Notes on Unix V6 for X86

Contents

1 Introduction

1.1 Problem statement

- **What we want to do**:
  - Multiple jobs (abtract notion) to do (OPTIONAL: owned by multiple users)
  - Jobs are given in the form of a "**program**" (in any possible form – e.g. in a disk)
- **What we are given**:
  - von-Neumann, stored-program machine (processor, memory, I/O)
- **Constraints**:
  - Protection: program executions should be protected from each other; also, kernel should be protected from program exeuctions
  - No deadlock, etc.

1.2 Solution: OS

- **TAPPING (DOWNWARD)**: HW (processor + devices + console + etc.) is just an *autonomous, free-running pieces of metal*.
  - Suppose you want to "use" these pieces of metal. Then, you need to add a "tap" to this free-running wheel, through which you do something useful. OS allows that.
  - "Stored program concept" already adds one level of abstraction (compare this to a circuitry which performs a single fixed-function).
  - **WHAT MACHINE CAN DO**:
    - If you put instruction on memory location and set EIP to that location, the machine will execute it automatically.
  - **WHAT TO DO TO USE THE MACHINE**:
    - Put (well-planned) instructions on memory and set EIP.
    - determine the meaning of "well-planned instructions"
    - Now, the notion of "program" comes in and program-to-machine-instruction mapping comes to play.
- **INTERFACE (UPWARD)**: OS provides "services" to user programs through an **interface**.
  - interfaces must be simple and narrow
  - but users want many sophisticated features
  - HOW TO RECONCILE THESE CONFLICTING NEEDS:
    - select a good set of small interfaces which can be **combined** to support any sophisticated feature
    - find a **basis** set from which every thing can be combined (*efficiently*)
- **YET ANOTHER PROGRAM**: An OS is yet another program. Just like an interpreter is yet another program.
  - Consider a simple example – uniprocessor, single-threaded system.
  - From HW viewpoint, there is only one program running – a program named "kernel".
  - Kernel allows multiple processes to "share" HW through **multiplexing**.
    - SHARE $\equiv$ MULTIPLEX
  - that is, it gives an illusion that that multiple processes are running (multiprogramming) even if actually a single program called "kernel" is running
- **SYSTEM CALLS**
  - processes: fork, exit, wait, kill, sbrk, sleep, getpid
  - files: ope, read, write, close, dup, mkdir, chdir, mknod, fstat, link, unlink
- **SHELL**
  - **THOUGHT EXPERIMENT**: trace in your brain how a shell is implemented on Unix
  - it has all essentials – process creation, deletion, resource allocation, etc.
  - provides the interaction between USERs and MACHINES (via OS)

1.3 x86 calling convention

- **HARDWARE** (x86) provides **stack registers** (`%esp`, `%ebp`), and **stack instructions** (`push`, `pop`, `call`, verb+ret+)

```
EXAMPLE INSTRUCTION    WHAT IT EFFECTIVELY DOES
-------------------    ------------------------
pushl %eax             subl $4, %esp          ; stack grows down
                       movl %eax, (%esp)
popl %eax              movl (%esp), %eax
                       addl $4, %esp
call 0x12345           pushl %eip
                       movl $0x12345, %eip
ret                    popl %eip
```

- **COMPILER** (GCC) dictates how the stack is used. Contract between caller and callee on x86:
  - **at entry to a function** (i.e. right after **`call`**instruction is executed)
    * `%eip` points at first instruction of function (`call` pushes `%eip` which is the **return address**)
    * `%esp`4+ points at first argument
    * `%esp` points at return address
  - **right after `ret` instruction is executed**
    * `%eip` contains return address
    * `%esp` points at arguments pushed by caller
    * CALLEE may have trashed arguments
    * `%eax` (and `%edx`, if return type is 64-bit) contains return value (or trash if function is void)
    * **caller-saved registers** (`%eax`, `%edx` (above), and `%ecx`) may be trashed during the "call"
      · so, in case caller have been using these registers, caller should save it (and restore it later before it will use it again)
      · COMPILER must insert save/restore code in CALLER-side
    * **callee-saved registers** (`%ebp`, `%ebx`, `%esi`, `%edi`) must contain contents from time of **`call`**
      · i.e. callee should have properly restored these registers to contain pre-call values
      · COMPILER must insert save/restore code in CALLEE-side
- Functions can do anything that doesn't violate contract. By convention, GCC does more:
  - each function has a stack frame marked by `%ebp`, `%esp`

```
        +-------------+   |
        | arg 2       |   \
        +-------------+    >- previous function's stack frame
        | arg 1       |   /
        +-------------+   |
        | ret %eip    |   /  <= "CALL" pushed %eip
        +=============+
        | saved %ebp |   \  <= by "push %ebp" inside callee
%ebp->  +-------------+   |  <= by "move %esp, %ebp" in callee
        |             |   |
        |   local     |   \
        | variables,  |    >- current function's stack frame
        |    etc.     |   /
        |             |   |
        |             |   |
%esp->  +-------------+   /  <= by "stack alloc" in callee
```

  - `%esp` can move to make stack frame bigger, smaller

– `%ebp` points at saved `%ebp` from previous function, chain to walk stack
– **function prologue**: CALLEE performs upon entry

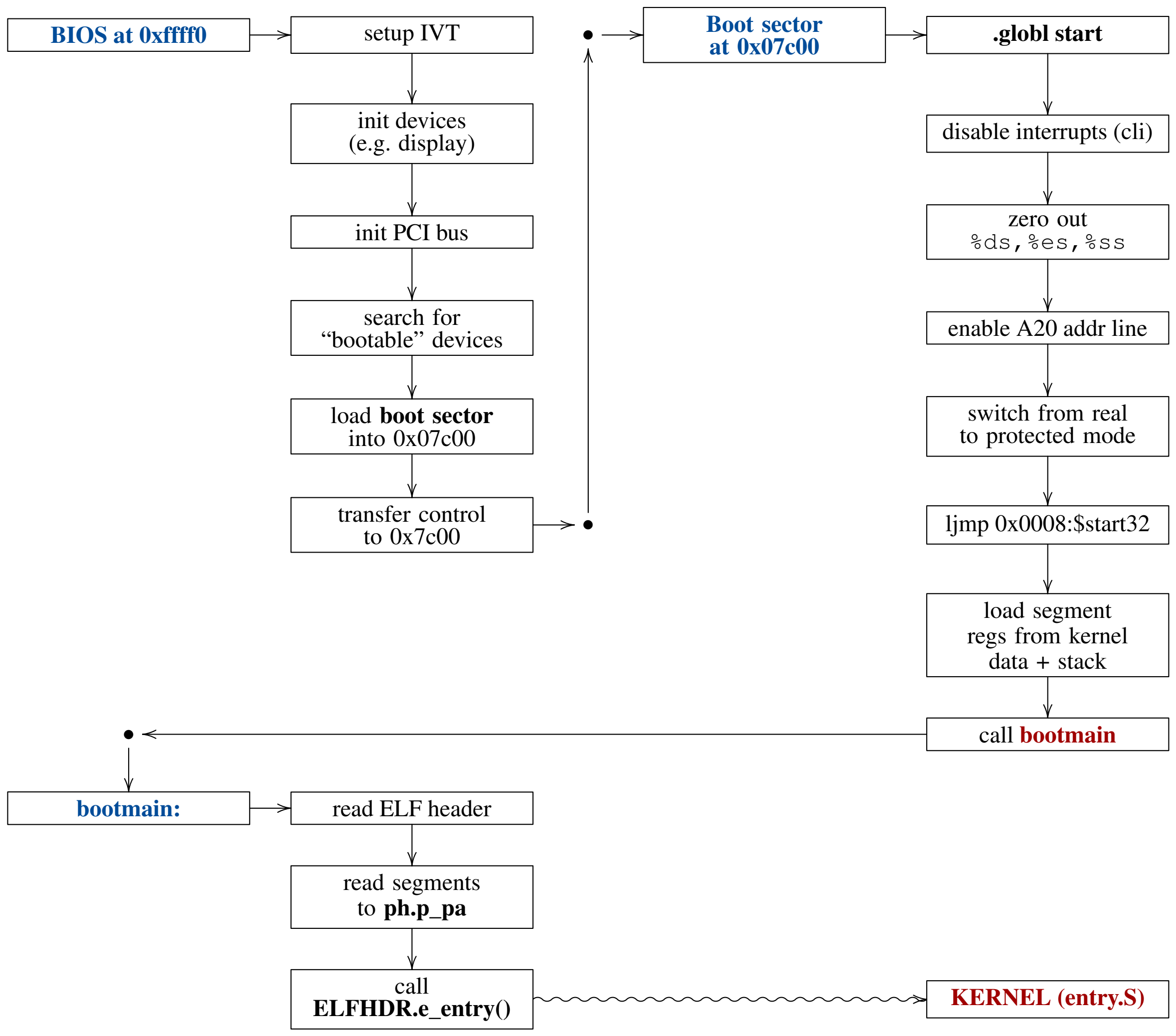
```
      pushl %ebp        ; push curr %ebp (%esp increments)
                                           ;
      movl %esp, %ebp
```

– textcolorred2function init setup: CALLEE prepares function execution; allows local vars, set current stackframe
– **function epilogue**: CALLEE performs upon exit

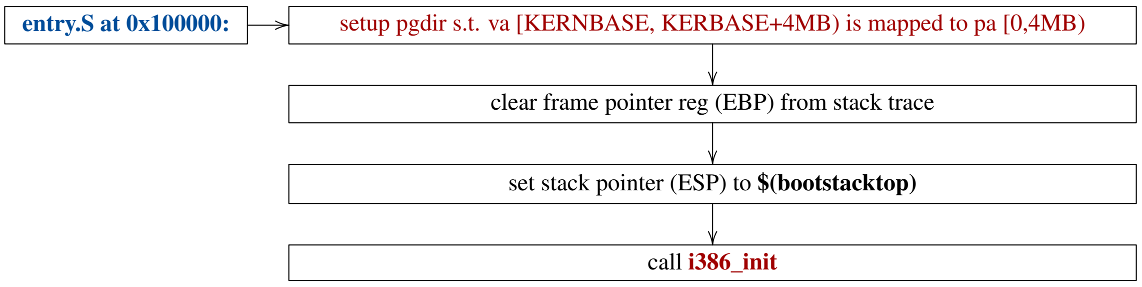
```
                          // can easily find return EIP on stack:
      movl %ebp, %esp
      popl %ebp
```

2 The First Process

2.1 Boot process



2.2 Kernel entry (entry.S)


entry.S at 0x100000:
setup pgdir s.t. va [KERNBASE, KERBASE+4MB) is mapped to pa [0,4MB)
clear frame pointer reg (EBP) from stack trace
set stack pointer (ESP) to $(bootstacktop)
call i386_init

- Entry sequence:

```
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

- Entry page table is defined in **main.c**.

```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

2.3 Kernel init (kern/init.c)

$$\begin{tikzcd}
\boxed{\textbf{i386\_init:}} \arrow[r] & \boxed{\text{cons\_init}} \arrow[r] \arrow[dd] & \boxed{\text{cga\_init}} \arrow[d] \\
 & & \boxed{\text{kbd\_init}} \arrow[d] \\
 & \boxed{\text{test\_backtrace}} \arrow[d] & \boxed{\text{serial\_init}} \\
 & \boxed{\text{monitor}} &
\end{tikzcd}$$

2.4 Console (kern/console.c)

$$\begin{tikzcd}
\boxed{\textbf{getchar:}} \arrow[r] & \boxed{\text{cons\_getc}} \arrow[r] & \boxed{\text{serial\_intr}} \arrow[r] \arrow[d] & \boxed{\text{cons\_instr(serial\_proc\_data)}} \\
 & & \boxed{\text{kbd\_intr}} \arrow[r] & \boxed{\text{cons\_instr(kbd\_proc\_data)}}
\end{tikzcd}$$

$$\begin{tikzcd}
\boxed{\textbf{cputchar:}} \arrow[r] & \boxed{\text{cons\_putc}} \arrow[r] & \boxed{\text{serial\_putc}} \arrow[d] \\
 & & \boxed{\text{lpt\_putc}} \arrow[d] \\
 & & \boxed{\text{cga\_putc}}
\end{tikzcd}$$

2.5 Monitor (kern/monitor.c)

$$\begin{tikzcd}
\boxed{\textbf{monitor:}} \arrow[r] & \boxed{\text{readline}} \arrow[r] \arrow[d] & \boxed{\text{getchar}} \arrow[r] \arrow[d] & \boxed{\text{cons\_getc}} \\
 & \boxed{\text{runcmd}} & \boxed{\textbf{cprintf}} \arrow[r] \arrow[d] & \boxed{\text{vcprintf}} \\
 & & \boxed{\text{cputchar}} \arrow[r] & \boxed{\text{cons\_putc}}
\end{tikzcd}$$

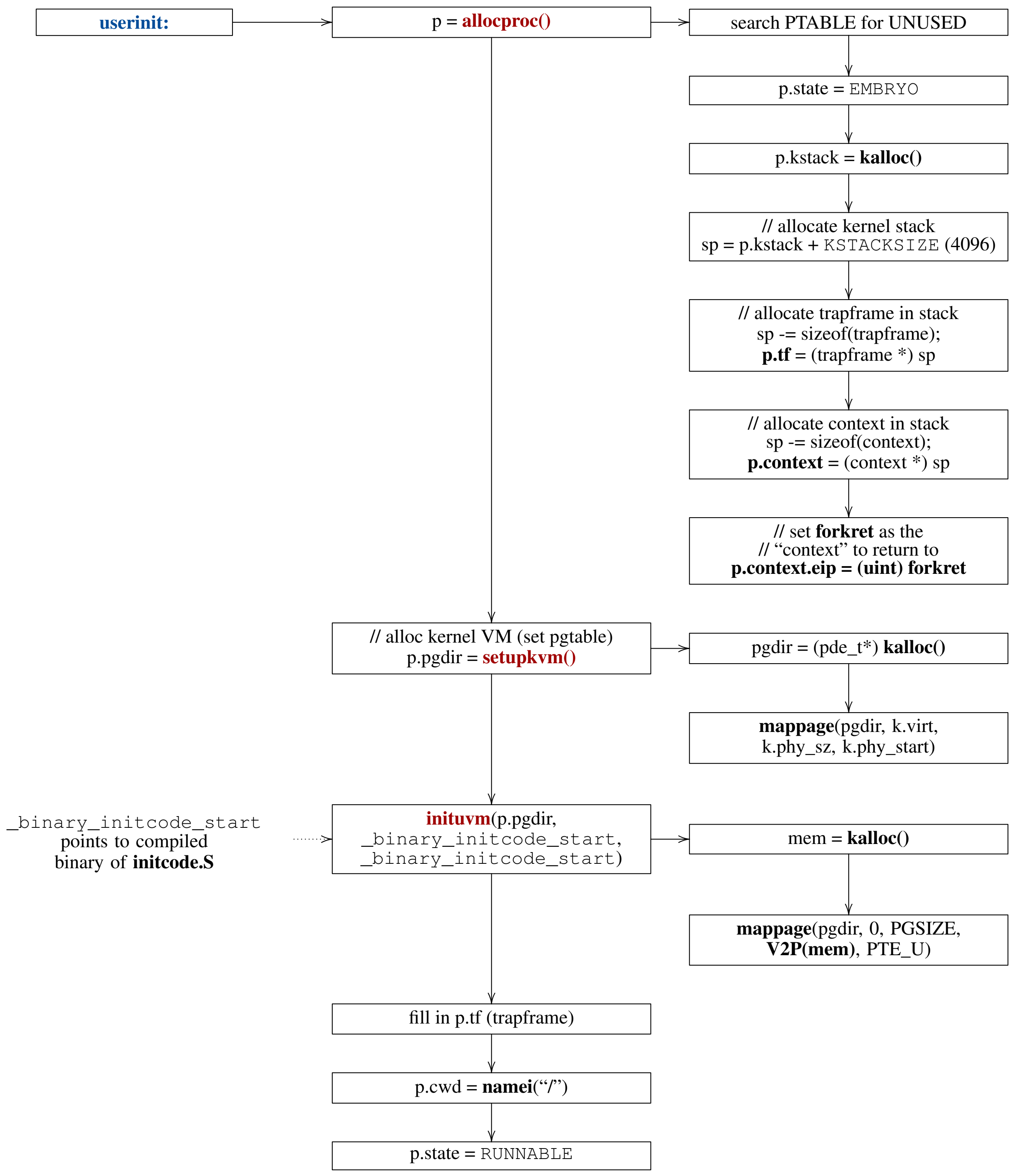
2.6 Printfmt (lib/printfmtc)

$$\begin{tikzcd}
\boxed{\textbf{cprintf:}} \arrow[r] & \boxed{\text{vcprintf}} \arrow[r] & \boxed{\text{vcprintfmt}} \arrow[r] & \boxed{\text{putch}} \arrow[d] \\
\boxed{\textbf{printfmt:}} \arrow[urr, bend right] & & & \boxed{\text{cputchar}}
\end{tikzcd}$$

2.7 proc: 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;        // State: running, ready, waiting for I/O, exiting
  volatile 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
  char name[16];               // Process name (debugging)
};

// saved registers for kernel context switches
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};

// Per-CPU state
struct cpu {
  uchar id;                    // Local APIC ID; index into cpus[] below
  struct context *scheduler;   // swtch() here to enter scheduler
  struct taskstate ts;         // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS];   // x86 global descriptor table
  volatile uint started;       // Has the CPU started?
  int ncli;                    // Depth of pushcli nesting.
  int intena;                  // Were interrupts enabled before pushcli?

  // CPU-local storage variables; see below
  struct cpu *cpu;
  struct proc *proc;           // The currently-running process.
};
```

2.8 Creating the first user process (main.c → proc.c)



userinit:
p = allocproc()
search PTABLE for UNUSED
p.state = EMBRYO
p.kstack = kalloc()
// allocate kernel stack
sp = p.kstack + KSTACKSIZE (4096)
// allocate trapframe in stack
sp -= sizeof(trapframe);
p.tf = (trapframe *) sp
// allocate context in stack
sp -= sizeof(context);
p.context = (context *) sp
// set forkret as the
// “context” to return to
p.context.eip = (uint) forkret
// alloc kernel VM (set pgtable)
p.pgdir = setupkvm()
pgdir = (pde_t*) kalloc()
mappage(pgdir, k.virt, k.phy_sz, k.phy_start)
_binary_initcode_start points to compiled binary of initcode.S
inituvm(p.pgdir, _binary_initcode_start, _binary_initcode_start)
mem = kalloc()
mappage(pgdir, 0, PGSIZE, V2P(mem), PTE_U)
fill in p.tf (trapframe)
p.cwd = namei(“/”)
p.state = RUNNABLE

- **allocproc** sets up the new process with a specially prepared kernel stack and set of kernel registers thta cause it to “return” to user space when it first runs

```
static struct proc *allocproc(void)
{
  struct proc *p;
  char *sp;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
  release(&ptable.lock);
  return 0;

found:
  p->state = EMBRYO;  p->pid = nextpid++;
  release(&ptable.lock);

  // Allocate kernel stack.
  if((p->kstack = kalloc()) == 0) {
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)trapret;

  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  memset(p->context, 0, sizeof *p->context);
  p->context->eip = (uint)forkret;
  return p;
}

// A fork child's very first scheduling by scheduler() will swtch
// here.  "Return" to user space.
void forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);
  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0; initlog();
  }
  // Return to "caller", actually trapret (see allocproc).
}
```

```
.globl trapret
trapret:
  // pop EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI
  // (as in bottom 8-words of trapframe)
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp # trapno and errcode
  iret
```

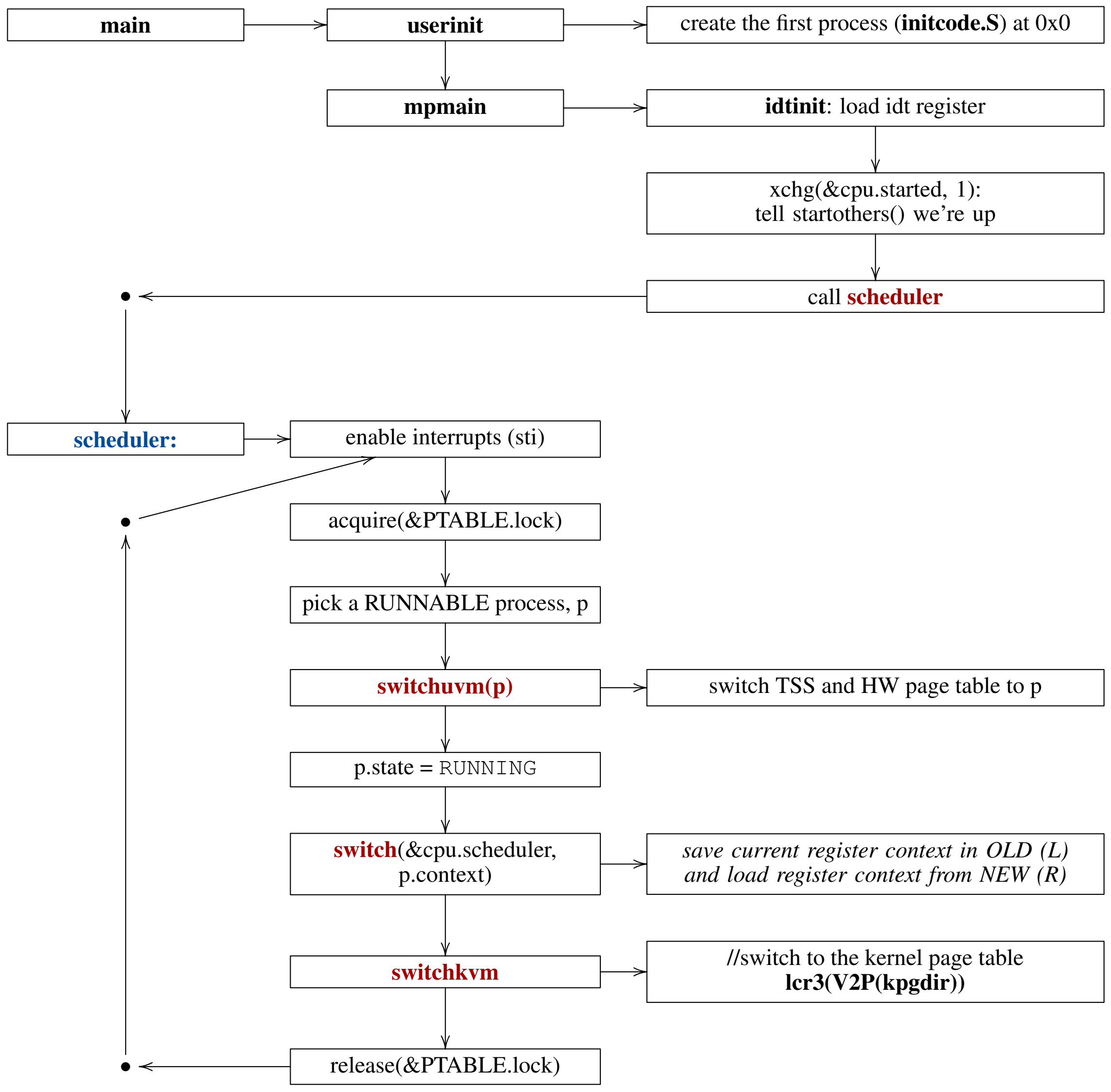
So, what happens when `fork` executed?

1. user code invokes system call **fork** (`int $T_SYSCALL`)
2. now in kernel space
3. **fork** calls **allocproc**
4. **allocproc** will set up kernel stack (see Figure 1-3)
    - will prepare two things: **forkret** and **trapret**
    - **forkret**: let `p->context.eip` point to address of function `forkret`
    - so, when `swtch(context **old, context*new)` is executed, it will context switch to a user function **forkret**
    - **forkret** will be a typical function and its **return** will look at the stack return address – which is **trapret**
    - so, after return from **forkret**, **trapret** code (assembly code defined in `vectors.S` will be executed
    - **trapret**: restores user registers from values stored at the top of the kernel stack and jumps into the process

First process vs. oridinary `fork`

- When setting trapframe in kernel stack,
    - first process: set EIP to user-space location 0
    - oridinary stack: set EIP to the instruction right after fork

2.9 Running the first process

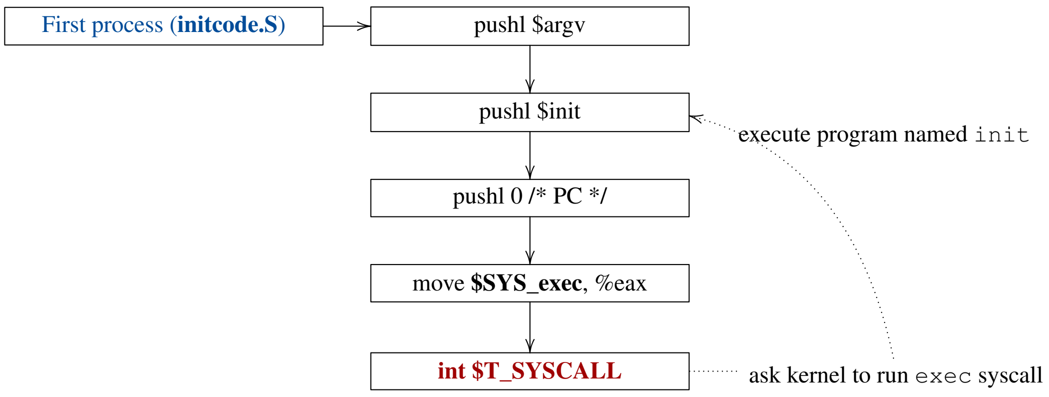


main
userinit
create the first process (initcode.S) at 0x0
mpmain
idtinit: load idt register
xchg(&cpu.started, 1):
tell startothers() we're up
call scheduler
scheduler:
enable interrupts (sti)
acquire(&PTABLE.lock)
pick a RUNNABLE process, p
switchuvm(p)
switch TSS and HW page table to p
p.state = RUNNING
switch(&cpu.scheduler, p.context)
save current register context in OLD (L) and load register context from NEW (R)
switchkvm
//switch to the kernel page table
lcr3(V2P(kpgdir))
release(&PTABLE.lock)

2.10 Context switch (switch.S)

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

2.11 The first system call: exec

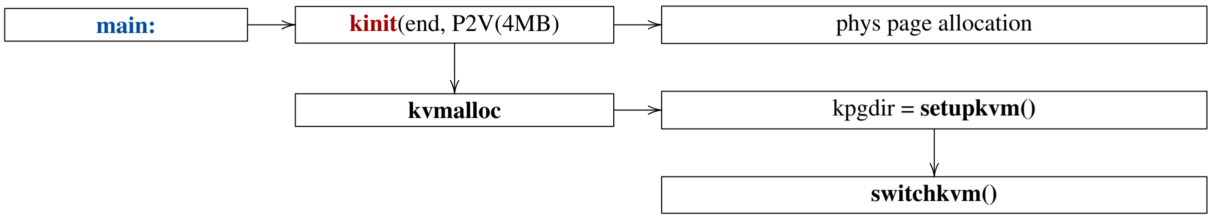

First process (initcode.S)
pushl $argv
pushl $init
execute program named init
pushl 0 /* PC */
move $SYS_exec, %eax
int $T_SYSCALL
ask kernel to run exec syscall

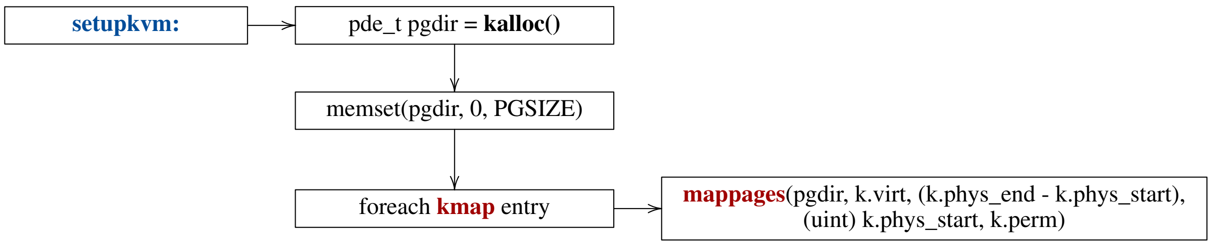
3 Virtual Memory

3.1 Basics

- **VIRTUAL MEMORY**: every xv6 process has an illusion that it has **its own address space**.
- x86 uses **paging** to implement virtual memory.
- paging is technically implemented by managing **per-process page tables** – i.e. set up page tages for each process
- **What HW provides for paging**
  - page directory register
- **What SW (OS) needs to provide for paging**:
  -

3.2 Setting up kernel virtual memory





setupkvm:
pde_t pgdir = kalloc()
memset(pgdir, 0, PGSIZE)
foreach kmap entry
mappages(pgdir, k.virt, (k.phys_end - k.phys_start), (uint) k.phys_start, k.perm)

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))`. The following table defines the kernel's mappings, which are present in every process's page table.

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W}, // I/O space
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},     // kern text+rodata
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W}, // kern data+memory
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W}, // more devices
};
```

3.3 Creating an address space by creating/populating page tables

Creates PTEs for virtual addresses starting at `va` that refer to physical addresses starting at `pa`. `va` and `size` may not be page-aligned.

```
static int 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;
}
```

The function **walkpgdir** returns the address of the PTE in page table pgdir that corresponds to virtual address va. If alloc is not 0, create any required page table pages.

```
// A virtual address 'la' has a three-part structure as follows:
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  -- PDX(va) --/ -- PTX(va) --/
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P) {
    pgtab = (pte_t*) p2v(PTE_ADDR(*pde));
  }
  else {
    if (!alloc || (pgtab = (pte_t*) kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

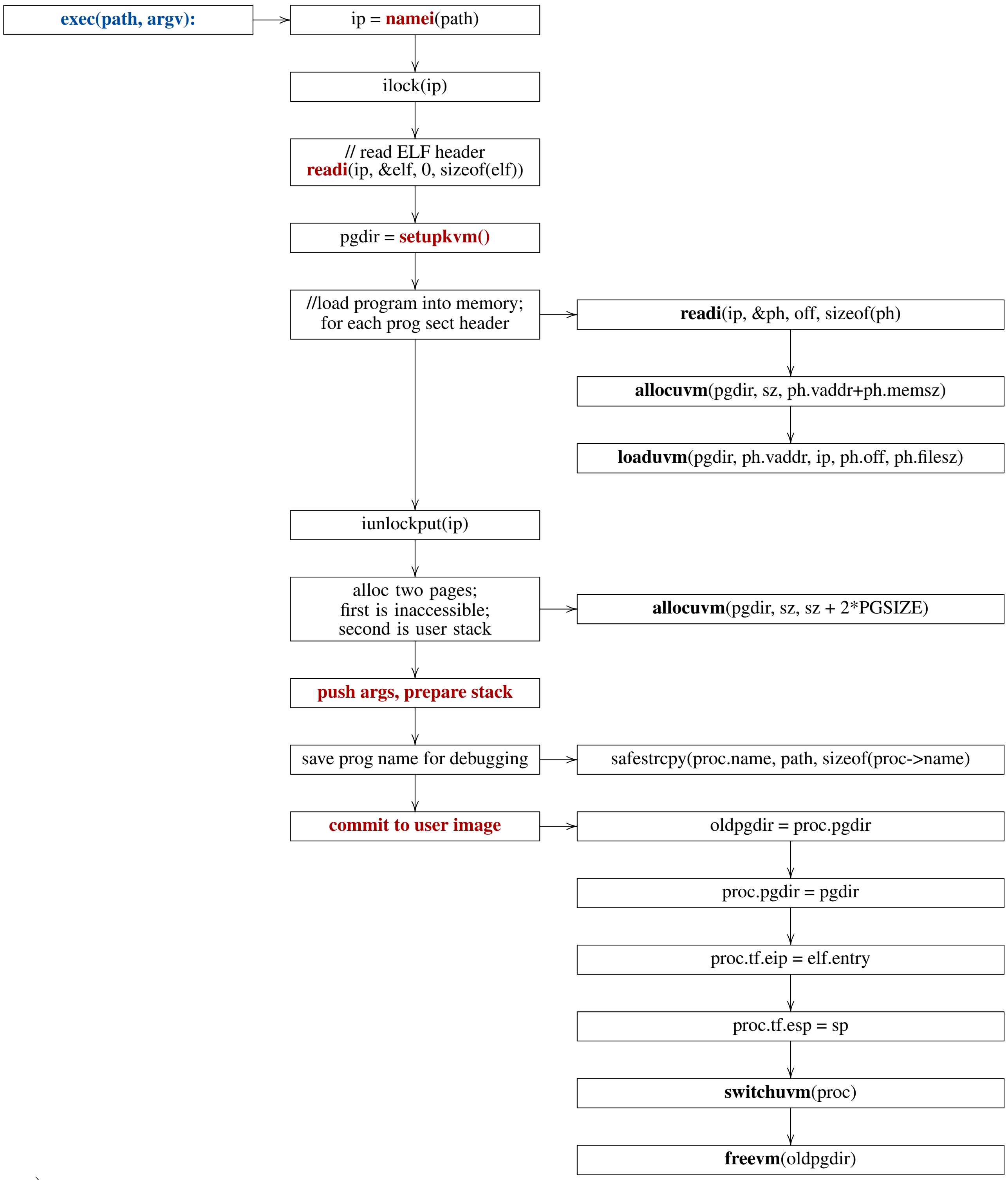
3.4 Paging in x86

- Since 10 bits for pgdir indices, there are 1024 pgdir's.
- Each pgdir is a table of 1024 pgdir entries (i.e. 1024 pgtbls), where each entry has 20 bits for PPN, and 12 bits for flags.
- Each pgtbl points to 1024 entries, where each entry points to one page. So, there are total $1024 \times 1024 = 1M$ pages.
- One page is 4K (12 bit offset), so, 4GB addressable.

3.5 Physical memory allocation

- all physical memory must be mapped (i.e. some page table should point to each physical page).
- page table with these mapping involves allocating page-table pages.

3.6 System call: exec – Creating user part of address space


exec(path, argv):
ip = namei(path)
ilock(ip)
// read ELF header
readi(ip, &elf, 0, sizeof(elf))
pgdir = setupkvm()
//load program into memory;
for each prog sect header
readi(ip, &ph, off, sizeof(ph)
allocuvm(pgdir, sz, ph.vaddr+ph.memsz)
loaduvm(pgdir, ph.vaddr, ip, ph.off, ph.filesz)
iunlockput(ip)
alloc two pages;
first is inaccessible;
second is user stack
allocuvm(pgdir, sz, sz + 2*PGSIZE)
push args, prepare stack
save prog name for debugging
safestrcpy(proc.name, path, sizeof(proc->name)
commit to user image
oldpgdir = proc.pgdir
proc.pgdir = pgdir
proc.tf.eip = elf.entry
proc.tf.esp = sp
switchuvm(proc)
freevm(oldpgdir)

)

3.7 allocuvm: Allocating user memory

```
// 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;

  // maximum process address space is [0, KERNBASE)
  if(newsz >= KERNBASE) return 0;
  if(newsz < oldsz) return oldsz;

  a = PGROUNDUP(oldsz);
  for (; a < newsz; a += PGSIZE) {
    mem = kalloc();  // allocate one page from kmem.freelist
    if(mem == 0) {
      cprintf("allocuvm out of memory");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);

    mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
  }
  return newsz;
}
```

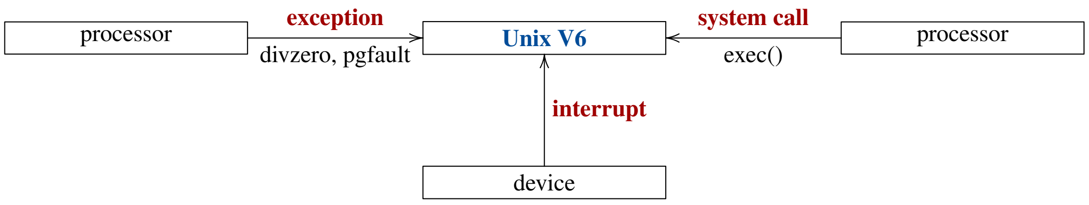
3.8 loaduvm: Loading user memory

```
// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  if ((uint) addr % PGSIZE != 0)
    panic("loaduvm: addr must be page aligned");
  for (i = 0; i < sz; i += PGSIZE) {
    if ((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
      panic("loaduvm: address should exist");
    pa = PTE_ADDR(*pte);
    if (sz - i < PGSIZE)
      n = sz - i;
    else
      n = PGSIZE;
    if (readi(ip, p2v(pa), offset+i, n) != n)
      return -1;
  }
  return 0;
}
```

3.9 System call: exec – Calling convention

4 Traps, Interrupts, and Device Drivers

4.1 System calls, exceptions, and interrupts

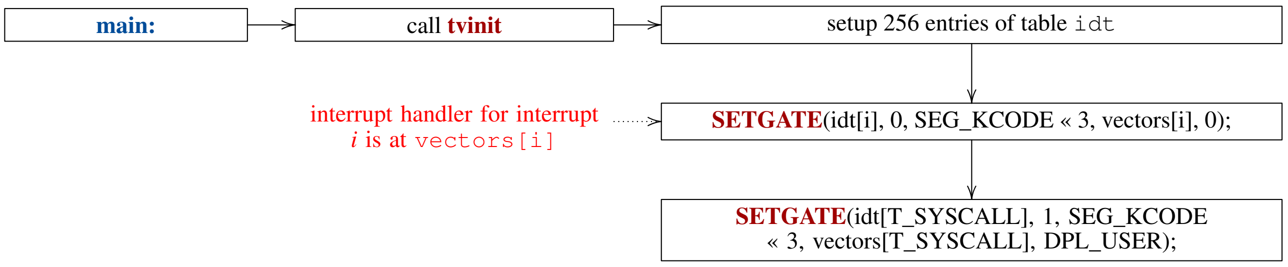


4.2 System call example: exec (initcode.S)

```
# exec(init, argv)
pushl $argv             // push arg on process's stack
pushl $init             //
pushl $0                // where caller pc would be
movl $SYS_exec, %eax    // put system call numuber, i.e.
                        // index to syscalls array, which
                        // is a table of function ptrs
int $T_SYSCALL          // generate "SYSCALL" trap
```

This interrupt (`int $T_SYSCALL`) will eventually trigger the `sys_exec` function shown in Section 3.6.

4.3 Assembly trap handlers


main:
call tvinit
setup 256 entries of table idt
interrupt handler for interrupt i is at vectors[i]
SETGATE(idt[i], 0, SEG_KCODE « 3, vectors[i], 0);
SETGATE(idt[T_SYSCALL], 1, SEG_KCODE « 3, vectors[T_SYSCALL], DPL_USER);

4.4 Setting up interrupt gate descriptor

```
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d) {
  (gate).off_15_0 = (uint)(off) & 0xffff;
  (gate).cs = (sel);
  (gate).args = 0;
  (gate).rsv1 = 0;
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;
  (gate).s = 0;
  (gate).dpl = (d);
  (gate).p = 1;
  (gate).off_31_16 = (uint)(off) >> 16; }
```

4.5 What happens on system calls, exceptions, and interrupts

1. **put arguments**: put the arguments
2. **interrupt generation – `int`** *n*: *kernel and user-process (or device) are independent, autonomous entities; so, for user-process to get service from kernel, it needs to tell them "I need your help" – this is interrupt generation*;
   from processor's viewpoint, interrupt is an **asynchronous** request
   (a) fetch *n*'s descriptor from the **IDT (Interrupt Descriptor Table)**, where *n* is the argument of `int`
   (b) check priviledge level
   (c) save `%ss` and `%esp` in CPU-internal registers
   (d) load `%ss` and `%esp` from a task segment descriptor
   (e) push `%ss`, `%esp`, `%eflags`, `%cs`, `%eip`
   (f) push `%cs` and `%eip` to the value in the descriptor

   stopped (by H/W mechanism)
3. **get arguments**: the kernel must be able to retrieve ifnormation about the event (e.g. system call arguments)

4.6 IDT: Interrupt descriptor table

IDT entries is setup by an assembly code `vectors.S`, which are generated by the Perl script `vectors.pl`.

```
# entry point to vector 0
.globl vector0
vector0:
  pushl $0       #
  pushl $0       # trap no
  jmp alltraps
# entry point to vector 1
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
...
vector8:
  pushl $8
  jmp alltraps
```

4.7 trapasm.S

```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

4.8 Trapframes

A trapframe (`struct trapframe`) captures the contents of the processor registers at the time of the trap. Actually, this structure "interprets" the stack contents as a parameter. Some parts of the frame is filled up by the processor (according to the calling convention), while others are filled up by `alltraps` code.

Typically, for a function call (i.e. "jumping to new code"), when C-compiler generated calling convention (pushing eip, return address, etc.) is enough, you can just call them and let C-compiler generate code. *However, when it's not enough, you need to generate the code and mimic the calling convention yourself in an assembly program. Trap handling is one of such situation.*

```
// Layout of the trap frame built on the stack by the
// hardware and by trapasm.S, and passed to trap().
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame (pushed by alltraps)
  ushort gs;   ushort padding1;
  ushort fs;   ushort padding2;
  ushort es;   ushort padding3;
  ushort ds;   ushort padding4;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;   ushort padding5;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;   ushort padding6;
};
```

4.9 trap: Execution of trap

```
void trap(struct trapframe *tf)
{
  if (tf->trapno == T_SYSCALL) {
    if (proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if (proc->killed)
      exit();
    return;
  }
  switch (tf->trapno) {
  case T_IRQ0 + IRQ_TIMER:
  ...
  }
}
```

4.10 System calls in Linux

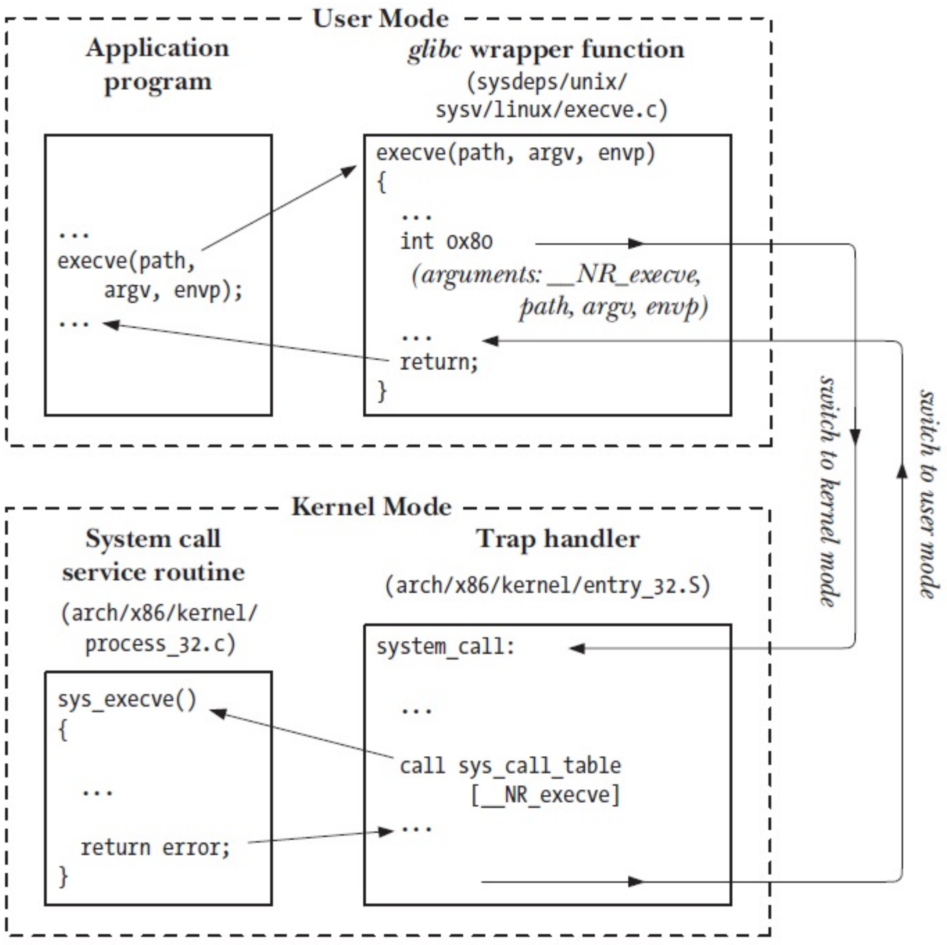
General points

- A system call changes the processor state from **user mode** to **kernel mode**, so that the CPU can access protected kernel memory.
- Each system call may have a set of arguments that specify information to be transferred from user space (the **process's virtual address space**) to kernel space and vice versa.

What happens on a system call

1. The application program makes a system call by invoking a wrapper function in the C library.
2. The wrapper function must make all of the system call arguments available to the system call trap-handling routine (described shortly). These arguments are passed to the wrapper via the stack, but the kernel expects them in specific registers. The wrapper function copies the arguments to these registers.
3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific CPU register (`%eax`).
4. The wrapper function executes a trap machine instruction (**int 0x80**), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location **0x80** of the system's trap vector. More recent x86-32 architectures implement the **`sysenter`** instruction, which provides a faster method of entering kernel mode than the conventional int `0x80` trap instruction. The use of sysenter is supported in the 2.6 kernel and from glibc 2.3.2 onward.
5. In response to the trap to location 0x80, the kernel invokes the routine **`system_call()`** (located in `arch/i386/entry.S`) to handle the trap. This handler:
   (a) Saves register values onto the kernel stack.
   (b) Checks the validity of the system call number.
   (c) Invokes the appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines (the kernel variable `sys_call_table`). If the system call service routine has any arguments, it first checks their validity; for example, it checks that addresses point to valid locations in user memory. Then the service routine performs the required task, which may involve modifying values at addresses specified in the given arguments and transferring data between user memory and kernel memory (e.g., in I/O operations). Finally, the service routine returns a result status to the `system_call()` routine.
   (d) Restores register values from the kernel stack and places the system call return value on the stack.
   (e) Returns to the wrapper function, simultaneously returning the processor to user mode.
6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable **`errno`** using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

On Linux, system call service routines follow a convention of returning a nonnegative value to indicate success. In case of an error, the routine returns a negative number, which is the negated value of one of the errno constants. When a negative value is returned, the C library wrapper function negates it (to make it positive), copies the result into errno, and returns -1 as the function result of the wrapper to indicate an error to the calling program. This convention relies on the assumption that system call service routines don't return negative values on success. However, for a few of these routines, this assumption doesn't hold. Normally, this is not a problem, since the range of negated errno values doesn't overlap with valid negative return values. However, this convention does cause a problem in one case: the `F_GETOWN` operation of the `fcntl()` system call.


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
...
execve(path, argv, envp);
...
execve(path, argv, envp)
{
...
int 0x80
(arguments: __NR_execve, path, argv, envp)
...
return;
}
switch to kernel mode
switch to user mode
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
system_call:
...
call sys_call_table [__NR_execve]
...
sys_execve()
{
...
return error;
}
Figure 3-1: Steps in the execution of a system call

4.11 Classification of interrupts and exceptions

- **interrupts**: asynchronously generated by H/W devices or timers
  - **maskable interrupts**
  - **unmaskable interrupts**
- **exceptions**: synchronously produced by the CPU while executing instructions
  - **faults**: something which can be corrected (e.g. page fault); **eip** is saved and exception handler is executed, and then control is back to saved **eip**
  - **traps**: reported immediately after execution of **trapping instruction**
  - **aborts**: some serious error

- **programmed exceptions (a.k.a. software interrupts)**: ocurr at the request of the programmer (e.g. `int` instruction)

5 File Systems

Unix file sytem provides **data files**, which are *uninterpreted byte arrays* and **directories**, which contain named references to data files and other directories.

5.1 Buffer cache

The buffer cache is a linked list of buf structures holding cached copies of disk block contents. Caching disk blocks in memory reduces the number of disk reads and also provides a synchronization point for disk blocks used by multiple processes.

Interface

- To get a buffer for a particular disk block, call bread.
- After changing buffer data, call bwrite to write it to disk.
- When done with the buffer, call brelse.
- Do not use the buffer after calling brelse.
- Only one process at a time can use a buffer,
- so do not keep them longer than necessary.

The implementation uses three state flags internally:

- `B_BUSY`: the block has been returned from bread and has not been passed back to brelse.
- `B_VALID`: the buffer data has been read from the disk.
- `B_DIRTY`: the buffer data has been modified and needs to be written to disk.

5.2 What happens when `fread` is executed?

- User calls `fread` in user space.

  ```
  size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
  ```

- Implementation of `fread` in LIBC calls read system call

  ```
  ssize_t read(int fd, void *buf, size_t count);
  ```

- Now, we're in kernel space.
- **SYSCALL.C**: `syscall()` is executed

  ```
  void syscall(void)
  {
    int num;

    num = proc->tf->eax;
    if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
      proc->tf->eax = syscalls[num]();
    } else {
      ERROR(unknown_syscall);
      proc->tf->eax = -1;
    }
  }
  ```

- **SYSFILE.C**: `sys_read()` is executed

  ```
  int sys_read(void)
  {
    struct file *f;
    int n;
    char *p;
  ```

```
  if (argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```

- **FILE.C**: `fileread()` is executed

```
int fileread(struct file *f, char *addr, int n)
{
  if (f->type == FD_PIPE) {
    return piperead(f->pipe, addr, n);
  }

  if (f->type == FD_INODE) {
    ilock(f->ip);
    if ((r = readi(fp->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
}
```

- **FS.C**: Read from the file system with `readi()`.

```
int readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV) {
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for (tot=0; tot<n; tot+=m, off+=m, dst+=m) {
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

- **BIO.C**: Read from buffer cache with `bread`

```
struct buf *bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
}
```

- **IDE.C**: Simple PIO-based (non-DMA) **IDE device driver**

```
void iderw(struct buf *b)
{
  struct buf **pp;

  if (!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if ((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if (b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);

  // Append b to idequeue.
  b->qnext = 0;
  for (pp=&idequeue; *pp; pp=&(*pp)->qnext)
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID) {
    sleep(b, &idelock);
  }
  release(&idelock);
}
```

6 Assembly: X86 Assembly Tutorial

For more detailed information about the architecture and about processor instructions, you will need access to a 486 (or 386) microprocessor manual. For more details, see The 80386 book, by Ross P. Nelson. Intel processor manuals may also be found at http://www.x86.org/intel.doc/586manuals.htm.

The GNU Assembler, gas, uses a different syntax from what you will likely find in any x86 reference manual, and the two-operand instructions have the source and destinations in the opposite order. Here are the types of the gas instructions:

```
opcode                    (e.g., pushal)
opcode operand            (e.g., pushl %edx)
opcode source,dest        (e.g., movl %edx,%eax) (e.g., addl %edx,%eax)
```

Where there are two operands, the rightmost one is the destination. The leftmost one is the source. For example, `movl %edx, %eax` means Move the contents of the edx register into the eax register. For another example, `addl %edx,%eax` means Add the contents of the edx and eax registers, and place the sum in the eax register.

Included in the syntactic differences between gas and Intel assemblers is that all register names used as operands must be preceeded by a percent (`%`) sign, and instruction names usually end in either "l", "w", or "b", indicating the size of the operands: long (32 bits), word (16 bits), or byte (8 bits), respectively. For our purposes, we will usually be using the "l" (long) suffix.

6.1 80386+ Register Set

There are different names for the same register depending on what part of the register you want to use. To use the first set of 8 bits of eax (bits 0-7), you would use

Here are the important processor registers:

```
EAX,EBX,ECX,EDX - "general purpose", more or less interchangeable

EBP             - used to access data on stack
                - when this register is used to specify an address, SS is
                  used implicitly

ESI,EDI         - index registers, relative to DS,ES respectively

SS,DS,CS,ES,FS,GS - segment registers
                  - (when Intel went from the 286 to the 386, they figured
                     that providing more segment registers would be more
                     useful to programmers than providing more general-
                     purpose registers... now, they have an essentially
                     RISC processor with only _FOUR_ GPRs!)
                  - these are all only 16 bits in size

EIP            - program counter (instruction pointer), relative to CS

ESP            - stack pointer, relative to SS

EFLAGS         - condition codes, a.k.a. flags
```

6.2 Segmentation

We are using the 32-bit segment addressing feature of the 486. Using 32-bit addressing as opposed to 16-bit addressing gives us many advantages: No need to worry about 64K segments. Segments can be 4 gigabytes in length under the 32-bit architecture. 32-bit segments have a protection mechanism for segments, which you have the option of using. You don't have to deal with any of that ugly 16-bit crud that is used in other operating systems for the PC, like DOS or OS/2; 32-bit segmentation is really a thing of beauty in comparison to that. i486 addresses are formed from a segment base address plus an offset. To compute an absolute memory address, the i486 figures out which segment register is being used, and uses the value in that segment register as an index into the global descriptor table (GDT). The entry in the GDT tells (among other things) what the absolute address of the start of the segment is. The processor takes this base address and adds on the offset to come up with the final absolute address for an operation. You'll be able to look in a 486 manual for more information about this or about the GDT's organization.

i486 has 6 16-bit segment registers, listed here in order of importance:

1. **CS (Code Segment Register):** Added to address during instruction fetch.
2. **SS (Stack Segment Register)**: Added to address during stack access.
3. **DS (Data Segment Register)**: Added to address when accessing a memory operand that is not on the stack.
4. **ES, FS, GS (Extra Segment Registers)**: Can be used as extra segment registers; also used in special instructions that span segments (like string copies).

The x86 architecture supports different addressing modes for the operands. A discussion of all modes is out of the scope of this tutorial, and you may refer to your favorite x86 reference manual for a painfully-detailed discussion of them. Segment registers are special, you can't do a

```
    movw seg-reg, seg-reg
```

You can, however, do

```
    movw seg-reg,memory
    movw memory,seg-reg
    movw seg-reg,reg
    movw reg,seg-reg
```

Note: If you `movw %ss,%ax`, then you should `xorl %eax,%eax` first to clear the high-order 16 bits of eax, so you can work with long values.

6.3 Common/Useful Instructions

```
mov (especially with segment registers)
    - e.g.,:
        movw %es,%ax
        movl %cs:4,%esp
        movw _processControlBlock,%cs

    - note:     mov's do NOT set flags

pushl, popl       - push/pop long
pushal, popal     - push/pop EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI

call  (jumps to piece of code, saves return address on stack)
         e.g., call _cFunction

int   - call a software interrupt

ret   (returns from piece of code entered due to call instruction)
iretl (returns from piece of code entered due to hardware or software interrupt)

sti, cli - set/clear the interrupt bit to enable/disable interrupts respectively
lea  - is Load Effective Address, it's basically a direct pipeline to the
       address you want to do calculations on without affecting any flags,
       or the need of pushing and popping flags.
```

6.4 A simple example

```
CODE
void funtction1() {
int A = 10;
A += 66;
}

compiles to...
funtction1:
```

```
1 pushl %ebp #
2 movl %esp, %ebp #,
3 subl $4, %esp #,
4 movl $10, -4(%ebp) #, A
5 leal -4(%ebp), %eax #,
6 addl $66, (%eax) #, A
7 leave
8 ret
```

Explanation:

1. push ebp
2. copy stack pointer to ebp
3. make space on stack for local data
4. put value 10 in A (this would be the address A has now)
5. load address of A into EAX (similar to a pointer)
6. add 66 to A

... don't think you need to know the rest

6.5 Mixing C and Assembly Language

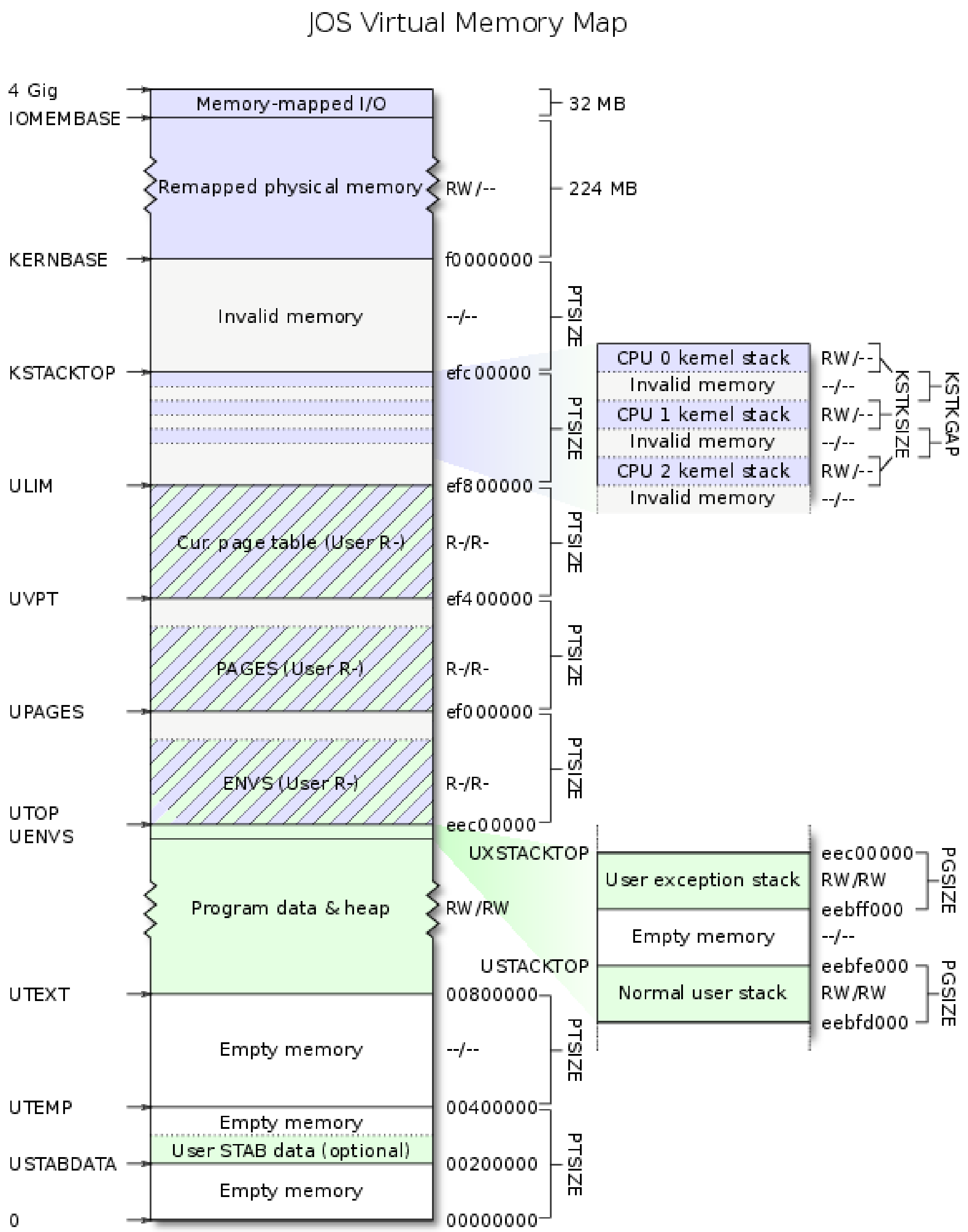
The way to mix C and assembly language is to use the "asm" directive. To access C-language variables from inside of assembly language, you simply use the C identifier name as a memory operand. These variables cannot be local to a procedure, and also cannot be static inside a procedure. They must be global (but can be static global). The newline characters are necessary.

```
unsigned long a1, r;
void junk( void )
{
   asm(
        "pushl %eax \n"
        "pushl %ebx \n"
        "movl $100,%eax \n"
        "movl a1,%ebx \n"
        "int $69 \n"
        "movl %eax,r \n"
        "popl %ebx \n"
        "popl %eax \n"
   );
}
```

This example does the following:

1. Pushes the value stored in eax and ebx onto the stack.
2. Puts a value of 100 into eax.
3. Copies the value in global variable a1 into ebx.
4. Executes a software interrupt number 69.
5. Copies the value in eax into the global variable r.
6. Restores (pops) the contents of the temporary registers eax and ebx.

7 Appendix: JOS Virtual Memory Map



8 Appendix: Linux Device Drivers

8.1 Classes of Devices and Modules

- **char devices**: can be accessed as **stream of bytes**
  - usually drive r implements: open, close, read, write system calls
  - e.g. text console (`/dev/console`) or serial ports (`/dev/ttyS0`)
- **block devices**: block devices are accessed by **filesystem nodes** in the `/dev` directory
  - in most systems, can only handle I/O operations that transfer one or more whole blocks (e.g. 512-byte block)
  - modern OS (e.g. Linux) allows to read any number of bytes from block devices
- **network interfaces**:
  - sends/receives data packets, driven by the network subsystem of the "kernel"
  - transmission is available in **packets**
  - since it's not a stream-oriented device, a network interface cannot be easily mapped to a file in `/dev/` directory
    - instead, Unix assignes a unique name (e.g. `eth0`)
- devices can also be classified according to device protocols: USB, I2C, FireWire, etc.