Sistemi Operativi

Lezione 22: i processi in JOS (ENVS)

Processi

- Con il termine processo si denota l'esecuzione di un programma (sequenza di istruzioni) nell'ambito di un determinato ambiente esecutivo caratterizzato da:
  - CPU
  - Memoria
  - File
- I processi sono tra loro scorrelati, cioè le risorse di ciascun processo sono private

Processi

- I processi sono gli oggetti usati dal sistema per eseguire tutte le attività richieste dall'utente :
  - Web browser
  - Word processors
  - Mail, ecc.

Stato di un processo (1)

- Le informazioni per l'identificazione del processo contengono:
  - Identificatore del processo
  - Identificatore del processo padre
  - Identificatore dell'utente proprietario del processo

Stato di un processo (2)

- Lo stato del processore
  - Tutti i registri del processore
  - PC (EIP)
  - Condition codes (EFLAGS)
  - Variabili di stato: flag di interrupt, execution mode
  - Stack pointer: puntatori allo stack associato al processo

Stato di un processo (3)

- Process Control Information
  - Process state: running, ready, waiting, halted.
  - Priorità di scheduling
  - Informazioni per l'algoritmo di scheduling: tempo di permanenza nel sistema, tempo di CPU,
  - Eventi: identificativo dell'evento di cui il processo è eventualmente in attesa

Stato di un processo (4)

- Process Control Information
  - Campi di strutture dati: puntatori utilizzati quando il PCB è inserito in code di attesa
  - Variabili per la comunicazione tra processi
  - Eventuali privilegi concessi al processo: quantità di memoria, risorse del sistema

Stato di un processo (5)

- Process Control Information
  - Gestione della Memoria
    - Tabelle delle pagine o dei segmenti
  - Risorse utilizzate
    - File aperti e/o creati

JOS PCB (ENV)

```
struct Env {
    struct Trapframe env_tf;    // Saved registers
    struct Env *env_link;       // Next free Env
    envid_t env_id;             // Unique environment identifier
    envid_t env_parent_id;      // env_id of this env's parent
    enum EnvType env_type;      // Indicates special system enviro
    unsigned env_status;        // Status of the environment
    uint32_t env_runs;          // Number of times environment has
    int env_cpunum;             // The CPU that the env is running on

    // Address space
    pde_t *env_pgdir;           // Kernel virtual address of page dir

    // Exception handling
    void *env_pgfault_upcall;   // Page fault upcall entry point

    // Lab 4 IPC
    bool env_ipc_recving;       // Env is blocked receiving
    void *env_ipc_dstva;        // VA at which to map received page
    uint32_t env_ipc_value;     // Data value sent to us
    envid_t env_ipc_from;       // envid of the sender
    int env_ipc_perm;           // Perm of page mapping received
};
```

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

env data structure (PCB)

```
#include <inc/types.h>

struct PushRegs {
    /* registers as pushed by pusha */
    uint32_t reg_edi;
    uint32_t reg_esi;
    uint32_t reg_ebp;
    uint32_t reg_oesp;      /* Useless */
    uint32_t reg_ebx;
    uint32_t reg_edx;
    uint32_t reg_ecx;
    uint32_t reg_eax;
} __attribute__((packed));

struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to
        kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

env_status

ENV_FREE: Indicates that the Env structure is inactive, and therefore on the env_free_list.

ENV_RUNNABLE: Indicates that the Env structure represents an environment that is waiting to run on the processor.

ENV_RUNNING: Indicates that the Env structure represents the currently running environment.

ENV_NOT_RUNNABLE: Indicates that the Env structure represents a currently active environment, but it is not currently ready to run: for example, because it is waiting for an interprocess communication (IPC) from another environment.

ENV_DYING: Indicates that the Env structure represents a zombie environment

Data structures

- The kernel maintains three main global variables pertaining to environments:
  - `struct Env *envs = NULL; // All environments`
  - `struct Env *curenv = NULL; //The current env`
  - `static struct Env *env_free_list; // Free // environment list`

Env List

- Once JOS gets up and running, the `envs pointer` points to an array of Env structures representing all the environments in the system
- The JOS kernel will support a maximum of NENV simultaneously active environments, although there will typically be far fewer running environments at any given time. (NENV is a constant defined in inc/env.h.)
- Once it is allocated, the envs array will contain a single instance of the Env data structure for each of the NENV possible environments

curenv

- The JOS kernel keeps all of the inactive Env structures on the env_free_list. This design allows easy allocation and deallocation of environments, as they merely have to be added to or removed from the free list
- The kernel uses the `curenv` symbol to keep track of the *currently executing* environment at any given time. During boot up, before the first environment is run, curenv is initially set to NULL

Env structures

- We previously allocated memory in `mem_init()` for the `pages[]` array, which is a table the kernel uses to keep track of which pages are free and which are not
- We need to modify `mem_init()` further to allocate a similar array of `Env` structures, called envs

Map envs

```
//////////////////////////////////////
// Make 'envs' point to an array of size 'NENV'
// of 'struct Env'.
envs = boot_alloc (NENV * sizeof(struct Env));

//////////////////////////////////////////////////
// Map the 'envs' array read-only by the user at linear
// address UENVS (ie. perm = PTE_U | PTE_P).
boot_map_region (kern_pgdir, UENVS,
       ROUNDUP(NENV*sizeof(struct Env), PGSIZE),
       PADDR(envs), PTE_U);
```

Process Management

- env_init()
  - Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).
- env_setup_vm()
  - Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.
- region_alloc()
  - Allocates and maps physical memory for an environment

Process Management

- `load_icode()`
  - You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.
- `env_create()`
  - Allocate an environment with `env_alloc` and call `load_icode` load an ELF binary into it
- `env_run()`
  - Start a given environment running in user mode.

env_init

```
// Mark all environments in 'envs' as free, set their
// env_ids to 0,  and insert them into the
// env_free_list.

// Make sure the environments are in the free list in
// the same order  they are in the envs array (i.e., so
// that the first call to env_alloc() returns envs[0]).
```

env_init

```
void
env_init(void)
{
// Set up envs array
int i;
env_free_list = envs;
for (i = 0; i<NENV; i++){
    (envs+i)->env_id = 0;
    (envs+i)->env_status = ENV_FREE;
    (envs+i)->env_link = (envs + i + 1);
    }
    (envs + i -1)->env_link = NULL;
// Per-CPU part of the initialization
    env_init_percpu();
}
```

env_init_percpu

```
/ Load GDT and segment descriptors.
Void env_init_percpu(void)
{   lgdt(&gdt_pd);
// The kernel never uses GS or FS, so we leave those set to
// the user data segment.
    asm volatile("movw %%ax,%%gs" :: "a" (GD_UD|3));
    asm volatile("movw %%ax,%%fs" :: "a" (GD_UD|3));
// The kernel does use ES, DS, and SS.  We'll change between
// the kernel and user data segments as needed.
    asm volatile("movw %%ax,%%es" :: "a" (GD_KD));
    asm volatile("movw %%ax,%%ds" :: "a" (GD_KD));
    asm volatile("movw %%ax,%%ss" :: "a" (GD_KD));
// Load the kernel text segment into CS.
    asm volatile("ljmp %0,$1f\n 1:\n" :: "i" (GD_KT));
// For good measure, clear the local descriptor table (LDT),
// since we don't use it.
    lldt(0);
}
```

```
struct Segdesc gdt[NCPU + 5] =
{
    // 0x0 - unused (always faults -- for trapping NULL far pointers)
    SEG_NULL,
    // 0x8 - kernel code segment
    [GD_KT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 0),
    // 0x10 - kernel data segment
    [GD_KD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 0),
    // 0x18 - user code segment
    [GD_UT >> 3] = SEG(STA_X | STA_R, 0x0, 0xffffffff, 3),
    // 0x20 - user data segment
    [GD_UD >> 3] = SEG(STA_W, 0x0, 0xffffffff, 3),
    // Per-CPU TSS descriptors (starting from GD_TSS0) are initialized
    // in trap_init_percpu()
    [GD_TSS0 >> 3] = SEG_NULL
};

struct Pseudodesc gdt_pd = {
    sizeof(gdt) - 1, (unsigned long) gdt
};
```

Definitions

```
/*
 * This file contains definitions for memory management in our OS,
 * which are relevant to both the kernel and user-mode software.
 */

// Global descriptor numbers
#define GD_KT     0x08     // kernel text
#define GD_KD     0x10     // kernel data
#define GD_UT     0x18     // user text
#define GD_UD     0x20     // user data
#define GD_TSS0   0x28     // Task segment selector for CPU 0
```

env_setup_vm

```
//
// Initialize the kernel virtual memory layout for
// environment e.
// Allocate a page directory, set e->env_pgdir
// accordingly,
// and initialize the kernel portion of the new
// environment's address space.
// Do NOT (yet) map anything into the user portion
// of the environment's virtual address space.
//
// Returns 0 on success, < 0 on error.  Errors include:
// -E_NO_MEM if page directory or table could not be
// allocated.
//
```

env_setup_vm

```
static int
env_setup_vm(struct Env *e)
{
int i;
struct PageInfo *p = NULL;
// Allocate a page for the page directory
if (!(p = page_alloc(ALLOC_ZERO)))
    return -E_NO_MEM;
p->pp_ref ++;
e->env_pgdir = page2kva(p);
for ( i = PDX(UTOP); i < NPDENTRIES; i++)
    e->env_pgdir[i] = kern_pgdir[i];

// UVPT maps the env's own page table read-only.
// Permissions: kernel R, user R
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
    return 0;
    }
```

env_alloc

```
// Allocates and initializes a new environment.
// On success, the new environment is stored in *newenv_store.
//
// Returns 0 on success, < 0 on failure.  Errors include:
//  -E_NO_FREE_ENV if all NENVS environments are allocated
//  -E_NO_MEM on memory exhaustion
//
    int
    env_alloc(struct Env **newenv_store, envid_t parent_id)
    {
    int32_t generation;
    int r;
    struct Env *e;
    if (!(e = env_free_list))
        return -E_NO_FREE_ENV;
```

env_alloc

```
// Allocate and set up the page directory for this environment.
    if ((r = env_setup_vm(e)) < 0)
        return r;
// Generate an env_id for this environment.
generation = (e->env_id + (1 << ENVGENSHIFT)) & ~(NENV - 1);
if (generation <= 0)  // Don't create a negative env_id.
    generation = 1 << ENVGENSHIFT;
e->env_id = generation | (e - envs);
// Set the basic status variables.
e->env_parent_id = parent_id;
e->env_type = ENV_TYPE_USER;
e->env_status = ENV_RUNNABLE;
e->env_runs = 0;
// Clear out all the saved register state, to prevent the register
// values of a prior environment inhabiting this Env structure
// from "leaking" into our new environment.
    memset(&e->env_tf, 0, sizeof(e->env_tf));
```

env_alloc

```
e->env_tf.tf_ds = GD_UD | 3;
e->env_tf.tf_es = GD_UD | 3;
e->env_tf.tf_ss = GD_UD | 3;
e->env_tf.tf_esp = USTACKTOP;
e->env_tf.tf_cs = GD_UT | 3;
// You will set e->env_tf.tf_eip later.
// Enable interrupts while in user mode.
e->env_tf.tf_eflags |= FL_IF;
// Clear the page fault handler until user installs one.
e->env_pgfault_upcall = 0;
// Also clear the IPC receiving flag.
e->env_ipc_recving = 0;
// commit the allocation
env_free_list = e->env_link;
*newenv_store = e;
```

sys_exofork

```
static envid_t
sys_exofork(void)
{
// Create the new environment with env_alloc(),
// from kern/env.c. It should be left as
// env_alloc created it, except that
// status is set to ENV_NOT_RUNNABLE, and the
// register set is copied
// from the current environment -- but tweaked so
// sys_exofork  will appear to return 0.
struct Env *newenv;
```

sys_exofork

```
int result = env_alloc(&newenv, thiscpu->cpu_env->env_id);
    if (result){
        warn ("sys_exofork: problems with env_alloc\n");
        return result;
        }

    newenv->env_status = ENV_NOT_RUNNABLE;
    newenv->env_tf = thiscpu->cpu_env->env_tf;
    newenv->env_tf.tf_regs.reg_eax = 0;
    return newenv->env_id;
}
```

```
envid_t
dumbfork(void)
{
	envid_t envid;
	uint8_t *addr;
	int r;
	extern unsigned char end[];

	// Allocate a new child environment.
	// The kernel will initialize it with a copy of our register state,
	// so that the child will appear to have called sys_exofork() too -
	// except that in the child, this "fake" call to sys_exofork()
	// will return 0 instead of the envid of the child.
	envid = sys_exofork();
	if (envid < 0)
		panic("sys_exofork: %e", envid);
	if (envid == 0) {
		// We're the child.
		// The copied value of the global variable 'thisenv'
		// is no longer valid (it refers to the parent!).
		// Fix it and return 0.
		thisenv = &envs[ENVX(sys_getenvid())];
		return 0;
	}

	// We're the parent.
	// Eagerly copy our entire address space into the child.
	// This is NOT what you should do in your fork implementation.
	for (addr = (uint8_t*) UTEXT; addr < end; addr += PGSIZE)
		duppage(envid, addr);

	// Also copy the stack we are currently running on.
	duppage(envid, ROUNDDOWN(&addr, PGSIZE));

	// Start the child environment running
	if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
		panic("sys_env_set_status: %e", r);

	return envid;
}
```

region_alloc

```
// Allocate len bytes of physical memory for environment env,
// and map it at virtual address va in the environment's
// address space.
// Does not zero or otherwise initialize the mapped pages in
// any way.  Pages should be writable by user and kernel.
// Panic if any allocation attempt fails.
static void
region_alloc(struct Env *e, void *va, size_t len)
{
uint32_t virt_add = ROUNDDOWN((uint32_t)va, PGSIZE);
uint32_t size = ROUNDUP((uint32_t)len, PGSIZE);
int i;
for ( i=0; i<size; i = i+PGSIZE) {
 if (page_insert(e->env_pgdir, page_alloc(size), (void *) virt_add+i,
                                                    PTE_U|PTE_W));
    panic("page_insert error during region_alloc\n");
                        }
    }
```

load_icode

```
//
// Set up the initial program binary, stack, and
// processor flags for a user process.
// Load each program segment into virtual memory
// at the address specified in the ELF section
// header.
// You should only load segments with ph->p_type
// == ELF_PROG_LOAD. Each segment's virtual
// address can be found in ph->p_va and its size
// in memory can be found in ph->p_memsz.
// Loading the segments is much simpler if you
// can move data directly into the virtual
// addresses stored in the ELF binary.
```

```
static void
load_icode(struct Env *e, uint8_t *binary, size_t size)
{
#define PROGHDR     ((struct Elf *) binary)
struct Proghdr *ph, *eph;
ph = (struct Proghdr *) ((uint8_t *) PROGHDR + PROGHDR->e_phoff);
eph = ph + PROGHDR->e_phnum;
lcr3(PADDR(e->env_pgdir));
```

load_icode

```
for (; ph<eph; ph++)
{ if (ph->p_type == ELF_PROG_LOAD )
   {
     region_alloc( e, (void *)ph->p_va, ph->p_memsz);
     memset ((void *)ph->p_va, 0, ph->p_memsz);
     memcpy ((void *)ph->p_va, binary+ph->p_offset,
                                ph->p_filesz);
       }
   }
lcr3(PADDR(kern_pgdir));
e->env_tf.tf_eip = PROGHDR->e_entry;
region_alloc (e, (void *)USTACKTOP-PGSIZE, PGSIZE);
}
```

env_run

```
//
// Context switch from curenv to env e.
// Note: if this is the first call to env_run,
curenv is NULL.
//
// This function does not return.
//
void
env_run(struct Env *e)
{
```

env_run

```
if (curenv && (curenv->env_status == ENV_RUNNING))
        curenv->env_status = ENV_RUNNABLE;

    curenv = e;
    curenv->env_status = ENV_RUNNING;
    curenv->env_runs ++;
    lcr3(PADDR(curenv->env_pgdir));

    unlock_kernel();
    env_pop_tf (&curenv->env_tf);
}
```

```
// Restores the register values in the Trapframe with the 'iret' instruction.
// This exits the kernel and starts executing some environment's code.
//
// This function does not return.
//
void
env_pop_tf(struct Trapframe *tf)
{
    // Record the CPU we are running on for user-space debugging
    curenv->env_cpunum = cpunum();

    __asm __volatile("movl %0,%%esp\n"
        "\tpopal\n"
        "\tpopl %%es\n"
        "\tpopl %%ds\n"
        "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
        "\tiret"
        : : "g" (tf) : "memory");
    panic("iret failed");  /* mostly to placate the compiler */
}
```

env_free

```
//
// Frees env e and all memory it uses.
//
void
env_free(struct Env *e)
{
	pte_t *pt;
	uint32_t pdeno, pteno;
	physaddr_t pa;

	// If freeing the current environment, switch to kern_pgdir
	// before freeing the page directory, just in case the page
	// gets reused.
	if (e == curenv)
		lcr3(PADDR(kern_pgdir));
	// Note the environment's demise.
	// cprintf("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e->env_id);
	// Flush all mapped pages in the user portion of the address space
	static_assert(UTOP % PTSIZE == 0);
	for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
```

env_free

```
// only look at mapped page tables
        if (!(e->env_pgdir[pdeno] & PTE_P))
            continue;

        // find the pa and va of the page table
        pa = PTE_ADDR(e->env_pgdir[pdeno]);
        pt = (pte_t*) KADDR(pa);

        // unmap all PTEs in this page table
        for (pteno = 0; pteno <= PTX(~0); pteno++) {
            if (pt[pteno] & PTE_P)
                page_remove(e->env_pgdir, PGADDR(pdeno, pteno, 0));
        }

        // free the page table itself
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa));
    }
```

env_free

```
    // free the page directory
    pa = PADDR(e->env_pgdir);
    e->env_pgdir = 0;
    page_decref(pa2page(pa));

    // return the environment to the free list
    e->env_status = ENV_FREE;
    e->env_link = env_free_list;
    env_free_list = e;
}
```