalloc(sizeof(unsigned long) * nRamFrames);
    if (allocSize == NULL) {
        /* reset to disable this vm management */
        freeRamFrames = NULL; return;
    }
    for (i = 0; i < nRamFrames; i++) {
        freeRamFrames[i] = (unsigned char)0;
        allocSize[i] = 0;
    }
    spinlock_acquire(&freemem_lock);
    allocTableActive = 1;
    spinlock_release(&freemem_lock);
}

/*
 * Check if we're in a context that can sleep. While most of the
 * operations in dumbvm don't in fact sleep, in a real VM system many
 * of them would
 */
static void dumbvm_can_sleep(void) {
    if (CURCPU_EXISTS()) {
        /* must not hold spinlocks */
        KASSERT(curcpu->c_spinlocks == 0);
        /* must not be in an interrupt handler */
        KASSERT(curthread->t_in_interrupt == 0);
    }
}

static paddr_t getfreepages(unsigned long npages) {
    paddr_t addr;
    long i, first, found, np = (long)npages;
    if (!isTableActive()) return 0;
    spinlock_acquire(&freemem_lock);
    for (i = 0, first = found = -1; i < nRamFrames; i++) {
        if (freeRamFrames[i]) {
            if (i == 0 || !freeRamFrames[i - 1]) first = i; /* set 1st free in a interval */
            if (i - first + 1 >= np) {
                found = first;
                break;
            }
        }
    }
    if (found >= 0) {
        for (i = found; i < found + np; i++) {
            freeRamFrames[i] = (unsigned char)0;
        }
        allocSize[found] = np;
        addr = (paddr_t) found * PAGE_SIZE;
    } else {
        addr = 0;
    }
    spinlock_release(&freemem_lock);
    return addr;
}

```

```

static paddr_t getppages(unsigned long npages) {
    paddr_t addr;

    /* try freed pages first */
    addr = getfreepages(npages);
    if (addr == 0) {
        /* call stealmem */
        spinlock_acquire(&stealmem_lock);
        addr = ram_stealmem(npages);
        spinlock_release(&stealmem_lock);
    }
    if (addr!=0 && isTableActive()) {
        spinlock_acquire(&freemem_lock);
        allocSize[addr/PAGE_SIZE] = npages;
        spinlock_release(&freemem_lock);
    }
    return addr;
}

static int freepages(paddr_t addr, unsigned long npages){
    long i, first, np=(long)npages;

    if (!isTableActive()) return 0;
    first = addr/PAGE_SIZE;
    KASSERT(allocSize!=NULL);
    KASSERT(nRamFrames>first);

    spinlock_acquire(&freemem_lock);
    for (i=first; i<first+np; i++) {
        freeRamFrames[i] = (unsigned char)1;
    }
    spinlock_release(&freemem_lock);
    return 1;
}

/* Allocate/free some kernel-space virtual pages */
vaddr_t alloc_kpages(unsigned npages) {
    paddr_t pa;

    dumbvm_can_sleep();
    pa = getppages(npages);
    if (pa==0) {
        return 0;
    }
    return PADDR_TO_KVADDR(pa);
}

void free_kpages(vaddr_t addr){
    if (isTableActive()) {
        paddr_t paddr = addr - MIPS_KSEG0;
        long first = paddr/PAGE_SIZE;
        KASSERT(allocSize!=NULL);
        KASSERT(nRamFrames>first);
        freepages(paddr, allocSize[first]);
    }
}

void vm_tlbshootdown(const struct tlbshootdown *ts) {
    (void)ts;
    panic("dumbvm tried to do tlb shootdown?! \n");
}

int vm_fault(int faulttype, vaddr_t faultaddress){
    vaddr_t vbase1, vtop1, vbase2, vtop2, stackbase, stacktop;
    paddr_t paddr;

```

```

int i;
uint32_t ehi, elo;
struct addrspace *as;
int spl;

faultaddress &= PAGE_FRAME;

DEBUG(DB_VM, "dumbvm: fault: 0x%llx", faultaddress);

switch (faulttype) {
    case VM_FAULT_READONLY:
        /* We always create pages read-write, so we can't get this */
        panic("dumbvm: got VM_FAULT_READONLY\n");
    case VM_FAULT_READ:
    case VM_FAULT_WRITE:
        break;
    default:
        return EINVAL;
}

if (curproc == NULL) {
    /*
     * No process. This is probably a kernel fault early
     * in boot. Return EFAULT so as to panic instead of
     * getting into an infinite faulting loop.
     */
    return EFAULT;
}

as = proc_getas();
if (as == NULL) {
    /*
     * No address space set up. This is probably also a
     * kernel fault early in boot.
     */
    return EFAULT;
}

/* Assert that the address space has been set up properly. */
KASSERT(as->as_vbase1 != 0);
KASSERT(as->as_pbase1 != 0);
KASSERT(as->as_npages1 != 0);
KASSERT(as->as_vbase2 != 0);
KASSERT(as->as_pbase2 != 0);
KASSERT(as->as_npages2 != 0);
KASSERT(as->as_stackpbase != 0);
KASSERT((as->as_vbase1 & PAGE_FRAME) == as->as_vbase1);
KASSERT((as->as_pbase1 & PAGE_FRAME) == as->as_pbase1);
KASSERT((as->as_vbase2 & PAGE_FRAME) == as->as_vbase2);
KASSERT((as->as_pbase2 & PAGE_FRAME) == as->as_pbase2);
KASSERT((as->as_stackpbase & PAGE_FRAME) == as->as_stackpbase);

vbase1 = as->as_vbase1;
vtop1 = vbase1 + as->as_npages1 * PAGE_SIZE;
vbase2 = as->as_vbase2;
vtop2 = vbase2 + as->as_npages2 * PAGE_SIZE;
stackbase = USERSTACK - DUMBVM_STACKPAGES * PAGE_SIZE;
stacktop = USERSTACK;

if (faultaddress >= vbase1 && faultaddress < vtop1) {
    paddr = (faultaddress - vbase1) + as->as_pbase1;
}
else if (faultaddress >= vbase2 && faultaddress < vtop2) {
    paddr = (faultaddress - vbase2) + as->as_pbase2;
}

```

```

else if (faultaddress >= stackbase && faultaddress < stacktop) {
    paddr = (faultaddress - stackbase) + as->as_stackpbase;
}
else {
    return EFAULT;
}
/* make sure it's page-aligned */
KASSERT((paddr & PAGE_FRAME) == paddr);
/* Disable interrupts on this CPU while frobbing the TLB. */
spl = splhigh();
for (i=0; i<NUM_TLB; i++) {
    tlb_read(&ehi, &elo, i);
    if (elo & TLBLO_VALID) {
        continue;
    }
    ehi = faultaddress;
    elo = paddr | TLBLO_DIRTY | TLBLO_VALID;
    DEBUG(DB_VM, "dumbvm: 0x%x -> 0x%x\n", faultaddress, paddr);
    tlb_write(ehi, elo, i);
    splx(spl);
    return 0;
}
kprintf("dumbvm: Ran out of TLB entries - cannot handle page fault\n");
splx(spl);
return EFAULT;
}

struct addrspace* as_create(void) {
    struct addrspace *as = kmalloc(sizeof(struct addrspace));
    if (as==NULL) {
        return NULL;
    }
    as->as_vbase1 = 0;
    as->as_pbase1 = 0;
    as->as_npages1 = 0;
    as->as_vbase2 = 0;
    as->as_pbase2 = 0;
    as->as_npages2 = 0;
    as->as_stackpbase = 0;

    return as;
}

void as_destroy(struct addrspace *as){
    dumbvm_can_sleep();
    freeppages(as->as_pbase1, as->as_npages1);
    freeppages(as->as_pbase2, as->as_npages2);
    freeppages(as->as_stackpbase, DUMBVM_STACKPAGES);
    kfree(as);
}

void as_activate(void) {
    int i, spl;
    struct addrspace *as;
    as = proc_getas();
    if (as == NULL) {
        return;
    }
    /* Disable interrupts on this CPU while frobbing the TLB. */
    spl = splhigh();
    for (i=0; i<NUM_TLB; i++) {
        tlb_write(TLBHI_INVALID(i), TLBLO_INVALID(), i);
    }
    splx(spl);
}

```

```

void as_deactivate(void) {
    /* nothing */
}

int as_define_region(struct addrspace *as, vaddr_t vaddr, size_t sz, int readable, int
writeable, int executable) {
    size_t npages;
    dumbvm_can_sleep();

    /* Align the region. First, the base... */
    sz += vaddr & ~(vaddr_t)PAGE_FRAME;
    vaddr &= PAGE_FRAME;
    /* ...and now the length. */
    sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;
    npages = sz / PAGE_SIZE;

    /* We don't use these - all pages are read-write */
    (void)readable;
    (void)writeable;
    (void)executable;

    if (as->as_vbase1 == 0) {
        as->as_vbase1 = vaddr;
        as->as_npagem1 = npages;
        return 0;
    }
    if (as->as_vbase2 == 0) {
        as->as_vbase2 = vaddr;
        as->as_npagem2 = npages;
        return 0;
    }

    /* Support for more than two regions is not available */
    kprintf("dumbvm: Warning: too many regions\n");
    return ENOSYS;
}

static void as_zero_region(paddr_t paddr, unsigned npages) {
    bzero((void *)PADDR_TO_KVADDR(paddr), npages * PAGE_SIZE);
}

int as_prepare_load(struct addrspace *as) {
    KASSERT(as->as_pbase1 == 0);
    KASSERT(as->as_pbase2 == 0);
    KASSERT(as->as_stackpbase == 0);
    dumbvm_can_sleep();

    as->as_pbase1 = getppages(as->as_npagem1);
    if (as->as_pbase1 == 0) {
        return ENOMEM;
    }
    as->as_pbase2 = getppages(as->as_npagem2);
    if (as->as_pbase2 == 0) {
        return ENOMEM;
    }
    as->as_stackpbase = getppages(DUMBVM_STACKPAGES);
    if (as->as_stackpbase == 0) {
        return ENOMEM;
    }

    as_zero_region(as->as_pbase1, as->as_npagem1);
    as_zero_region(as->as_pbase2, as->as_npagem2);
    as_zero_region(as->as_stackpbase, DUMBVM_STACKPAGES);
    return 0;
}

```

```

int as_complete_load(struct addrspace *as) {
    dumbvm_can_sleep();
    (void)as;
    return 0;
}

int as_define_stack(struct addrspace *as, vaddr_t *stackptr) {
    KASSERT(as->as_stackpbase != 0);
    *stackptr = USERSTACK;
    return 0;
}

int as_copy(struct addrspace *old, struct addrspace **ret){
    struct addrspace *new;
    dumbvm_can_sleep();

    new = as_create();
    if (new==NULL) {
        return ENOMEM;
    }
    new->as_vbase1 = old->as_vbase1;
    new->as_npages1 = old->as_npages1;
    new->as_vbase2 = old->as_vbase2;
    new->as_npages2 = old->as_npages2;

    /* (Mis)use as_prepare_load to allocate some physical memory. */
    if (as_prepare_load(new)) {
        as_destroy(new);
        return ENOMEM;
    }
    KASSERT(new->as_pbase1 != 0);
    KASSERT(new->as_pbase2 != 0);
    KASSERT(new->as_stackpbase != 0);

    memmove((void *)PADDR_TO_KVADDR(new->as_pbase1), (const void *)PADDR_TO_KVADDR(old->as_pbase1), old->as_npages1*PAGE_SIZE);
    memmove((void *)PADDR_TO_KVADDR(new->as_pbase2), (const void *)PADDR_TO_KVADDR(old->as_pbase2), old->as_npages2*PAGE_SIZE);
    memmove((void *)PADDR_TO_KVADDR(new->as_stackpbase), (const void *)PADDR_TO_KVADDR(old->as_stackpbase), DUMBVM_STACKPAGES*PAGE_SIZE);

    *ret = new;
    return 0;
}

#else
/* G.Cabodi - original dumbvm */
...

```

3) LAB3 (implementazione di lock e CV [solo kernel-side])

Si può dividere in 2 parti:

- TEST DI SINCRONIZZAZIONE e IMPLEMENTAZIONE DEI LOCK:

Debuggare i test **sy1** (sincronizzazione mediante **semafori** [già realizzati in OS161]) e **sy2** (sincronizzazione mediante **lock** [non realizzati ancora])

⚠ I **semafori** sono realizzati “*non busy-waiting*”; per realizzarli si ricorre a **wait_channel** (una CV OS161) associato a **spinlock** (un lock “*busy-waiting*”, quindi adatti a casi nel kernel in cui c’è attesa limitata).

Si chiede quindi di **realizzare i lock in 2 modi diversi** (usando le **opzioni** in conf.kern):

1. con **semafori** → un lock è un *semaforo binario* “praticamente” (counter con valore max = 1); quindi modificare la **struct lock** in kern/include/synch.h + fare **lock_create, lock_destroy, lock_acquire, lock_release, lock_do_i_hold** in kern/thread/synch.c
2. con **wchan** e **spinlock** (versione migliore) [praticamente copia la realizzazione dei semafori]

⚠ Il **lock** non è proprio un semaforo binario in quanto ha il concetto di ownership: **lock_release** può essere fatta solo dal thread che ha ottenuto il lock con **lock_acquire** e che ne è **owner** (quindi la struct lock deve avere un puntatore al thread owner; è disponibile per aiuto **curthread**) [se non owner → KASSERT]

⚠ La **lock_do_i_hold** indica al programma chiamante se il thread corrente sia l’owner di un lock ricevuto come parametro (in teoria basterà leggere in mutua esclusione il puntatore nella struct lock → occhio se si usa spinlock per la mutua esclusione, in quanto OS161 non consente l’attesa su spinlock [**spinlock_acquire**] se il thread corrente è già owner di un altro spinlock)

Soluzione (proposta) [con anche una sezione sui semafori per capire come sono fatti]:

```
***** SEMAFORI *****

struct semaphore *sem_create(const char *name, unsigned initial_count) {
    struct semaphore *sem = kmalloc(sizeof(*sem));
    if (sem == NULL) return NULL;

    sem->sem_name = kstrdup(name);
    if (sem->sem_name == NULL) { kfree(sem); return NULL; }

    sem->sem_wchan = wchan_create(sem->sem_name);
    if (sem->sem_wchan == NULL) { kfree(sem->sem_name); kfree(sem); return NULL; }

    spinlock_init(&sem->sem_lock);
    sem->sem_count = initial_count;
    return sem;
}

void sem_destroy(struct semaphore *sem) {
    KASSERT(sem != NULL);

    spinlock_cleanup(&sem->sem_lock); /* wchan_cleanup will assert if anyone's waiting on it */

    wchan_destroy(sem->sem_wchan);
    kfree(sem->sem_name); kfree(sem);
}

void P(struct semaphore *sem) {
    KASSERT(sem != NULL);
    // May not block in an interrupt handler
    KASSERT(curthread->t_in_interrupt == false);

    spinlock_acquire(&sem->sem_lock);
}
```

```

        while (sem->sem_count == 0) {
            wchan_sleep(sem->sem_wchan, &sem->sem_lock);
        }
        KASSERT(sem->sem_count > 0);
        sem->sem_count--;
    }

    spinlock_release(&sem->sem_lock);
}

void V(struct semaphore *sem) {
    KASSERT(sem != NULL);

    spinlock_acquire(&sem->sem_lock);

    sem->sem_count++;
    KASSERT(sem->sem_count > 0);
    wchan_wakeone(sem->sem_wchan, &sem->sem_lock);

    spinlock_release(&sem->sem_lock);
}

/***** LOCK *****/
struct lock* lock_create(const char *name) {

    struct lock* lock = kmalloc(sizeof(*lock));
    if (lock == NULL) return NULL;

    lock->lk_name = kstrdup(name);
    if (lock->lk_name == NULL) { kfree(lock); return NULL; }

#ifndef OPT_SYNCH
// se uso semaforo (richiesta 1)
#ifndef USE_SEMAPHORE_FOR_LOCK
    lock->lk_sem = sem_create(lock->lk_name,1);
    if (lock->lk_sem == NULL) {
// se uso wchan (richiesta 2)
#else
    lock->lk_wchan = wchan_create(lock->lk_name);
    if (lock->lk_wchan == NULL) {
#endif
#endif
    kfree(lock->lk_name);
    kfree(lock);
    return NULL;
}
    lock->lk_owner = NULL;
    spinlock_init(&lock->lk_lock);
#endif
    return lock;
}

void lock_destroy(struct lock *lock) {
    KASSERT(lock != NULL);

#ifndef OPT_SYNCH
    spinlock_cleanup(&lock->lk_lock);
#ifndef USE_SEMAPHORE_FOR_LOCK
    sem_destroy(lock->lk_sem);
#else
    wchan_destroy(lock->lk_wchan);
#endif
#endif
    kfree(lock->lk_name);
    kfree(lock);
}

```

```

}

void lock_acquire(struct lock *lock) {
#if OPT_SYNCH
    KASSERT(lock != NULL);
    if (lock_do_i_hold(lock)) {
        kprintf("AAACKK!\n");
    }
    KASSERT(!(lock_do_i_hold(lock)));
    KASSERT(curthread->t_in_interrupt == false);

#endif USE_SEMAPHORE_FOR_LOCK
    P(lock->lk_sem);
    spinlock_acquire(&lock->lk_lock);
#else
    spinlock_acquire(&lock->lk_lock);
    while (lock->lk_owner != NULL) {
        wchan_sleep(lock->lk_wchan, &lock->lk_lock);
    }
#endif
    KASSERT(lock->lk_owner == NULL);
    lock->lk_owner=curthread;
    spinlock_release(&lock->lk_lock);
#endif
    (void)lock;
}

void lock_release(struct lock *lock) {
#if OPT_SYNCH
    KASSERT(lock != NULL);
    KASSERT(lock_do_i_hold(lock));

    spinlock_acquire(&lock->lk_lock);
    lock->lk_owner=NULL;

#endif USE_SEMAPHORE_FOR_LOCK
    V(lock->lk_sem);
#else
    wchan_wakeone(lock->lk_wchan, &lock->lk_lock);
#endif
    spinlock_release(&lock->lk_lock);
#endif
    (void)lock;
}

bool lock_do_i_hold(struct lock *lock) {
#if OPT_SYNCH
    bool res;
    spinlock_acquire(&lock->lk_lock);
    res = lock->lk_owner == curthread;
    spinlock_release(&lock->lk_lock);
    return res;

#endif
    (void)lock;
    return true;
}

```

- TEST DI SINCRONIZZAZIONE e IMPLEMENTAZIONE DELLE CONDITION VARIABLE:

Vanno fatte anche le **CV** (in OS161 vuote). Debuggare i test **sy3** e **sy4**.

Una **CONDITION VARIABLE** (CV) è una primitiva di sincronizzazione che consente di **attendere che una condizione diventi vera**; la CV è sempre **associata ad un lock** (⚠ diverso da **wait_channel** che ha uno

spinlock associato) che viene passato come parametro alle funzioni per garantire accesso protetto in mutua esclusione.

Le funzioni sono **cv_wait** (rilascia il lock ricevuto come parametro [che deve essere stato acquisito prima dal thread chiamante] e si mette in attesa), **cv_signal** e **cv_broadcast** (svegliano i thread in attesa [la signal solo 1, la broadcast tutti]).

Una CV si potrebbe fare con i semafori (**cv_wait** con **P()**; **cv_signal** con **V()**); inoltre si dovrebbe gestire il fatto che una **V()** viene “ricordata” se non ho thread in attesa ora, e potrebbe sbloccare una futura **P()**, mentre **con le CV si sbloccano solo thread in attesa ora**).

Perciò è meglio realizzare le CV con wait_channel (e spinlock) [molto simili, ma CV lock, mentre WC spinlock]

⚠ Sia **rilascio** del lock sia **messaggio in attesa** del thread (nella **cv_wait**) sono da fare **ATOMICHE** (evitando cioè che un altro thread acquisisca il lock prima che il thread vada in attesa (sul **wait_channel**)) [sfruttare lo spinlock del WC a tale scopo → mutua esclusione]

Soluzione (proposta):

```
***** CV *****  
  
struct cv* cv_create(const char *name) {  
    struct cv* cv = kmalloc(sizeof(*cv));  
    if (cv == NULL) return NULL;  
  
    cv->cv_name = kstrdup(name);  
    if (cv->cv_name==NULL) { kfree(cv); return NULL; }  
  
#if OPT_SYNCH  
    cv->cv_wchan = wchan_create(cv->cv_name);  
    if (cv->cv_wchan == NULL) { kfree(cv->cv_name); kfree(cv); return NULL; }  
  
    spinlock_init(&cv->cv_lock);  
#endif  
    return cv;  
}  
  
void cv_destroy(struct cv *cv) {  
    KASSERT(cv != NULL);  
  
#if OPT_SYNCH  
    spinlock_cleanup(&cv->cv_lock);  
    wchan_destroy(cv->cv_wchan);  
#endif  
    kfree(cv->cv_name);  
    kfree(cv);  
}  
  
void cv_wait(struct cv *cv, struct lock *lock) {  
#if OPT_SYNCH  
    KASSERT(lock != NULL);  
    KASSERT(cv != NULL);  
    KASSERT(lock_do_i_hold(lock));  
  
    spinlock_acquire(&cv->cv_lock);  
    lock_release(lock);  
  
    wchan_sleep(cv->cv_wchan,&cv->cv_lock);  
  
    spinlock_release(&cv->cv_lock);  
    lock_acquire(lock);  
  
#endif  
    (void)cv;  
    (void)lock;  
}
```

```
void cv_signal(struct cv *cv, struct lock *lock) {
#ifndef OPT_SYNCH
    KASSERT(lock != NULL);
    KASSERT(cv != NULL);
    KASSERT(lock_do_i_hold(lock));

    spinlock_acquire(&cv->cv_lock);

    wchan_wakeone(cv->cv_wchan,&cv->cv_lock);

    spinlock_release(&cv->cv_lock);
#endif
    (void)cv;
    (void)lock;
}

void cv_broadcast(struct cv *cv, struct lock *lock) {
#ifndef OPT_SYNCH
    KASSERT(lock != NULL);
    KASSERT(cv != NULL);
    KASSERT(lock_do_i_hold(lock));

    spinlock_acquire(&cv->cv_lock);

    wchan_wakeall(cv->cv_wchan,&cv->cv_lock);

    spinlock_release(&cv->cv_lock);
#endif
    (void)cv;
    (void)lock;
}
```

4) LAB4 (waitpid - attesa fine processo)

Si vuole realizzare la syscall **waitpid** (permette a processo di attendere il cambio di stato di un altro processo identificato da **pid**) [per semplicità si chiede solo il cambio di stato a “**terminato**”]; dopo un **thread_exit** (dell’ultimo thread di processo), il processo resta in stato “**zombie**” fino a che un altro processo non fa **wait**/**waitpid** [in OS161 solo **waitpid**]. Il laboratorio è divisibile in 3 parti:

- **ATTESA DI TERMINAZIONE di processo user con ritorno del suo stato di uscita:**

Realizzare **int proc_wait(struct proc *p)** che gestisca (senza pid) l’attesa della fine (con chiamata a **syscall_exit()**) di un altro processo [di cui si ha **struct *proc**], mediante semaforo o CV (aggiunti come campo della struct proc)

⚠ Dato che usa la **struct proc**, si può usare solo in kernel

Questa funzione potrebbe essere fatta in kern/proc/proc.c e chiamata in **common_prog** (dopo che questa ha chiamato **thread_fork** con successo) per aspettare la fine del processo attivato, ovvero la **common_prog** può aspettare la fine del processo figlio avviato con **exit_code = proc_wait(proc)**; poi stampa su console il codice di ritorno della _exit [salvato nella struct proc] prima di tornare al chiamante

Soluzione (proposta):

```
int proc_wait(struct proc *proc){  
#if OPT_WAITPID  
    int return_status;  
    /* NULL and kernel proc forbidden */  
    KASSERT(proc != NULL);  
    KASSERT(proc != kproc);  
  
    /* wait on semaphore or condition variable */  
#if USE_SEMAPHORE_FOR_WAITPID  
    P(proc->p_sem);  
#else  
    lock_acquire(proc->p_lock);  
    cv_wait(proc->p_cv);  
    lock_release(proc->p_lock);  
#endif  
    return_status = proc->p_status;  
    proc_destroy(proc);  
    return return_status;  
#else  
    /* this doesn't synchronize */  
    (void)proc;  
    return 0;  
#endif  
}  
}
```

- **DISTRUZIONE della struct proc:**

struct proc di un processo non può essere distrutta (durante la **_exit**) finchè un altro processo che chiama **wait**/**waitpid** non ne riceva la segnalazione (con lo stato di uscita).

Si consiglia di chiamare la **proc_destroy** (che distrugge questa struct) nella **proc_wait** dopo l’attesa su semaforo o CV [ovvero la struct non viene distrutta quando termina il processo, ma nella **proc_wait** fatta da un altro processo (eventualmente il kernel)].

Questo però modifica anche la **sys_exit** (in quanto ora deve termina il thread, ma non distrugge la **struct proc**, bensì ne segnala solo la terminazione; infatti come detto prima, la distruzione completa avviene nella **proc_wait** che chiama la **proc_destroy** dopo aver settato lo **status**)

⚠ **proc_destroy()** richiede che la **struct proc** che si sta distruggendo non abbia thread attivi (**KASSERT(proc->p_numthreads == 0)**); ma dato che **sys_exit()** segnala la fine del processo prima di chiamare

`thread_exit()`, è possibile che il kernel in attesa nella `common_prog()` venga svegliato e chiami `proc_destroy` prima che `thread_exit` "stacchi" il thread dal processo (chiamata a `sproc_remthread()`)

Soluzione per evitare questa race condition: chiamare `proc_remthread` in modo esplicito nella `sys_exit` "prima" di segnalare la fine del processo, modificando `thread_exit` in modo che accetti un thread già staccato dal processo (⚠ non OBBLIGARE `thread_exit` a vedere SEMPRE un thread "staccato" in quanto viene usata anche in altri contesti)

Soluzione (proposta):

```
void proc_destroy(struct proc *proc) {
    KASSERT(proc != NULL);
    KASSERT(proc != kproc);

    /* VFS fields */
    if (proc->p_cwd) {
        VOP_DECREF(proc->p_cwd);
        proc->p_cwd = NULL;
    }

    /* VM fields */
    if (proc->p_addrspace) {
        struct addrspace *as;

        if (proc == curproc) {
            as = proc_setas(NULL);
            as_deactivate();
        }
        else {
            as = proc->p_addrspace;
            proc->p_addrspace = NULL;
        }
        as_destroy(as);
    }

    KASSERT(proc->p_numthreads == 0);
    spinlock_cleanup(&proc->p_lock);

    proc_end_waitpid(proc);

    kfree(proc->p_name);
    kfree(proc);
}

void sys_exit(int status) {
#if OPT_WAITPID
    struct proc *p = curproc;
    p->p_status = status & 0xff; /* just lower 8 bits returned */
    proc_remthread(curthread);

#if USE_SEMAPHORE_FOR_WAITPID
    V(p->p_sem);
#else
    lock_acquire(p->p_lock);
    cv_signal(p->p_cv);
    lock_release(p->p_lock);
#endif
#endif

#else
    /* get address space of current process and destroy */
    struct addrspace *as = proc_getas();
    as_destroy(as);
#endif
    thread_exit();

    panic("thread_exit returned (should not happen)\n");
    (void) status; // TODO: status handling
}
```

- ASSEGNAZIONE DI PID a processo:

`proc_wait` non realizza completamente ciò richiesto dalla `waitpid` perché parte da `*proc` e non dal `pid` (di tipo `pid_t` [un intero compreso tra `PID_MIN` e `PID_MAX`, definiti in `kern/include/limits.h` in base a `__PID_MIN` e `__PID_MAX` in `kern/include/kern/limits.h`]); per la *relazione proc-pid* serve una **TABELLA** (per semplicità un *vettore di *proc* in cui indice = `pid` oppure un vettore di coppie (`pid, *proc`)). Se invece mettessimo un campo `pid` nella struct `proc` potremmo prenderlo da `*proc` come suo campo.

Ogni processo creato va aggiunto in tabella, ogni processo distrutto va rimosso dalla tabella (dopo `waitpid`). La tabella può essere realizzata come **variabile globale** in `kern/proc/proc.c`; si deve poi realizzare la funzione `sys_waitpid` da chiamare in `syscall()` [usare lo stesso file usato per la `sys_exit`, ovvero `proc_syscalls.c`]

Soluzione (proposta):

```
#define MAX_PROC 100
static struct _processTable {
    int active;           /* initial value 0 */
    struct proc *proc[MAX_PROC+1]; /* [0] not used. pids are >= 1 */
    int last_i;           /* index of last allocated pid */
    struct spinlock lk;   /* Lock for this table */
} processTable;

int sys_waitpid(pid_t pid, userptr_t statusp, int options) {
#ifndef OPT_WAITPID
    struct proc *p = proc_search_pid(pid);
    int s;
    (void)options; /* not handled */

    if (p==NULL) return -1;
    s = proc_wait(p);
    if (statusp!=NULL) *(int*)statusp = s;
    return pid;
#else
    (void)options; /* not handled */
    (void)pid;
    (void)statusp;
    return -1;
#endif
}
```

- GESTIONE della `processTable` e `waitpid`:

`common_prog` (interna al kernel) non ha bisogno di gestire il `pid` in quanto ha `*proc`; per la fine di un processo con `_exit/sys_exit` non serve la `waitpid` se ad aspettare è il kernel (che ha `*proc`) [basta `proc_wait`] `waitpid` serve per la gestione di processi da parte di programmi user; `testbin/forktest` permette di verificare la `waitpid`, ma serve realizzare la `getpid` (che ottiene il `pid` del `curproc`) e la `fork` (che genera un processo figlio a livello user)

Un modo semplice per testare (senza creare la `fork`) la `sys_waitpid` (e **non direttamente** la `waitpid`) chiamata in `syscall`, è ottenere in `common_prog` il `pid` del processo figlio [con `sys_getpid`] e attendere la `sys_waitpid` anziché `proc_wait`

Soluzione (proposta):

```
pid_t sys_getpid(void) {
#ifndef OPT_WAITPID
    KASSERT(curproc != NULL);
    return curproc->p_pid;
#else
    return -1;
#endif
}
```