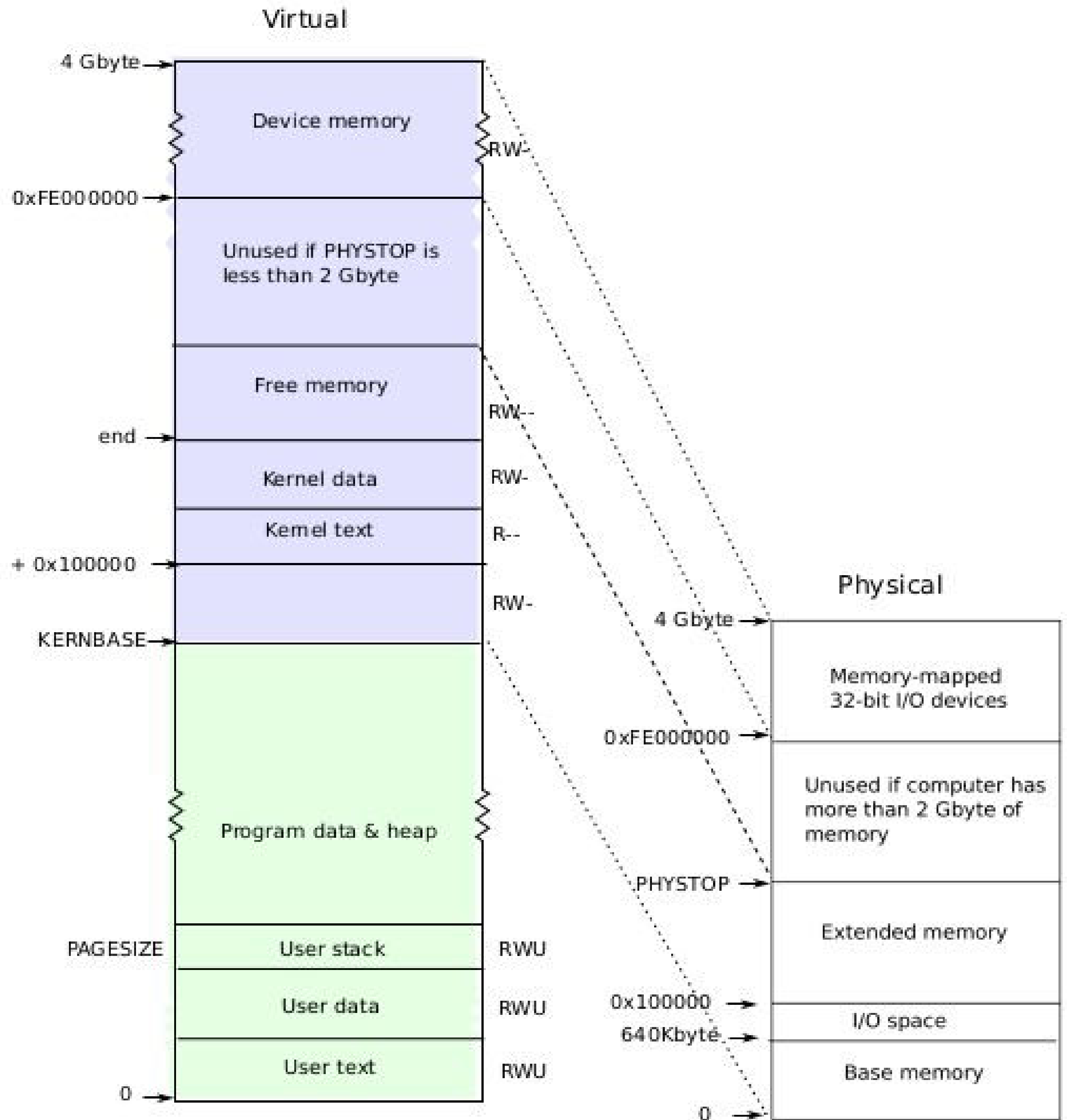
Processes in xv6 code

Process Table

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

- **One single global array of processes**
- **Protected by** `ptable.lock`

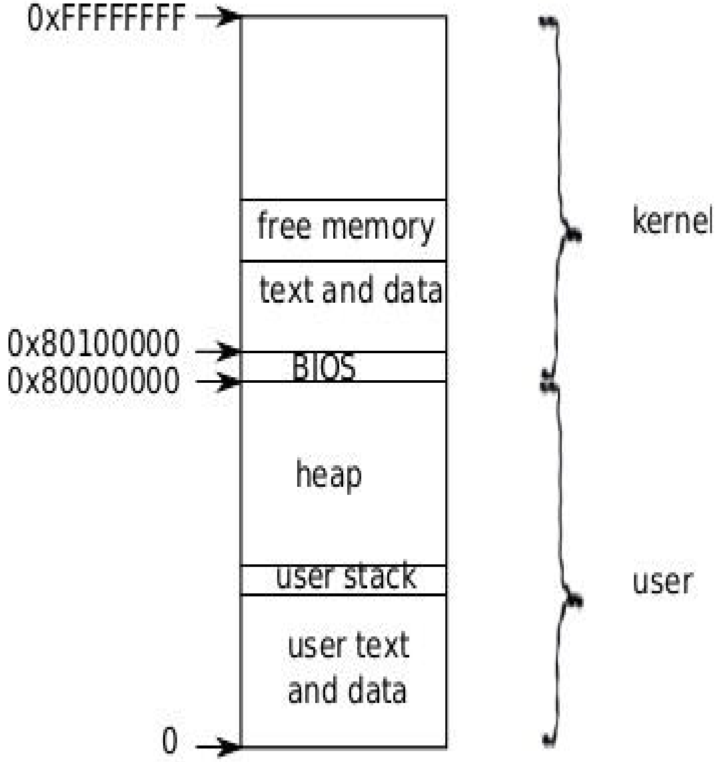


Layout of process's VA space

xv6 schema!

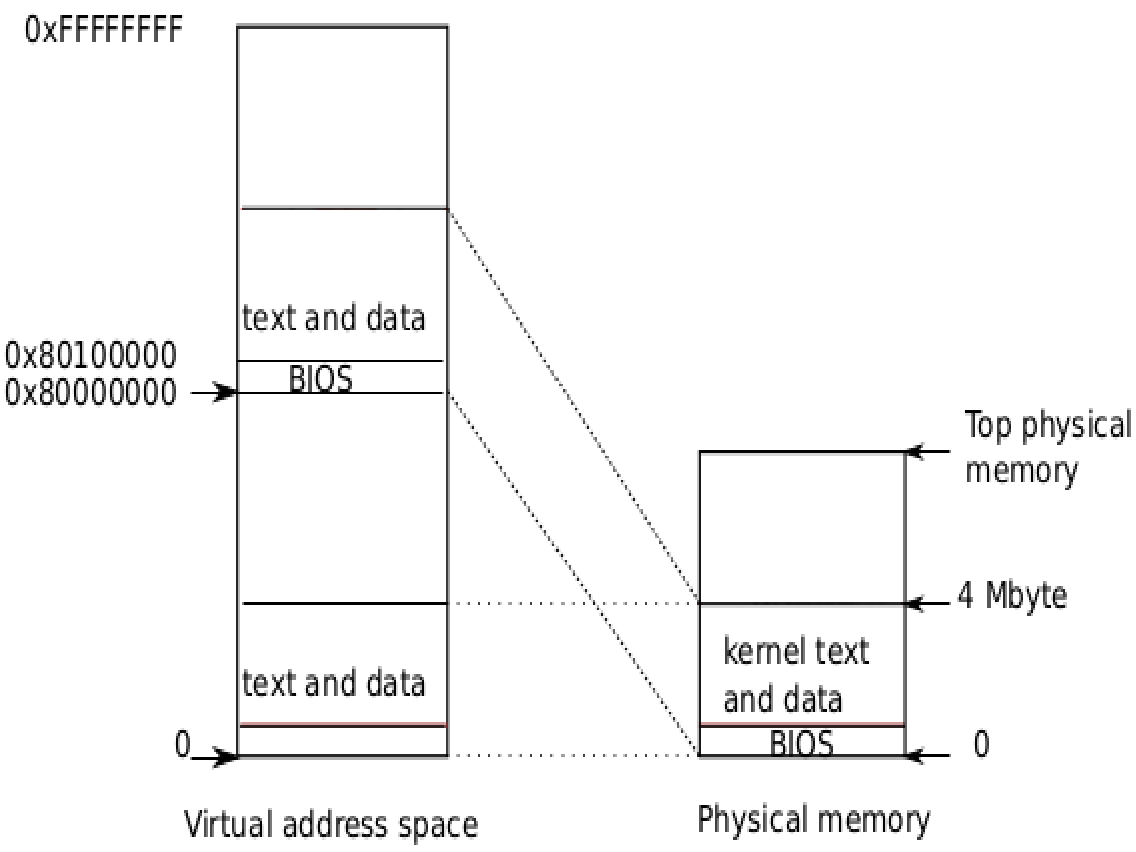
different from Linux

Logical layout of memory for a process



- **Address 0: code**
- **Then globals**
- **Then stack**
- **Then heap**
- **Each processe's address space maps kernel's text, data also --> so that system calls run with these mappings**
- **Kernel code can directly access user memory now**

Kernel mappings in user address space actual location of kernel

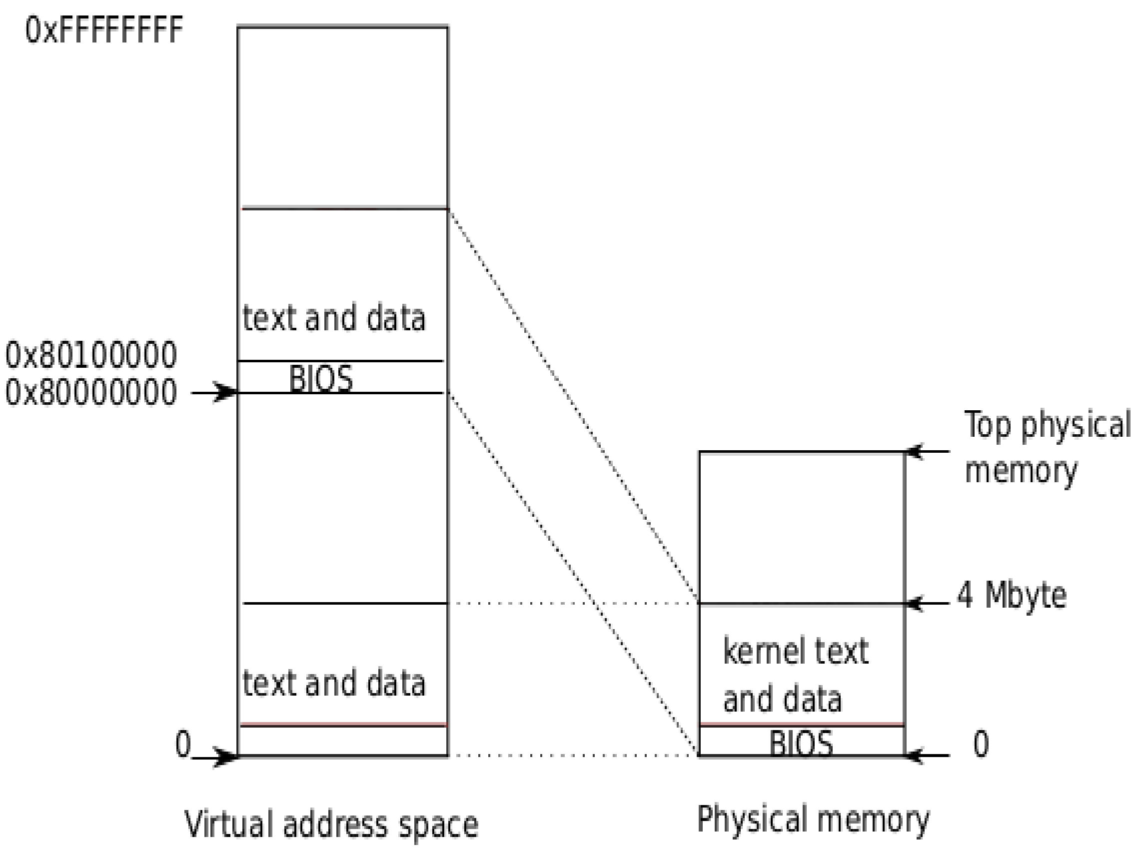


- **Kernel is loaded at 0x100000 physical address**
- **PA 0 to 0x100000 is BIOS and devices**
- **Process's page table will map**

  VA 0x80000000 to PA 0x00000 and

  VA 0x8010000 to 0x100000

Kernel mappings in user address space actual location of kernel



- **Kernel is not loaded at the PA 0x80000000 because some systems may not have that much memory**
- **0x80000000 is called KERNBASE in xv6**

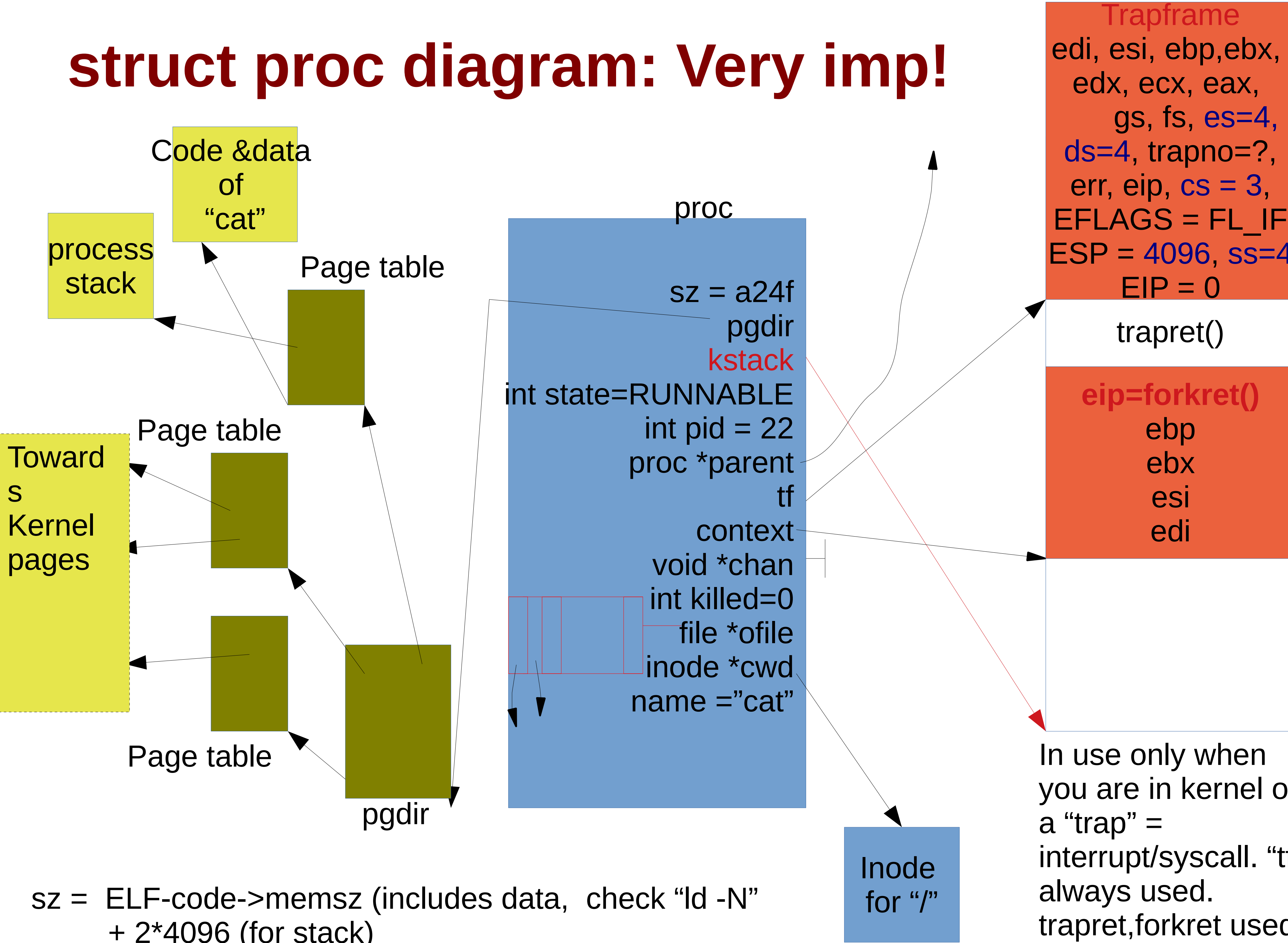
Imp Concepts

- **A process has two stacks**
  - **user stack: used when user code is running**
  - **kernel stack: used when kernel is running on behalf of a process**
- **Note: there is a third stack also!**
  - **The kernel stack used by the scheduler itself**
  - **Not a per process stack**

Struct 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;         // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  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. Process's context
  void *chan;                   // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                   // If non-zero, have been killed
  struct file *ofile[NOFILE];   // Open files, used by open(), read(),...
  struct inode *cwd;            // Current directory, changed with "chdir()"
  char name[16];                // Process name (for debugging)
};
```

struct proc diagram: Very imp!

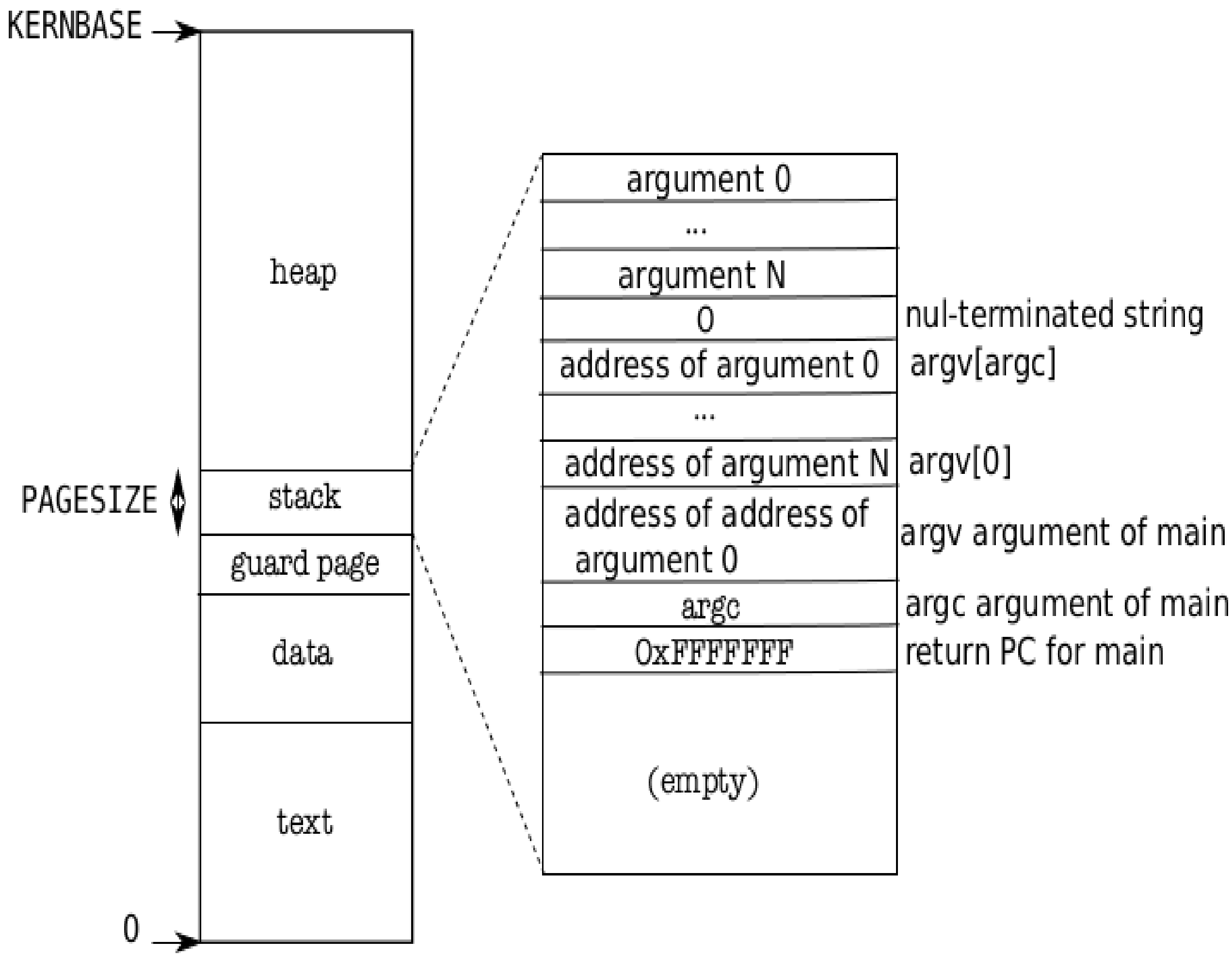


In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

Memory Layout of a user process

Memory Layout of a user process



After exec()

Note the argc, argv on stack

The "guard page" is just a mapping in page table. No frame allocated. It's marked as invalid. So if stack grows (due to many function calls), then OS will detect it with an exception

Handling Traps

Handling traps

- Transition from user mode to kernel mode
  - On a system call
  - On a hardware interrupt
  - User program doing illegal work (exception)
- Actions needed, particularly w.r.t. to hardware interrupts
  - Change to kernel mode & switch to kernel stack
  - Kernel to work with devices, if needed
  - Kernel to understand interface of device

Handling traps

- Actions needed on a trap
  - Save the processor’s registers (context) for future use
  - Set up the system to run kernel code (kernel context) on kernel stack
  - Start kernel in appropriate place (sys call, intr handler, etc)
  - Kernel to get all info related to event (which block I/O done?, which sys call called, which process did exception and what type, get arguments to system call, etc)

Privilege level

- **The x86 has 4 protection levels, numbered 0 (most privilege) to 3 (least privilege).**
- **In practice, most operating systems use only 2 levels: 0 and 3, which are then called kernel mode and user mode, respectively.**
- **The current privilege level with which the x86 executes instructions is stored in %cs register, in the field CPL.**

Privilege level

- Changes automatically on
  - “int” instruction
  - hardware interrupt
  - exeception
- Changes back on
  - iret
- “int” 10 --> makes 10^{th} hardware interrupt. S/w interrupt can be used to create hardware interrupt’
- Xv6 uses “int 64” for actual system calls

Interrupt Descriptor Table (IDT)

- IDT defines intertupt handlers
- Has 256 entries
  - each giving the %cs and %eip to be used when handling the corresponding interrupt.
- Interrupts 0-31 are defined for software exceptions, like divide errors or attempts to access invalid memory addresses.
- Xv6 maps the 32 hardware interrupts to the range 32-63
- and uses interrupt 64 as the system call interrupt

Interrupt Descriptor Table (IDT) entries

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};
```

Setting IDT entries

```
void
tvinit(void)
{
  int i;
  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
    SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                 vectors[T_SYSCALL], DPL_USER);
  /* value 1 in second argument --> don't disable
interrupts
           * DPL_USER means that processes can raise this
interrupt. */
    initlock(&tickslock, "time");
}
```

Setting IDT entries

```
#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;                  \
}
```

Setting IDT entries

Vectors.S

```
# generated by vectors.pl - do
not edit
# handlers
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
 jmp alltraps
```

trapasm.S

```
#include "mmu.h"
# vectors.S sends all traps
here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  Pushal
  ....
```

How will interrupts be handled?

On int instruction/interrupt the CPU does this:

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()

- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- **IDT was already set**
  - **Remember vectors.S**
- **So jump to 64th entry in vector’s**

  vector64:

  pushl $0

  pushl $64

  jmp alltraps

  - **So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64**
  - **Next run alltraps from trapasm.S**

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- Now stack contains
- ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi
  - This is the struct trapframe !
  - So the kernel stack now contains the trapframe
  - Trapframe is a part of kernel stcak

trap()

```
void
trap(struct trapframe *tf)
{
 if(tf->trapno == T_SYSCALL){
  if(myproc()->killed)
   exit();
  myproc()->tf = tf;
  syscall();
  if(myproc()->killed)
   exit();
  return;
 }
 switch(tf->trapno){
 .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler

- Timer
  - wakeup(&ticks)
- IDE: disk interrupt
  - Ideintr()
- KBD
  - Kbdintr()
- COM1
  - Uatrintr()
- If Timer
  - Call yield() -- calls sched()
- If process was killed (how is that done?
  - Call exit()!

when trap() returns

- #Back in alltraps

```
  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
```

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

Scheduler

Scheduler – in most simple terms

- **Selects a process to execute and passes control to it !**
  - **The process is chosen out of "READY" state processes**
  - **Saving of context of "earlier" process and loading of context of "next" process needs to happen**
- **Questions**
  - **What are the different scenarios in which a scheduler called ?**
  - **What are the intricacies of "passing control"**
  - **What is "context" ?**

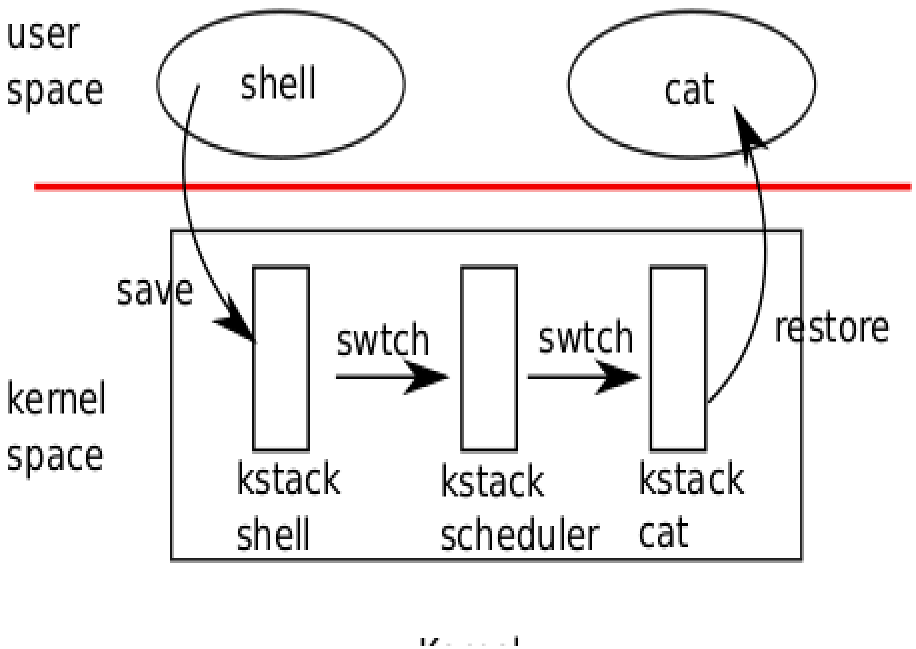
Steps in scheduling scheduling

- Suppose you want to switch from P1 to P2 on a timer interrupt
- P1 was doing

  F() { i++; j++;}
- P2 was doing

  G() { x--; y++; }
- P1 will experience a timer interrupt, switch to kernel (scheduler) and scheduler will schedule P2

4 stacks need to change!



- **User stack of process -> kernel stack of process**
  - **Switch to kernel stack**
  - **The normal sequence on any interrupt !**
- **Kernel stack of process -> kernel stack of scheduler**
  - **Why?**
- **Kernel stack of scheduler -> kernel stack of new process . Why?**
- **Kernel stack of new process -> user stack of new process**

scheduler()

- Disable interrupts
- Find a RUNNABLE process. Simple round-robin!
- c->proc = p
- switchuvm(p) : Save TSS of scheduler’s stack and make CR3 to point to new process pagedir
- p->state = RUNNING
- swtch(&(c->scheduler), p->context)

swtch

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

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

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

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

scheduler()

- swtch(&(c->scheduler), p->context)
- Note that when scheduler() was called, when P1 was running
- After call to swtch() shown above
  - The call does NOT return!
  - The new process P2 given by ‘p’ starts running !
  - Let’s review swtch() again

swtch(old, new)

- The magic function in swtch.S
- Saves callee-save registers of old context
- Switches esp to new-context's stack
- Pop callee-save registers from new context

  ret
  - where? in the case of first process – returns to forkret() because stack was setup like that !
  - in case of other processes, return where?
    - Return address given on kernel stack. But what's that?
    - The EIP in p->context
    - When was EIP set in p->context ?

scheduler()

- **Called from?**
  - **mpmain()**
  - **No where else!**
- **sched() is another scheduler function !**
  - **Who calls sched() ?**
    - **exit() - a process exiting calls sched ()**
    - **yield() - a process interrupted by timer calls yield()**
    - **sleep() - a process going to wait calls sleep()**

sched()

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();
  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
/*A*/  mycpu()->intena = intena;
}
```

- get current process
- Error checking code (ignore as of now)
- get interrupt enabled status on current CPU (ignore as of now)
- call to swtch
  - Note the arguments' order
  - p->context first, mycpu()->scheduler second
- swtch() is a function call
  - pushes address of */*A*/* on stack of current process p
  - switches stack to mycpu()->scheduler. Then pops EIP from that stack and jumps there.
  - when was mycpu()->scheduler set? Ans: during scheduler()!

sched() and schduler()

```
sched() {
...
  swtch(&p->context, mycpu()->scheduler); /* X */

}
```

```
scheduler(void) {
...
      swtch(&(c->scheduler), p->context);  / * Y */
}
```

- scheduler() saves context in c->scheduler, sched() saves context in p->context
- after swtch() call in sched(), the control jumps to Y in scheduler
  - Switch from process stack to scheduler's stack
- after swtch() call in scheduler(), the control jumps to X in sched()
  - Switch from scheduler's stack to new process's stack
- Set of co-operating functions

sched() and scheduler() as co-routines

- In sched()

  ```
  swtch(&p->context, mycpu()->scheduler);
  ```

- In scheduler()

  ```
  swtch(&(c->scheduler), p->context);
  ```

- These two keep switching between processes
- These two functions work together to achieve scheduling
- Using asynchronous jumps
- Hence they are co-routines

To summarize

- On a timer interrupt during P1
  - trap() is called. Stack has changed from P1's user stack to P1's kernel stack
  - trap()->yield()
  - yield()->sched()
  - sched() -> swtch(&p->context, c->scheduler()
  - Stack changes to scheduler's kernel stack.
  - Switches to location "Y" in scheduler().
-

- Now the loop in scheduler()
  - calls switchkvm()
  - Then continues to find next process (P2) to run
  - Then calls switchuvm(p): changing the page table to the P2's page tables
  - then calls swtch(&c->scheduler, p2's->context)
  - Stack changes to P2's kernel stack.
  - P2 runs the last instruction it was was in ! Where was it?
    - mycpu()->intena = intena; in sched()
    - Then returns to the one who called sched() i.e. exit/sleep, etc
    - Finally returns from it's own "TRAP" handler and returns to P2's user stack and user code

Creation of first process by kernel

Why first process needs ‘special’ treatment?

- Normally process is created using fork()
  - and typically followed by a call to exec()
- Fork will use the PCB of existing process to create a new process
  - as a clone
- The first process has nothing to copy from!
- So it’s PCB needs to “built” by kernel code

Why first process needs ‘special’ treatment?

- **XV6 approach**
  - **Create the process as if it was created by “fork”**
  - **Ensure that the process starts in a call to “exec”**
  - **Let “Exec” do the rest of the JOB as expected**
  - **In this case exec() will call**
    - **exec(“/init”, NULL);**
- **See the code of init.c**
  - **opens console() device for I/O; dups 0 on 1 and 2!**
    - **Same device file for I/O**
  - **forks a process and execs (“sh”) on it.**
  - **Itself keeps waiting for zombie processes**

Why first process needs ‘special’ treatment?

- **What needs to be done ?**
  - **Build struct proc by hand**
  - **How data structures (proc, stack, etc) are hand-crafted so that when kernel returns, the process starts in code of init**

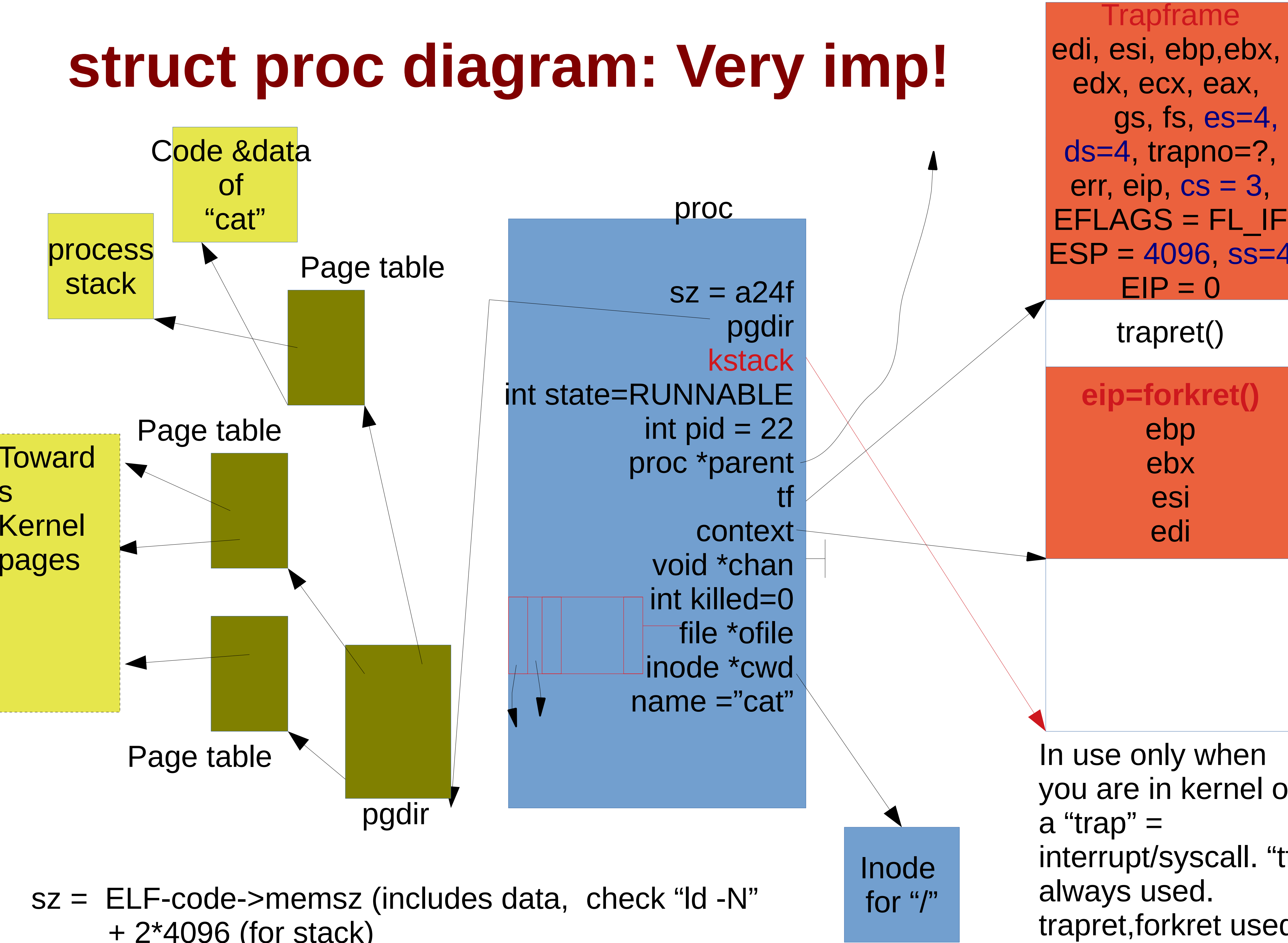
Imp Concepts

- A process has two stacks
  - user stack: used when user code is running
  - kernel stack: used when kernel is running on behalf of a process
- Note: there is a third stack also!
  - The kernel stack used by the scheduler itself
  - Not a per process stack

Imp Concepts

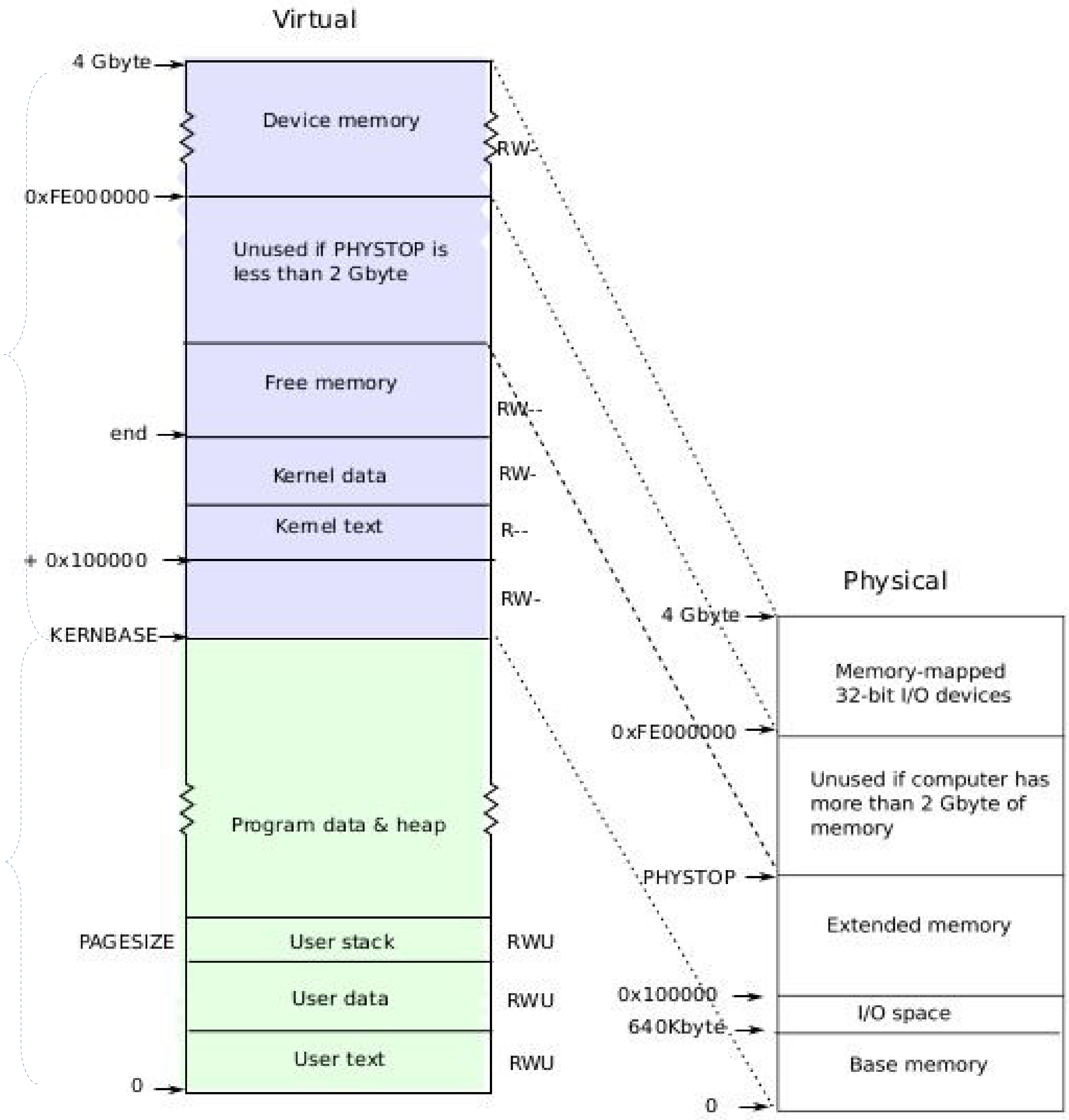
```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

struct proc diagram: Very imp!

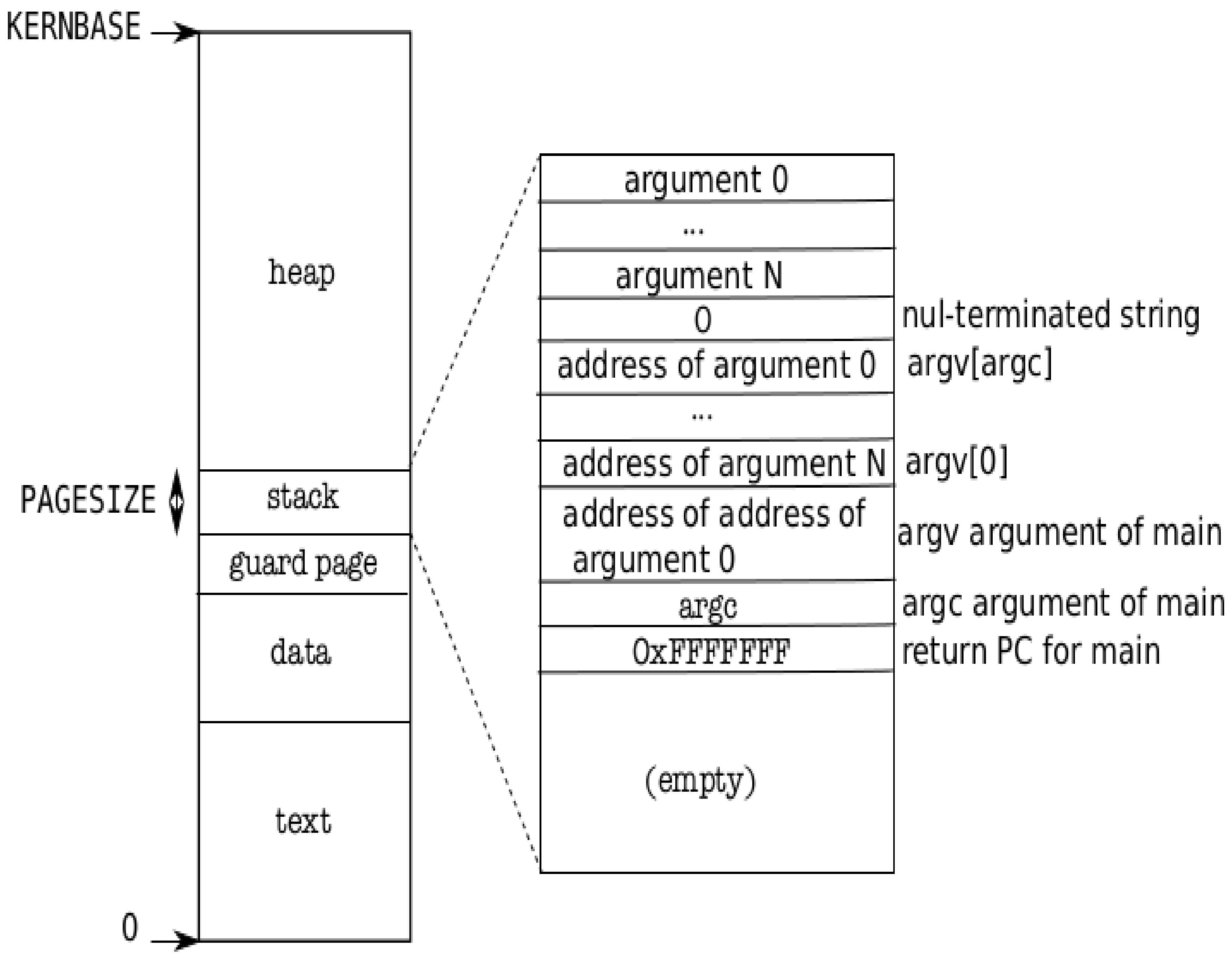


setupkvm()
does this mapping

These mappings
need to be
created per
process



Memory Layout of a user process After exec()



Note the argc, argv on stack

stack is just one page.

size of text and data is derived from ELF file

main()->userinit()
Creating first process by hand

- Code of the first process
  - initcode.S and init.c
  - init.c is compiled into “/init” file
    - During make !
  - Trick:
    - Use initcode.S to “exec(“/init”)”
    - And let exec() do rest of the job
  - But before you do exec()
    - Process must exist as if it was forked() and running

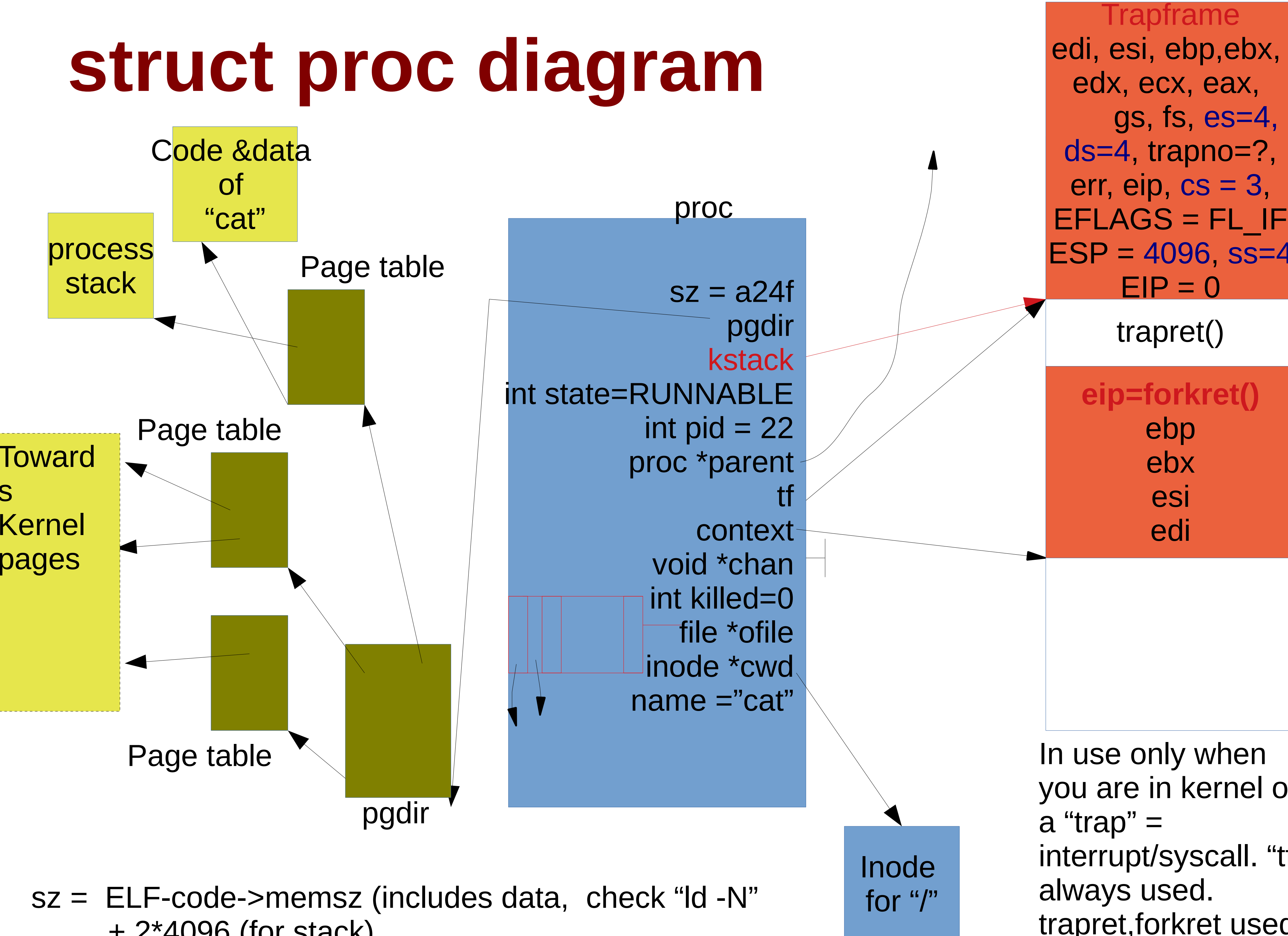
main()->userinit()
Creating first process by hand

```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  // Abhijit: obtain proc 'p', with stack initialized
  // and trapframe created and eip set to 'forkret'
  p = allocproc();
// let’s see what allocproc() does
```

First process creation
Let's revisit struct 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;       // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  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. Process's context
  void *chan;                 // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files, used by open(), read(),...
  struct inode *cwd;          // Current directory, changed with "chdir()"
  char name[16];              // Process name (for debugging)
};
```

struct proc diagram



In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

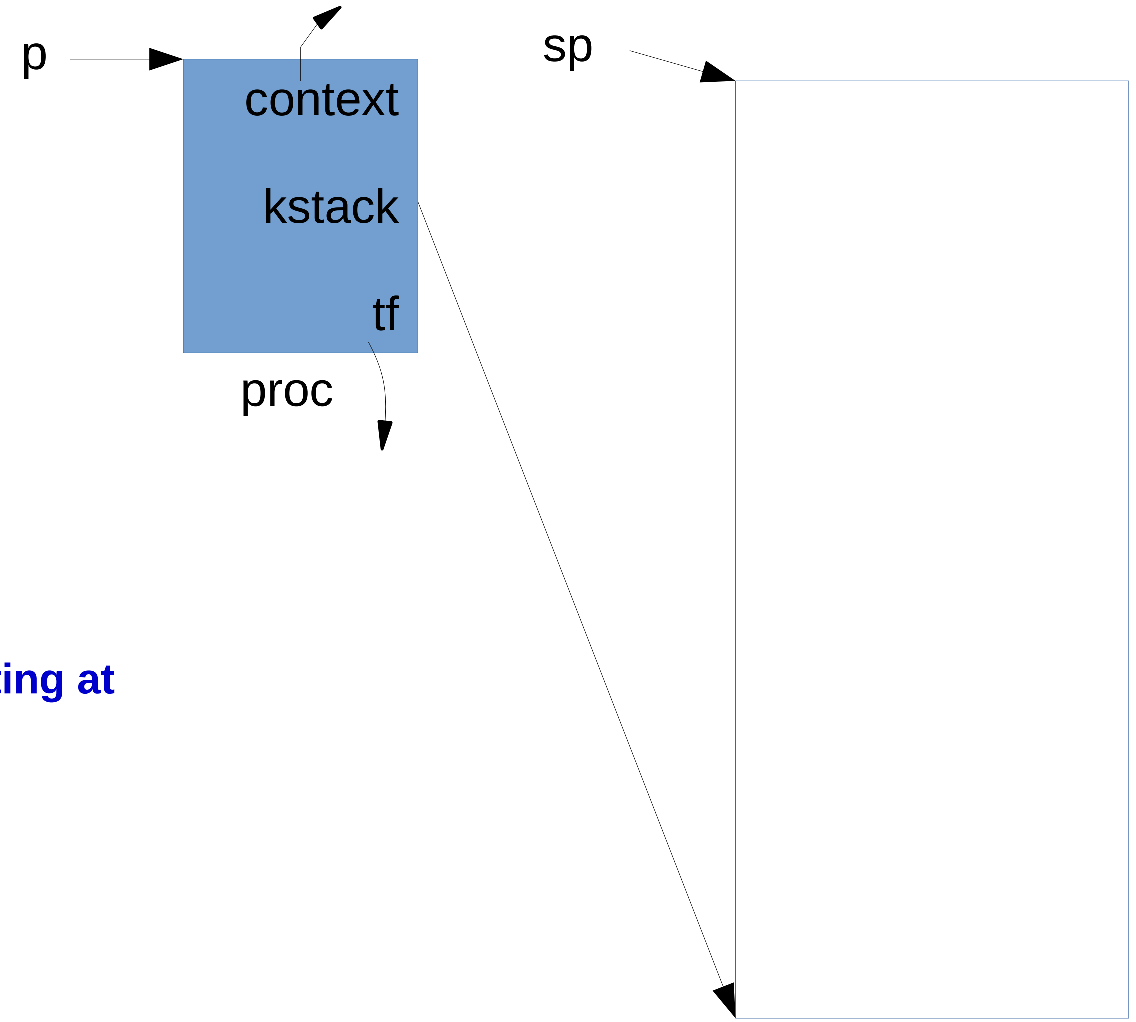
allocproc()

```
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);
```

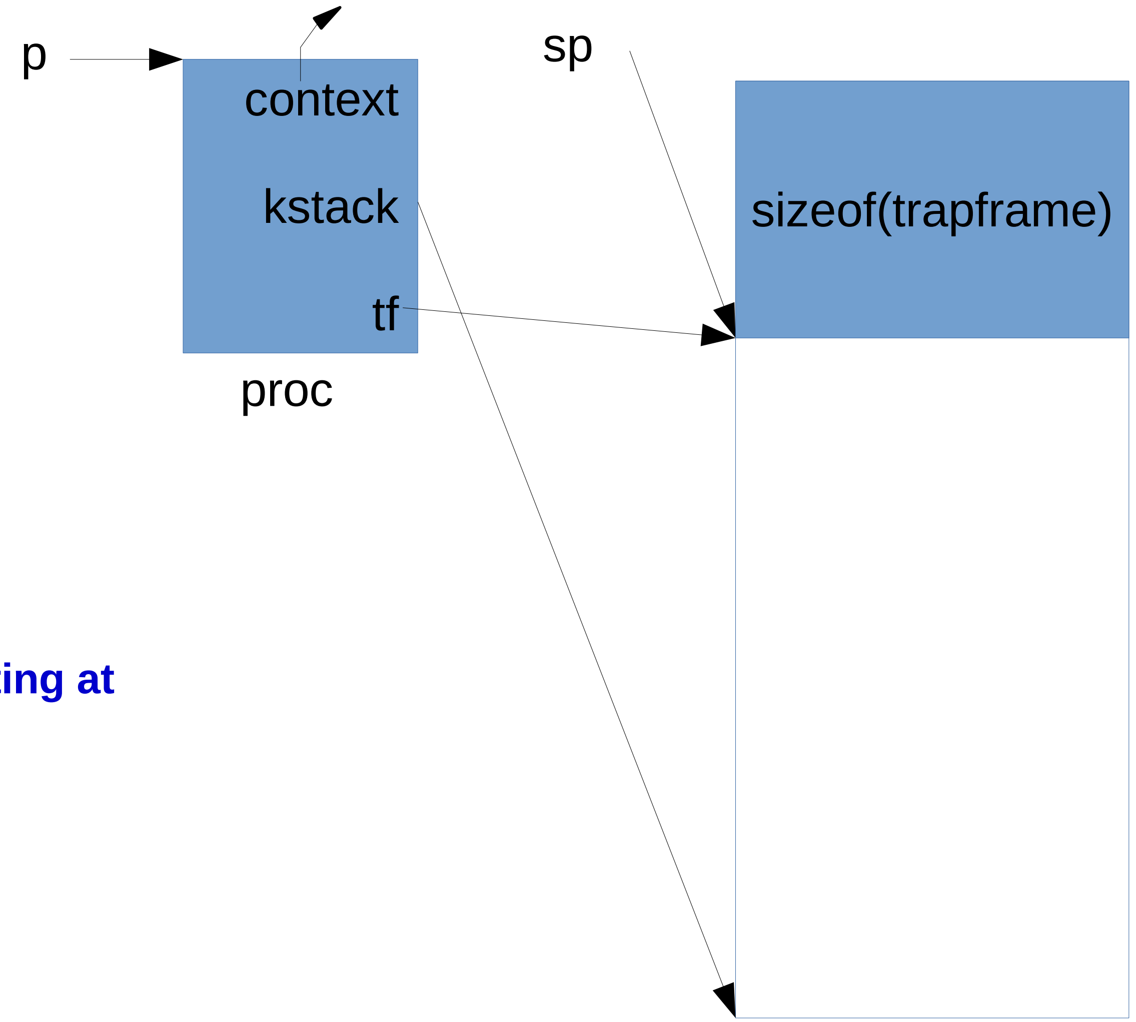
allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // 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;
```



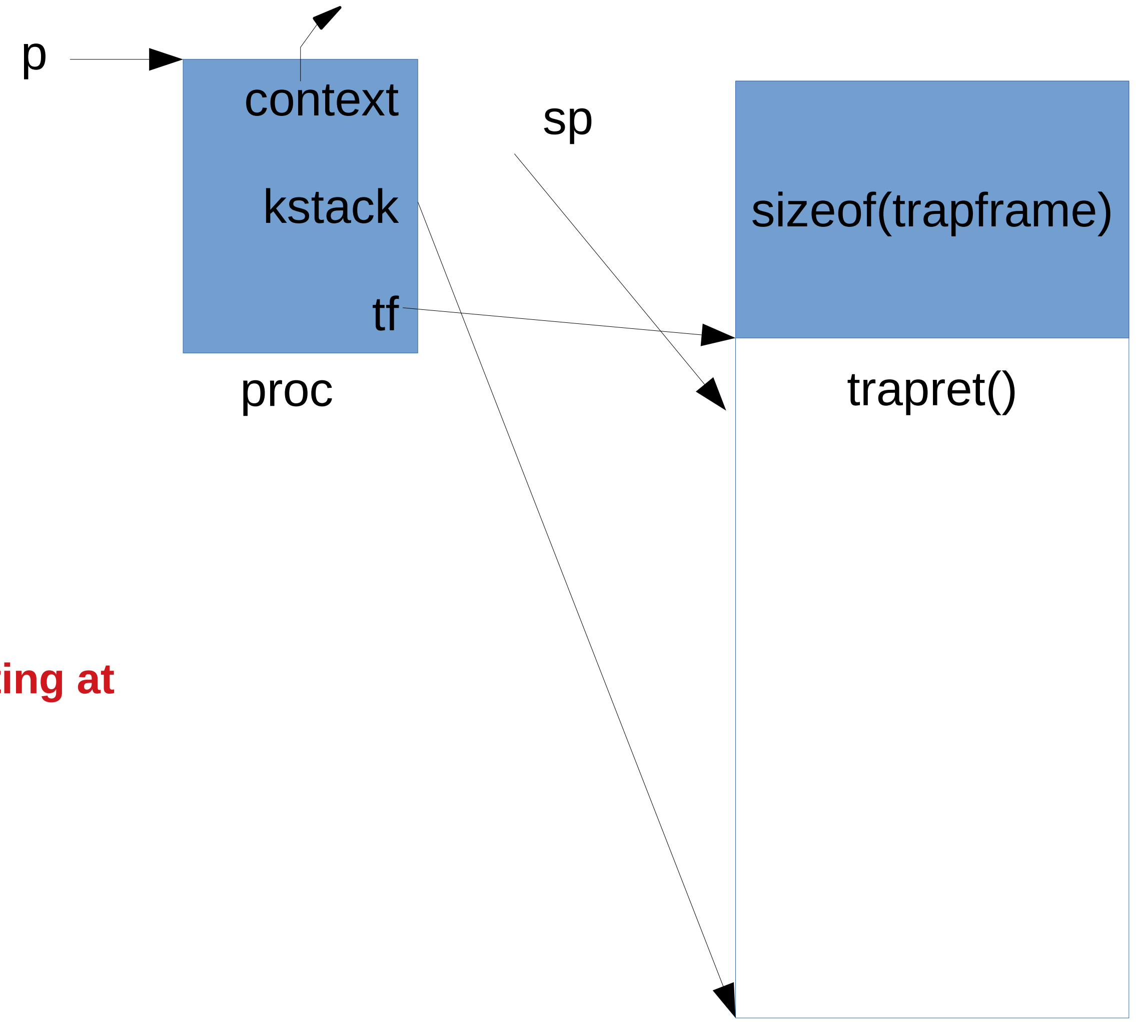
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
// 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;
```



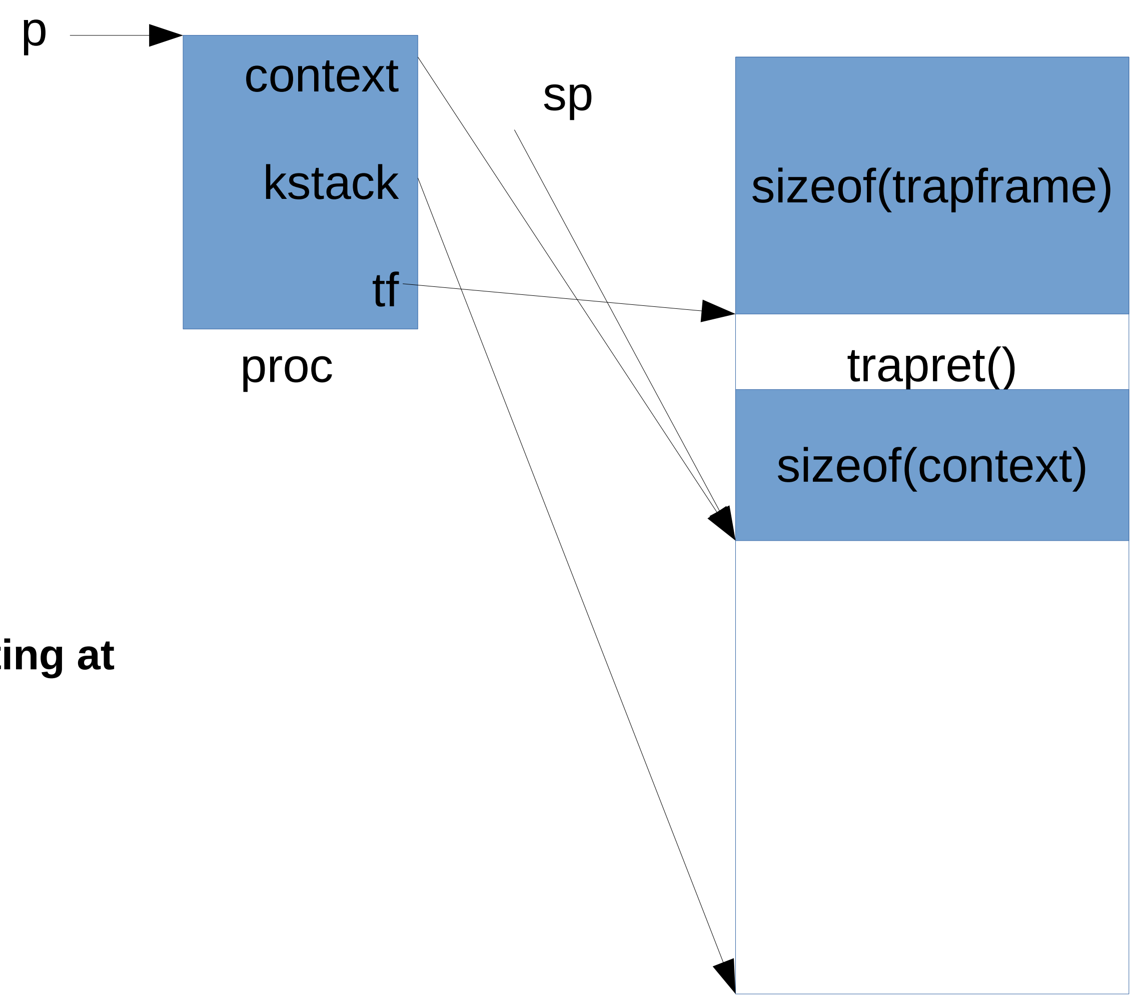
allocproc() setting up stack

```
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
// 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;
```



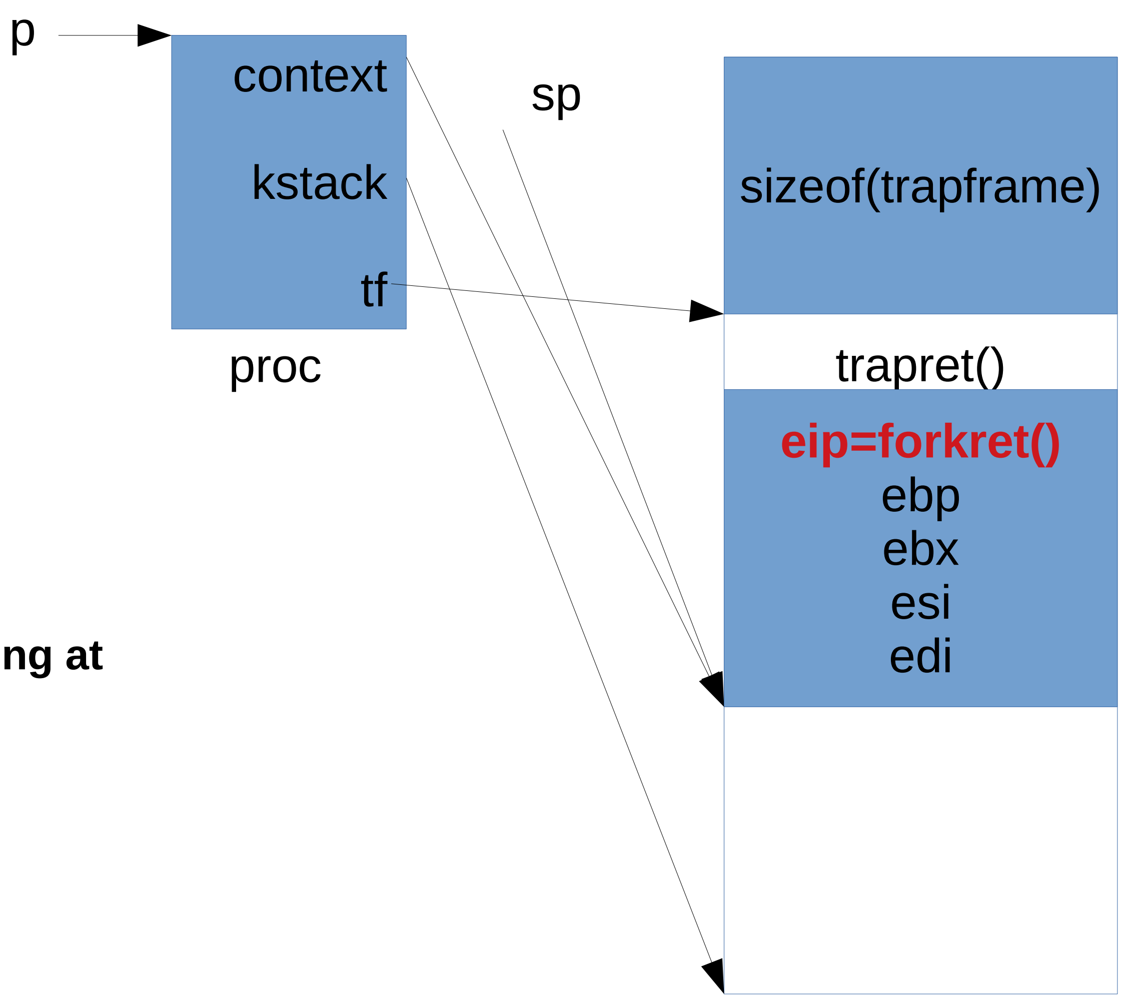
allocproc() setting up stack

```
  if((p->kstack = kalloc()) == 0){
    p->state = UNUSED;
    return 0;
  }
  sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
  // 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;
```



allocproc() setting up stack

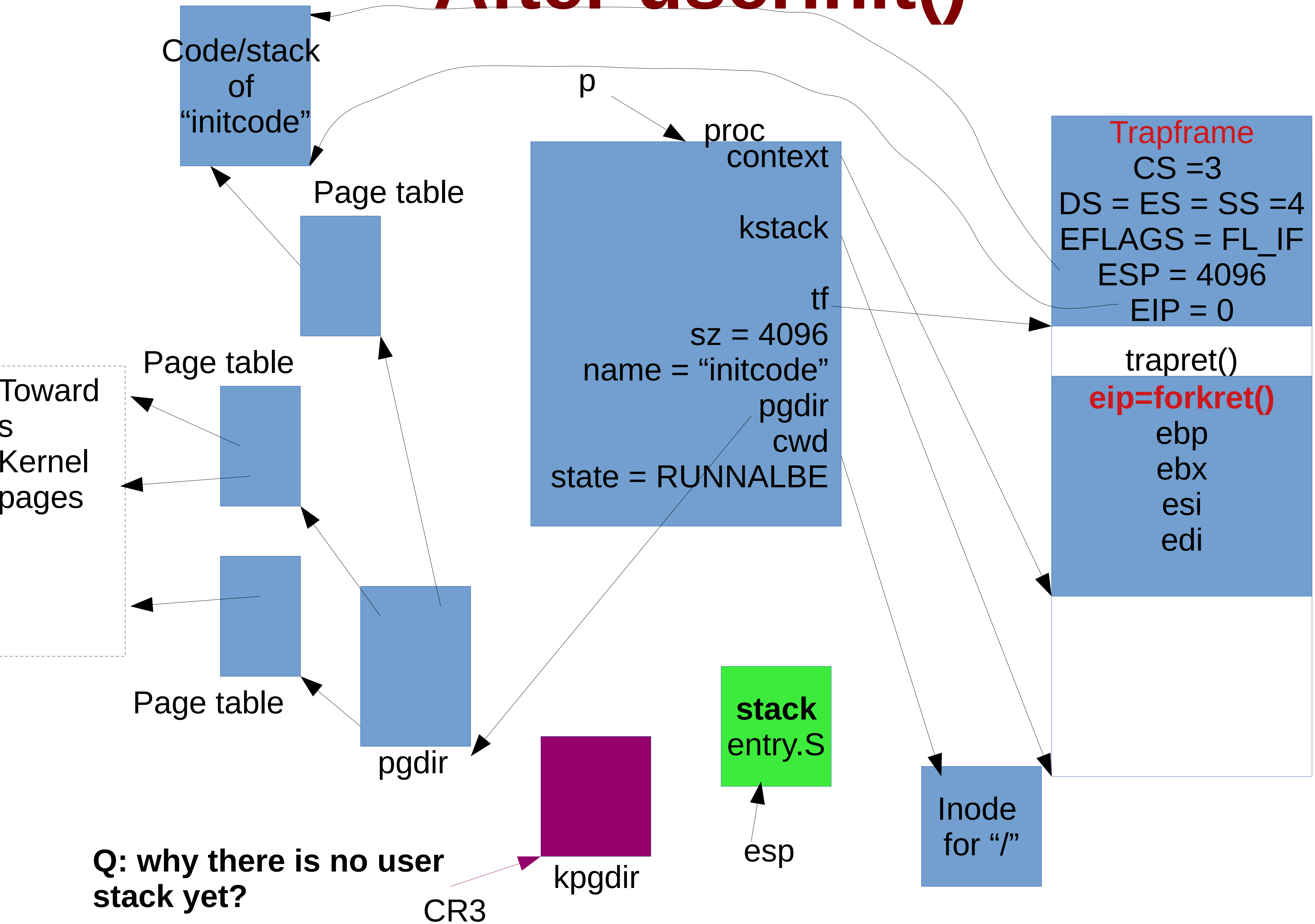
```
if((p->kstack = kalloc()) == 0){
  p->state = UNUSED;
  return 0;
}
sp = p->kstack + KSTACKSIZE;
// Abhijit KSTCKSIZE = PGSIZE
// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;
// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 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;
```



Next in userinit()

```
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S
  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");
  acquire(&ptable.lock);
  p->state = RUNNABLE;

  release(&ptable.lock);
```

After userinit()



Q: why there is no user stack yet?

main()->mpmain()

```
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```

- Load IDT register
  - Copy from idt[] array into IDTR
- Call scheduler()
  - One process has already been made runnable
  - Let's enter scheduler now

Before reading scheduler(): Note

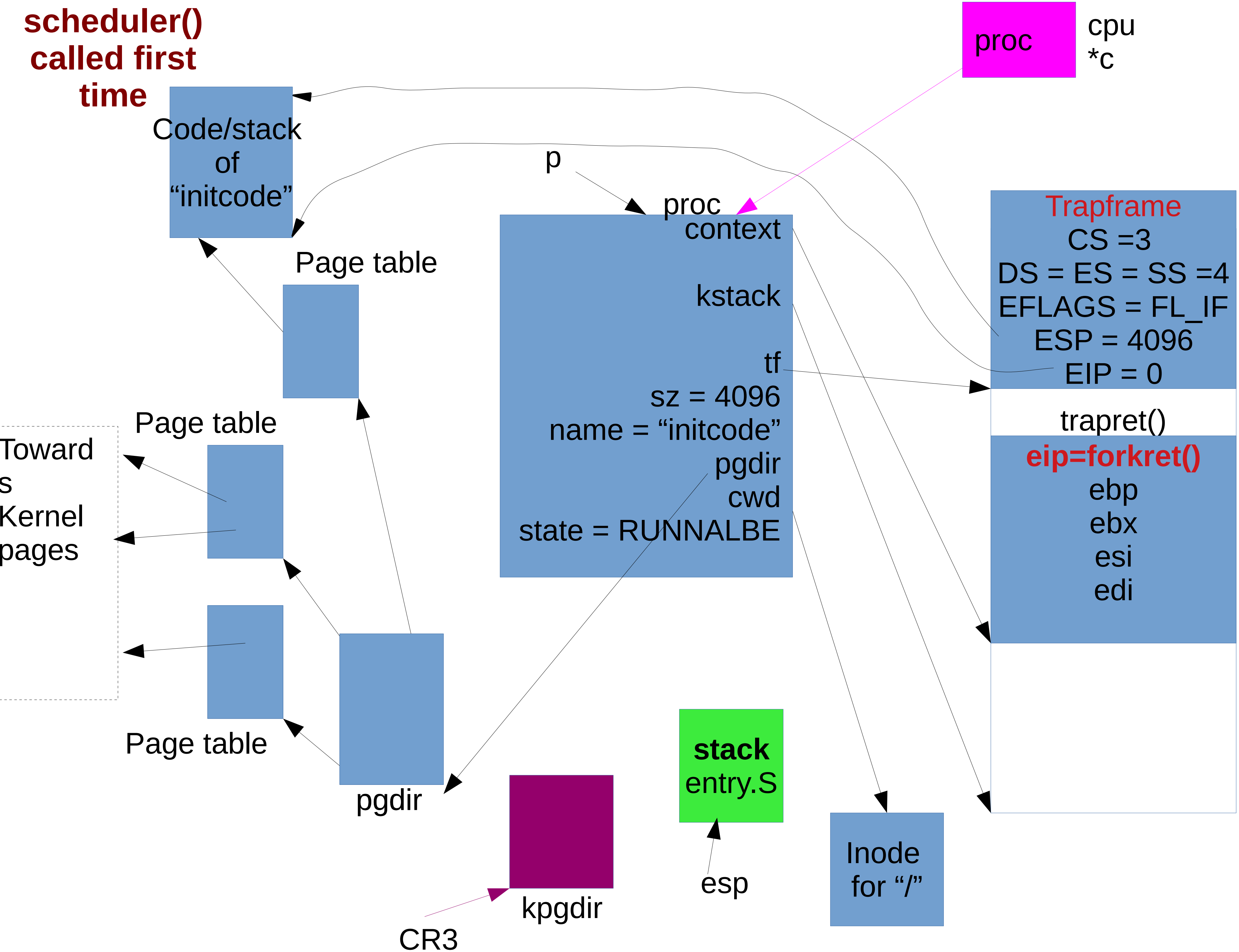
- The esp is still pointing to the stack which was allocated in entry.S !
  - this is the kernel only stack
  - Not the per process kernel stack.
- CR3 points to kpgdir
- Struct cpu[ ] has been setup up already
  - apicid – in mpinit()
  - segdesc gdt – in seginit()
  - started – in mpmain()

- Fields in cpu[] not yet set
  - context * scheduler --> will be setup in sched()
  - taskstate ts --> large structure, only parts used in switchuvm()
  - ncli, intena --> used while locking
  - proc *proc -> set during scheduler()

scheduler()

```
void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
     sti();
    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
     // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
```

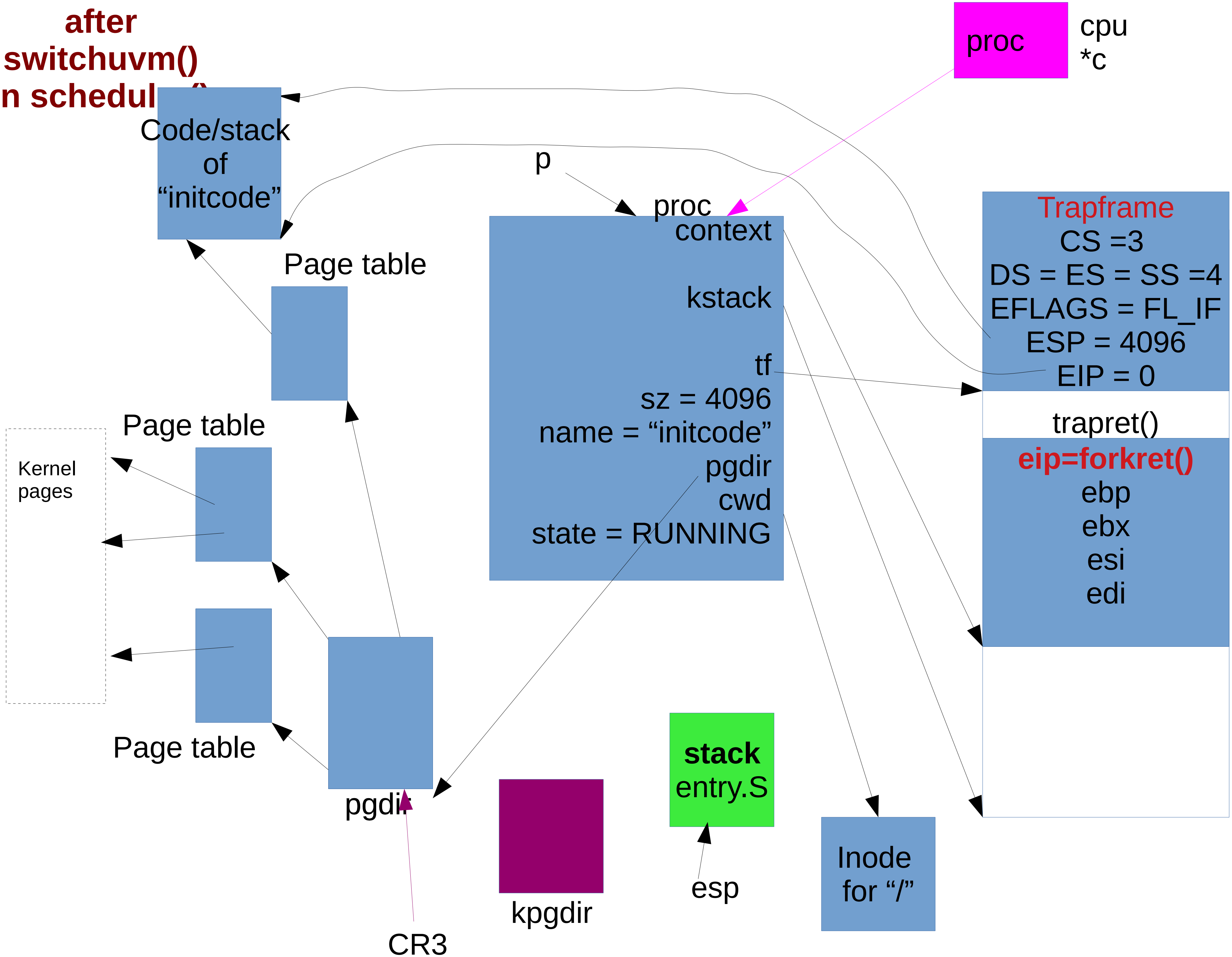
scheduler() called first time



scheduler()

```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;
```

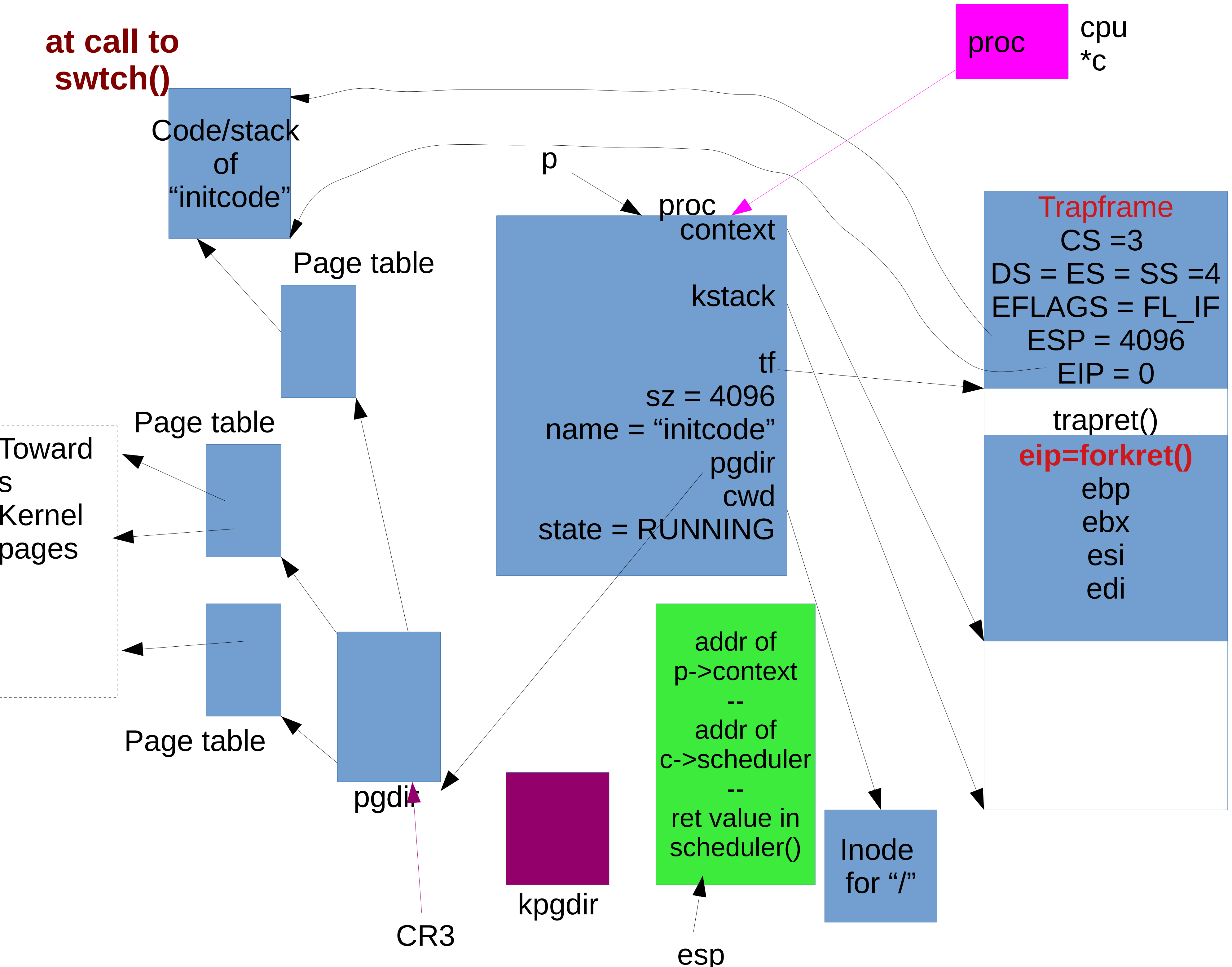
after switchuvm() in schedul()



scheduler()

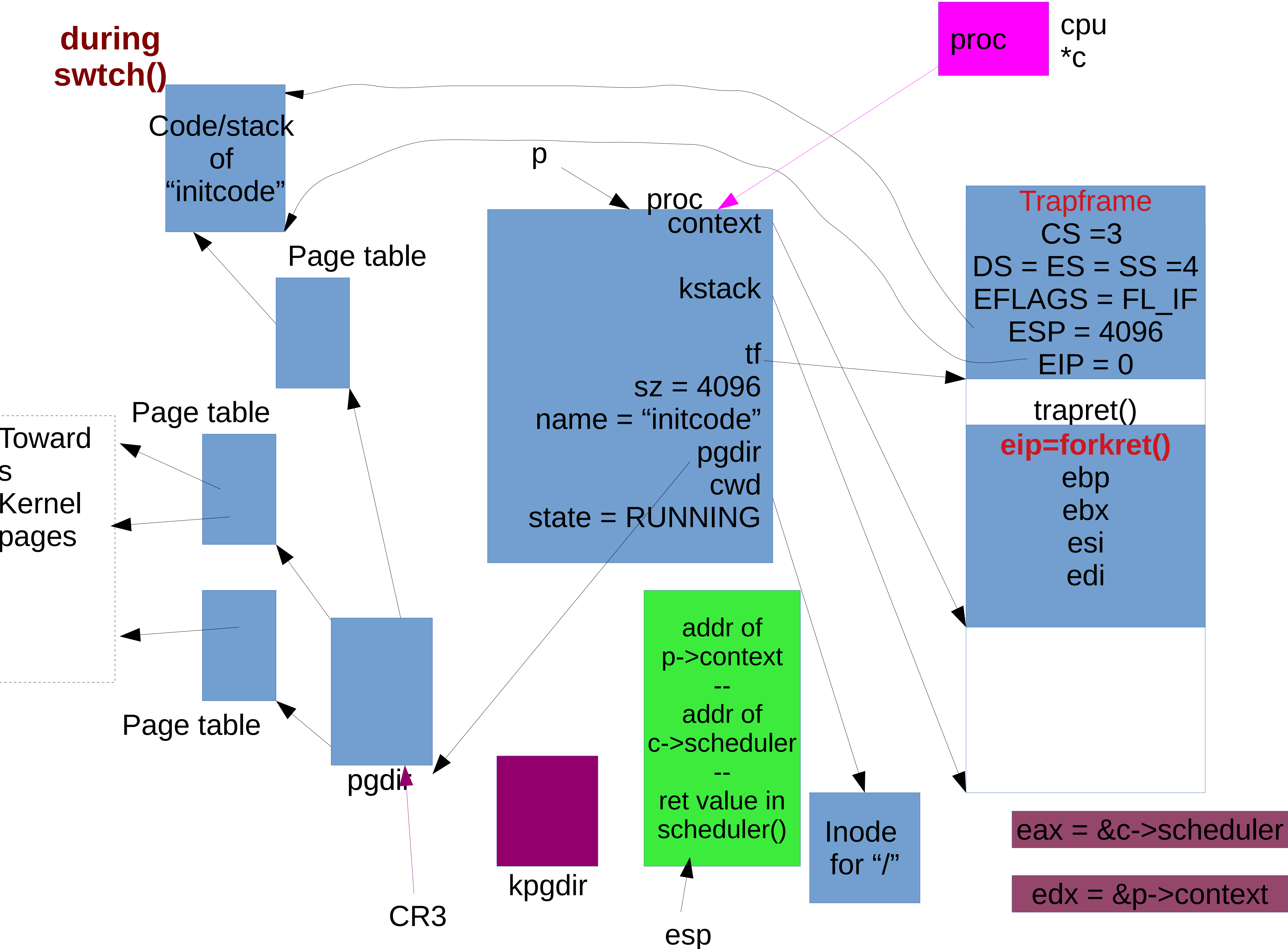
```
acquire(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->state != RUNNABLE)
    continue;

  // Switch to chosen process.  It is the process's job
  // to release ptable.lock and then reacquire it
  // before jumping back to us.
  c->proc = p;
  switchuvm(p);
  p->state = RUNNING
  swtch(&(c->scheduler), p->context);
;
```

at call to swtch()



swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
```

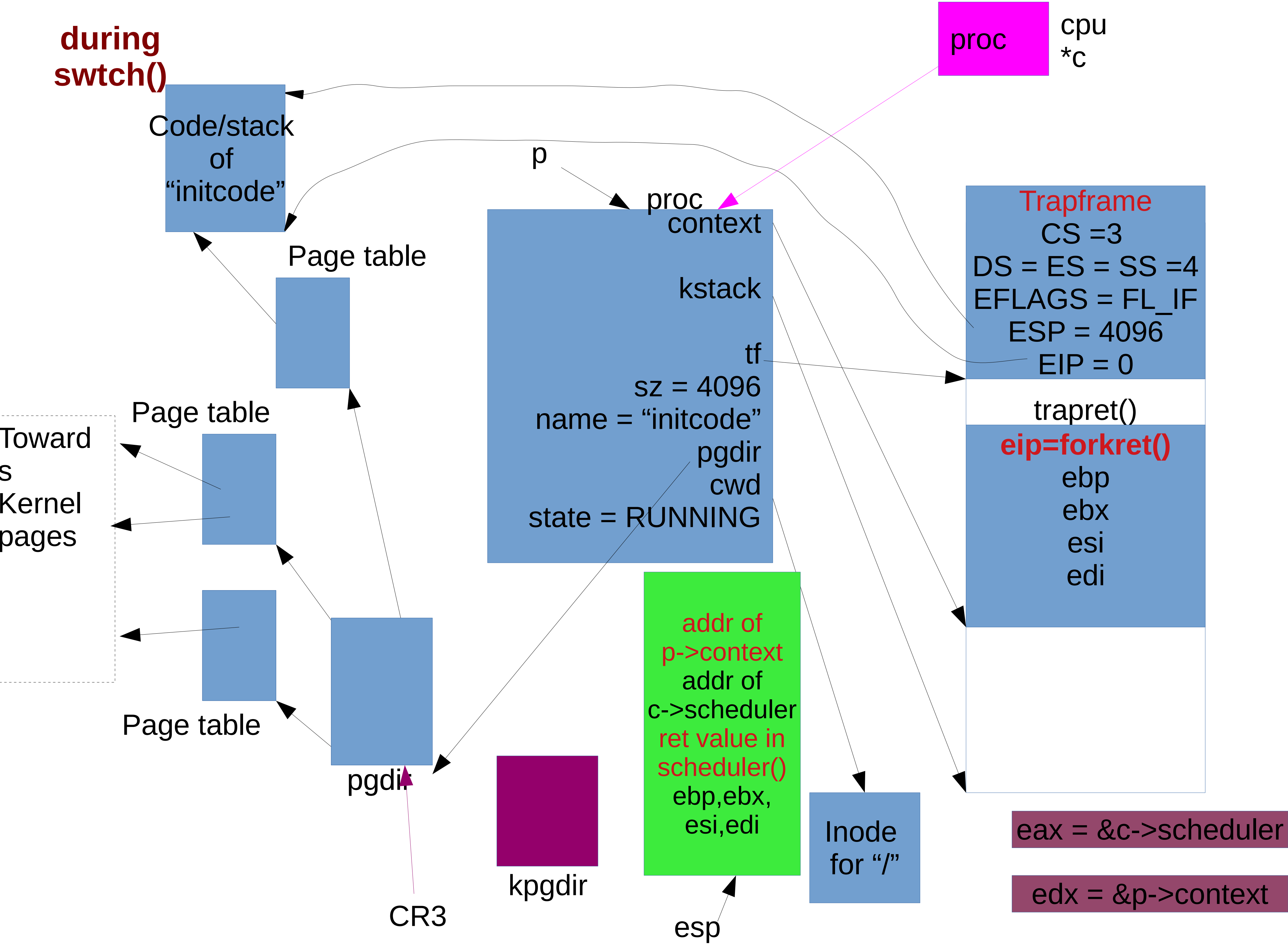


during swtch()
proc
cpu *c
Code/stack of “initcode”
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Towards Kernel pages
Page table
pgdir
CR3
kpgdir
addr of p->context
--
addr of c->scheduler
--
ret value in scheduler()
esp
Inode for “/”
eax = &c->scheduler
edx = &p->context

swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
```

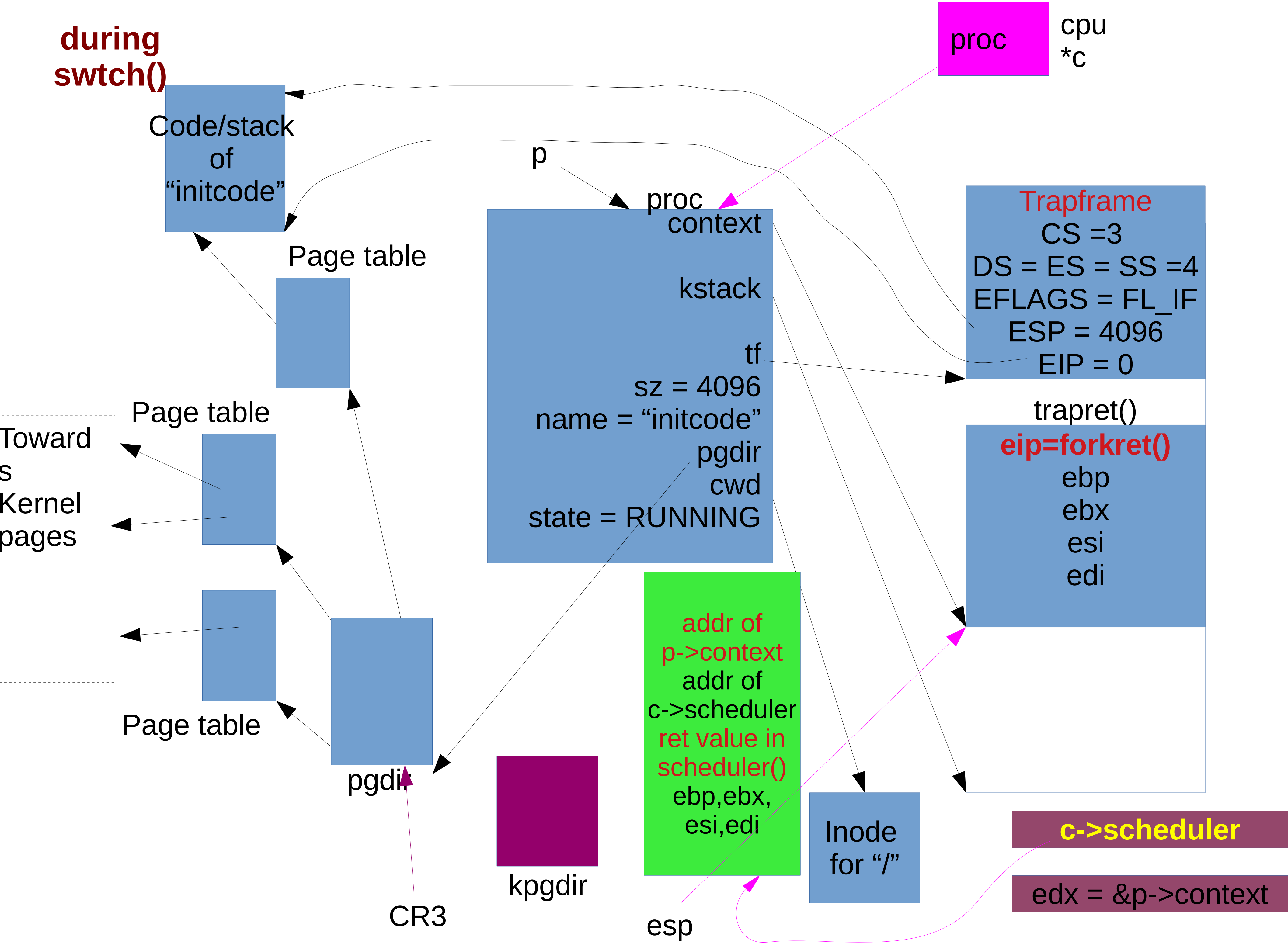
during swtch()



swtch

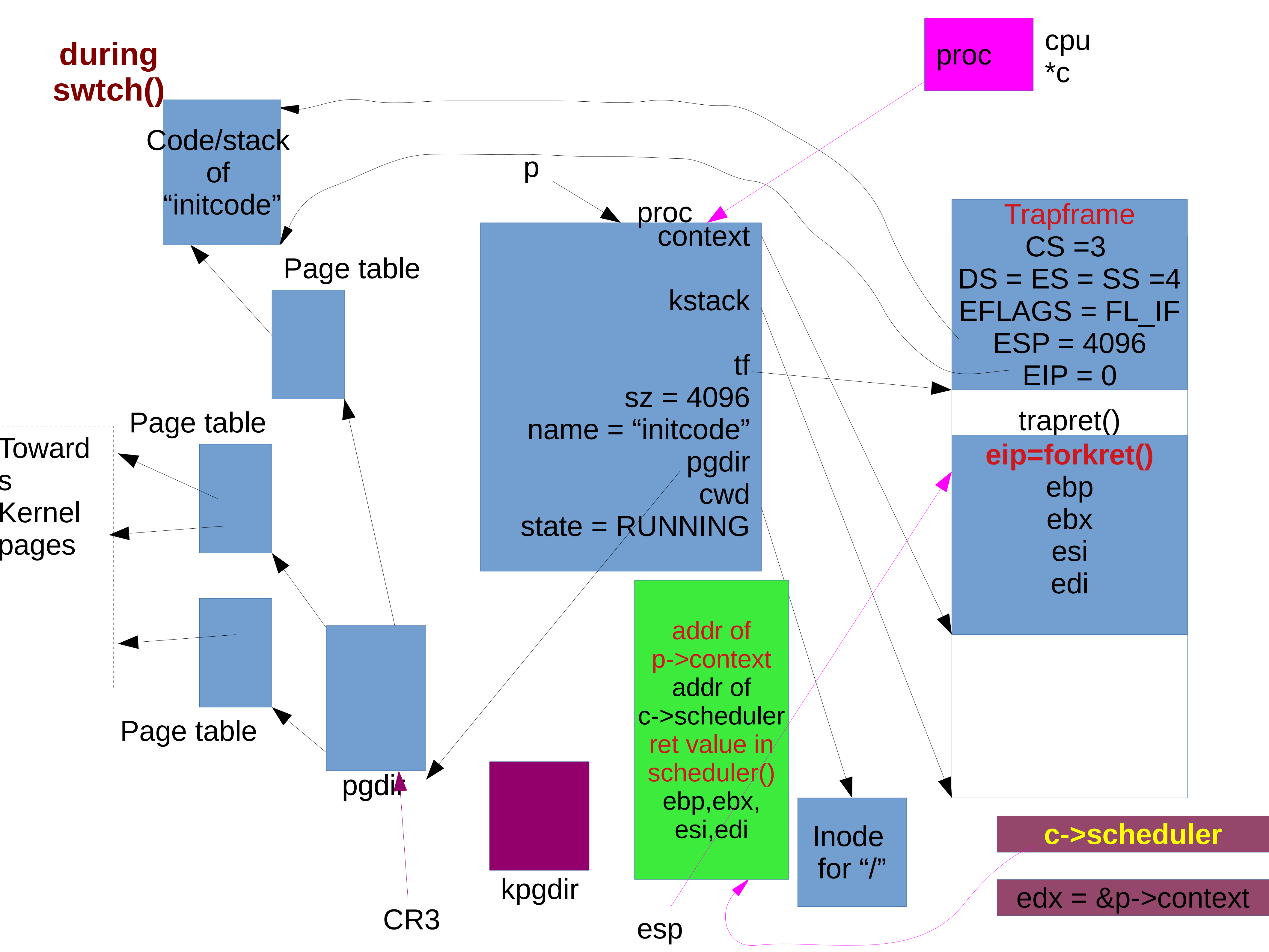
```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
```

during swtch()



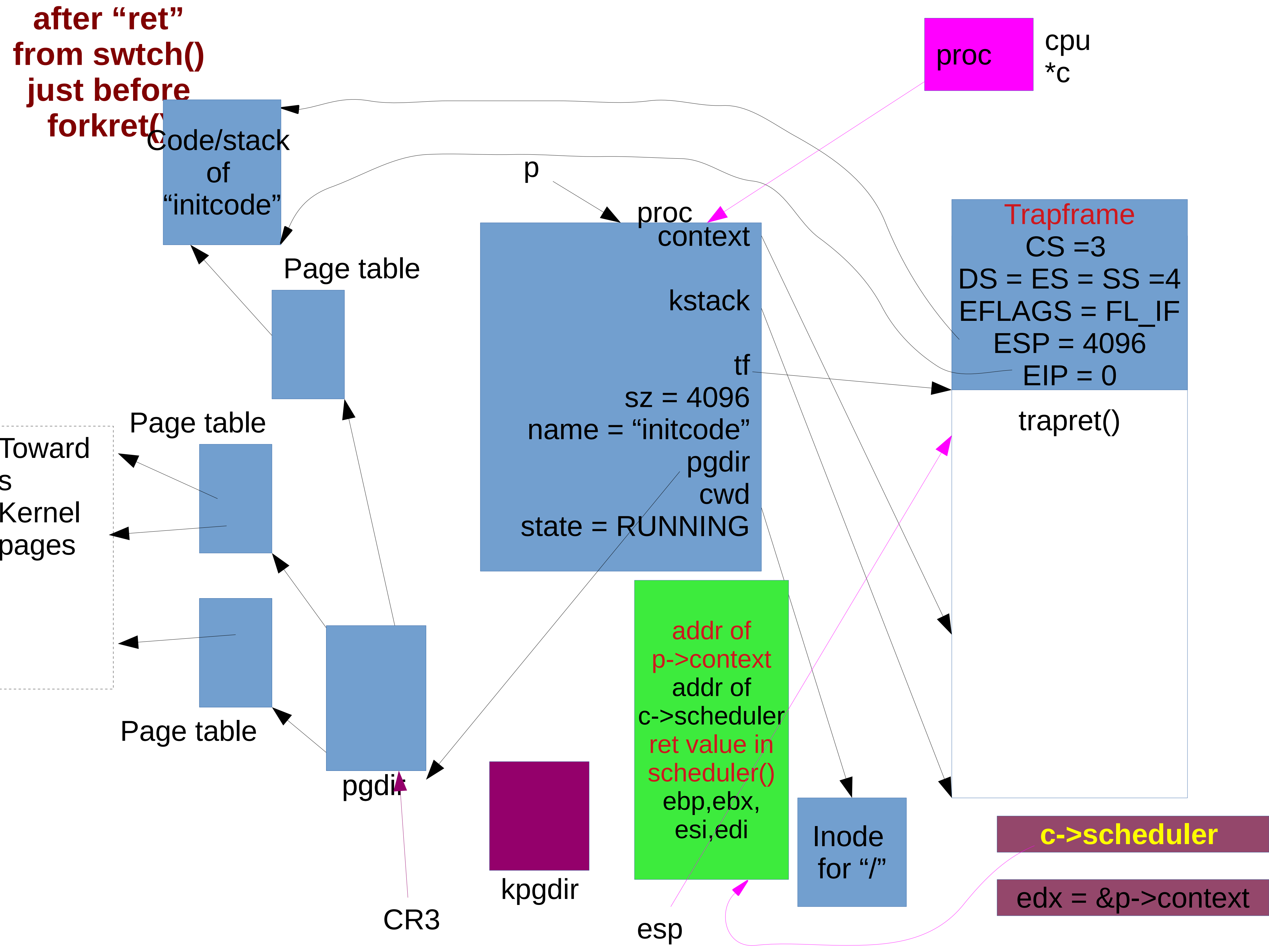
swtch

```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp     # Abhijit: newesp = newesp - 16, context restored
```


during swtch()
proc
cpu *c
Code/stack of “initcode”
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
eip=forkret()
ebp
ebx
esi
edi
Page table
Page table
Towards Kernel pages
Page table
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for “/”
c->scheduler
edx = &p->context

swtch

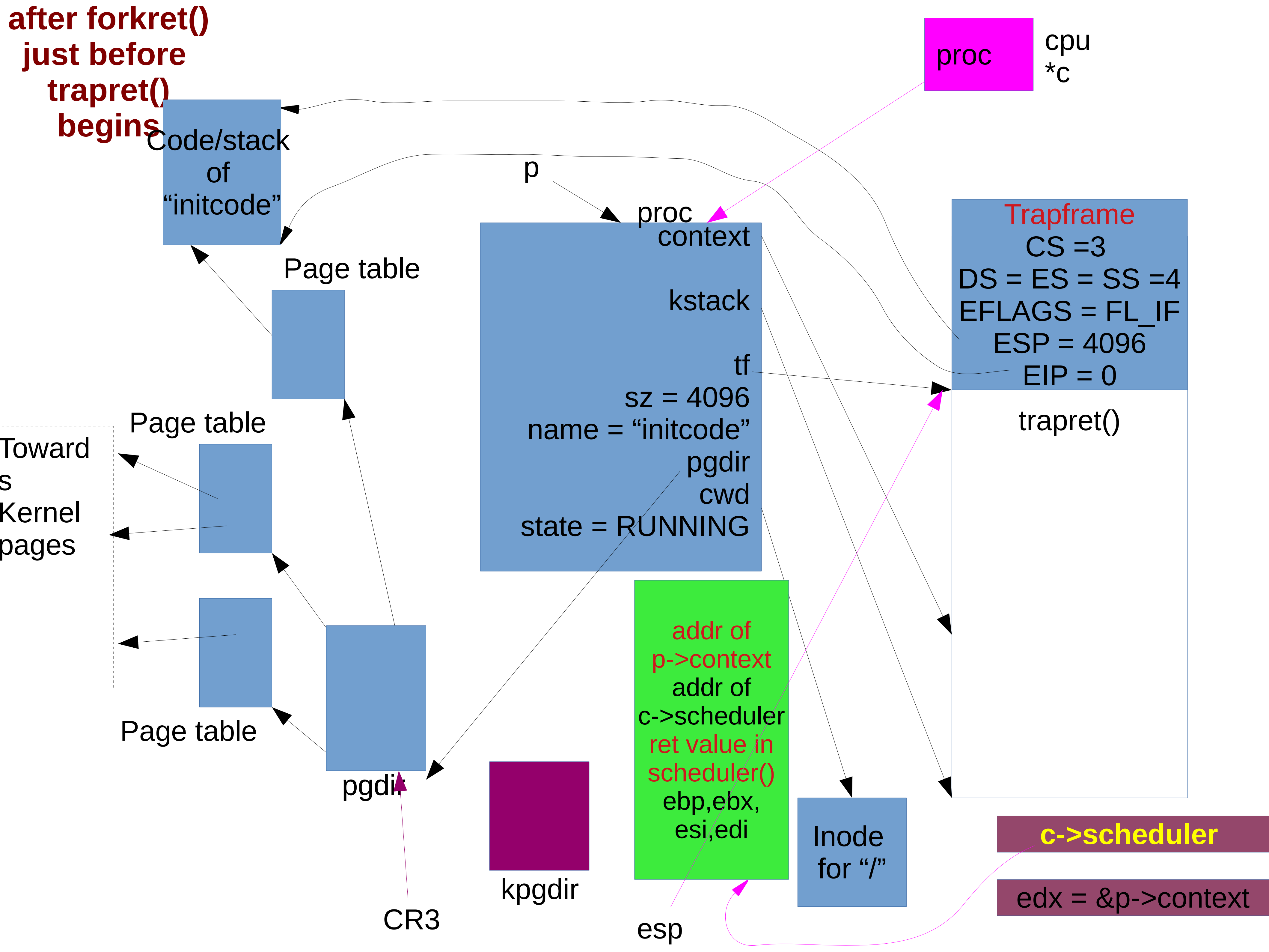
```
swtch:
  #Abhijit: swtch was called through a function call.
  #So %eip was saved on stack already
  movl 4(%esp), %eax    # Abhijit: eax = old
  movl 8(%esp), %edx    # Abhijit: edx = new
  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi            # Abhijit: esp = esp + 16
  # Switch stacks
  movl %esp, (%eax)     # Abhijit: *old = updated old stack
  movl %edx, %esp       # Abhijit: esp = new
  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp      # Abhijit: newesp = newesp - 16, context restored
  ret            # Abhijit: will pop from esp now -> function where to
return.
```



after "ret" from swtch() just before forkret()
proc
cpu *c
Code/stack of "initcode"
p
proc
context
kstack
tf
sz = 4096
name = "initcode"
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Towards Kernel pages
Page table
pgdir
CR3
kpgdir
addr of p->context
addr of c->scheduler
ret value in scheduler()
ebp,ebx, esi,edi
esp
Inode for "/"
c->scheduler
edx = &p->context

After swtch()

- Process is running in forkret()
- c->cscheduler has saved the old kernel stack
  - with the context of p, return value in scheduler, ebp, ebx, esi, edi on stack
  - remember {edi, esi, ebx, ebp, ret-value } = context
  - The c->scheduler is pointing to old context
- CR3 is pointing to process pgdir

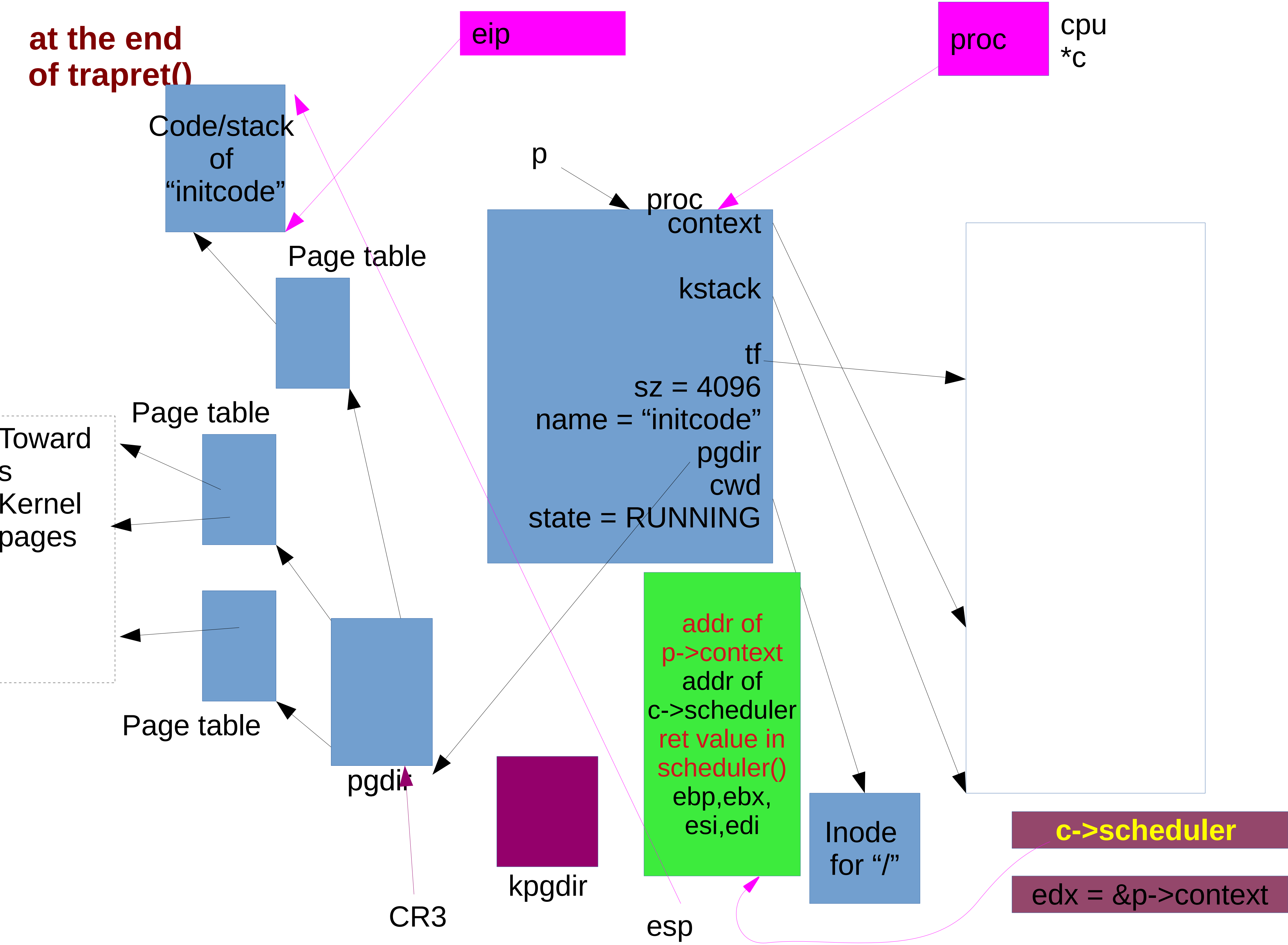


after forkret()
just before
trapret()
begins
proc
cpu
*c
Code/stack
of
“initcode”
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
Trapframe
CS =3
DS = ES = SS =4
EFLAGS = FL_IF
ESP = 4096
EIP = 0
trapret()
Page table
Page table
Towards
Kernel
pages
Page table
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for “/”
c->scheduler
edx = &p->context

After iret in trapret

- The CS, EIP, ESP will be changed
  - to values already stored on trapframe
  - this is done by iret
- Hence after this user code will run
  - On user stack!
- Hence code of *initcode* will run now

at the end of trapret()



initcode

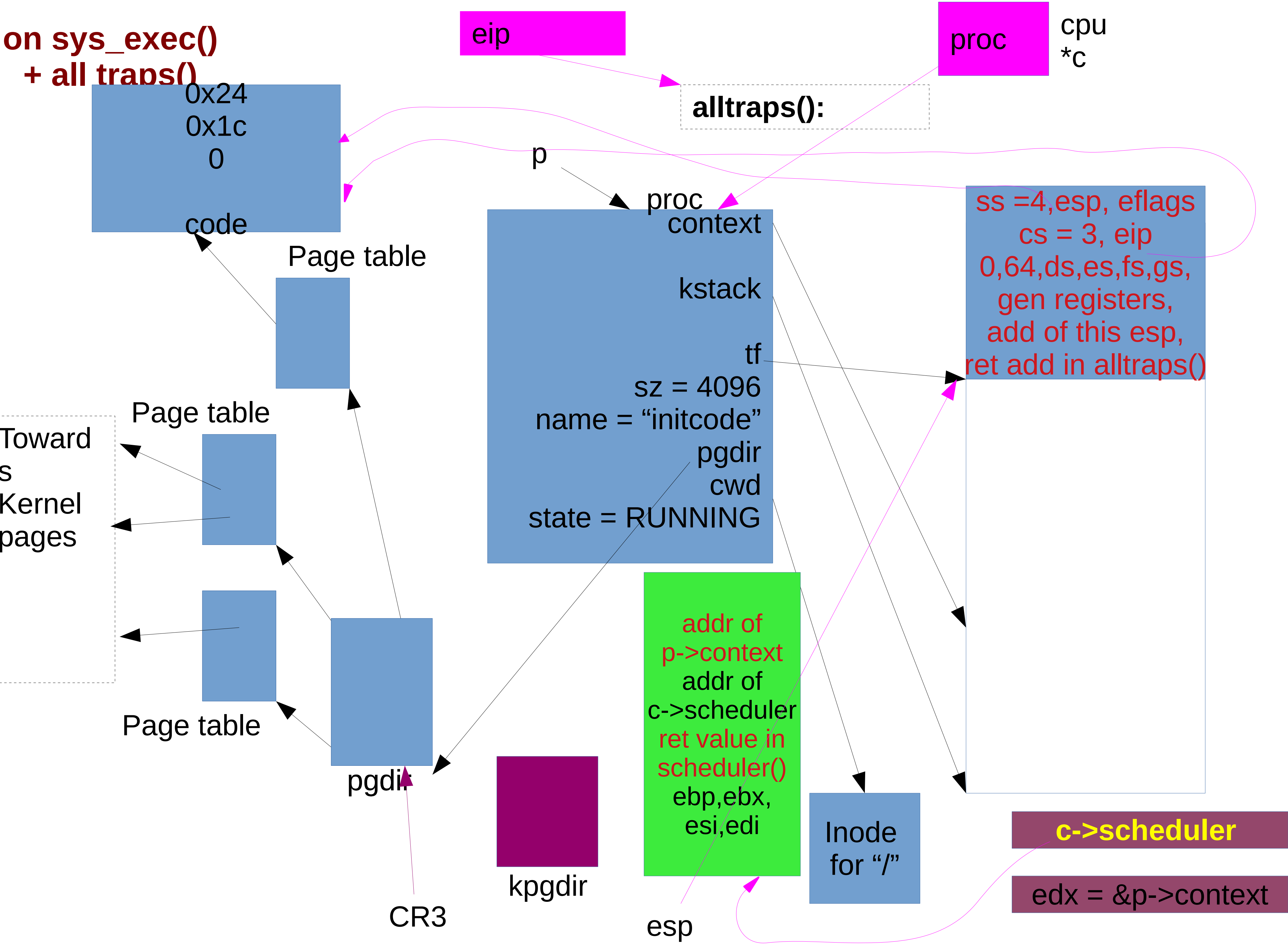
```
# char init[] = "/init\0";
init:
  .string "/init\0"

# char *argv[] = { init, 0 };
.p2align 2
argv:
  .long init
  .long 0
```

```
start:
  pushl $argv
  pushl $init
  pushl $0  // where caller pc would be
  movl $SYS_exec, %eax
  int $T_SYSCALL

# for(;;) exit();
exit:
  movl $SYS_exit, %eax
  int $T_SYSCALL
  jmp exit
```

esp

0x24 = addr of argv
0x1c = addr of init
0x0

```
00000000 <start>:
   0:     68 24 00 00 00          push   $0x24
   5:     68 1c 00 00 00          push   $0x1c
   a:     6a 00                   push   $0x0
   c:     b8 07 00 00 00          mov    $0x7,%eax
  11:     cd 40                   int    $0x40

00000013 <exit>:
  13:     b8 02 00 00 00          mov    $0x2,%eax
  18:     cd 40                   int    $0x40
  1a:     eb f7                   jmp    13 <exit>
0000001c <init>:
          "/init\0"
00000024 <argv>:
      1c 00
      00 00
```


on sys_exec()
+ all traps()
eip
proc
cpu
*c
alltraps():
0x24
0x1c
0
code
p
proc
context
kstack
tf
sz = 4096
name = “initcode”
pgdir
cwd
state = RUNNING
ss =4,esp, eflags
cs = 3, eip
0,64,ds,es,fs,gs,
gen registers,
add of this esp,
ret add in alltraps()
Page table
Page table
Towards
Kernel
pages
Page table
pgdir
CR3
kpgdir
addr of
p->context
addr of
c->scheduler
ret value in
scheduler()
ebp,ebx,
esi,edi
esp
Inode
for “/”
c->scheduler
edx = &p->context

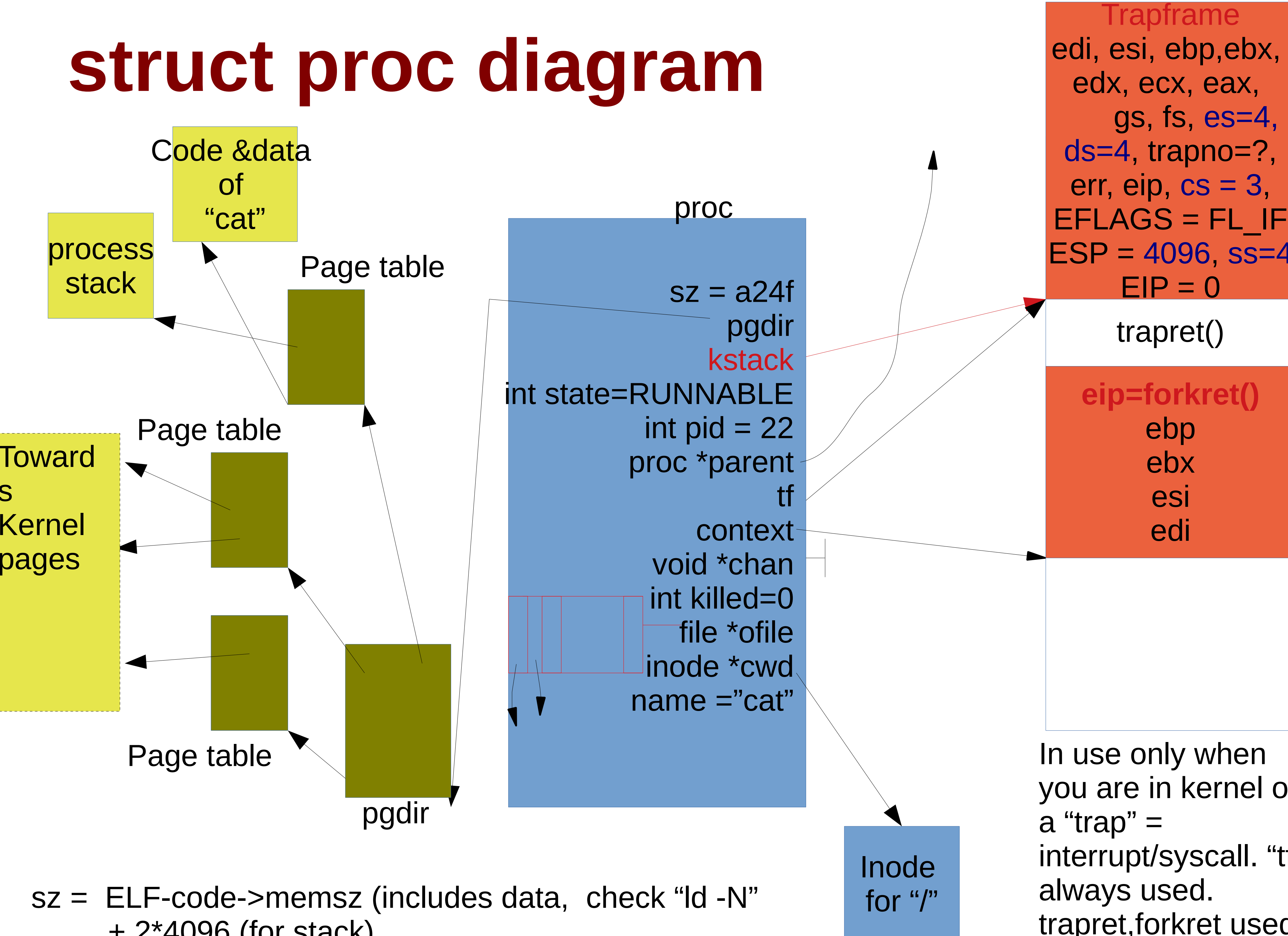
Understanding fork() and exec()

First, revising some concepts already learnt then code of fork(), exec()

First process creation
Let’s revisit struct 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;       // Process state. allocated, ready to run, running, wait-
ing for I/O, or exiting.
  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. Process’s context
  void *chan;                 // If non-zero, sleeping on chan. More when we discuss
sleep, wakeup
  int killed;                 // If non-zero, have been killed
  struct file *ofile[NOFILE]; // Open files, used by open(), read(),...
  struct inode *cwd;          // Current directory, changed with “chdir()”
  char name[16];              // Process name (for debugging)
};
```

struct proc diagram



sz = ELF-code->memsz (includes data, check “ld -N”
+ 2*4096 (for stack)

In use only when you are in kernel on a “trap” = interrupt/syscall. “tf” always used. trapret,forkret used during fork()

fork()/exec() are syscalls. On every syscall this happens

- Fetch the n'th descriptor from the IDT, where n is the argument of int.
- Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
- Save %esp and %ss in CPU-internal registers, but only if the target segment selector's PL < CPL.
  - Switching from user mode to kernel mode. Hence save user code's SS and ESP
- Load %ss and %esp from a task segment descriptor.
  - Stack changes to kernel stack now. TS descriptor is on GDT, index given by TR register. See switchuvm()
- Push %ss. // optional
- Push %esp. // optional (also changes ss,esp using TSS)
- Push %eflags.
- Push %cs.
- Push %eip.
- Clear the IF bit in %eflags, but only on an interrupt.
- Set %cs and %eip to the values in the descriptor.

After “int” ‘s job is done

- IDT was already set, during idtinit()
  - Remember vectors.S – gives jump locations for each interrupt
- “int 64” ->jump to 64^{th} entry in vector table

```
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

  - So now stack has ss, esp,eflags, cs, eip, 0 (for error code), 64
  - Next run alltraps from trapasm.S

alltraps:

```
# Build trap frame.
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal // push all gen purpose regs
# Set up data segments.
movw $(SEG_KDATA<<3), %ax
movw %ax, %ds
movw %ax, %es
# Call trap(tf), where tf=%esp
pushl %esp # first arg to trap()
call trap
addl $4, %esp
```

- **Now stack contains**

  ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi

  - **This is the struct trapframe !**
  - **So the kernel stack now contains the trapframe**
  - **Trapframe is a part of kernel stcak**

trap()

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
  switch(tf->trapno){
  .....
```

- Argument is trapframe
- In alltraps
  - Before “call trap”, there was “push %esp” and stack had the trapframe
  - Remember calling convention --> when a function is called, the stack contains the arguments in reverse order (here only 1 arg)

trap()

- Has a switch
  - switch(tf->trapno)
  - Q: who set this trapno?
- Depending on the type of trap
  - Call interrupt handler

- Timer
  - wakeup(&ticks)
- IDE: disk interrupt
  - Ideintr()
- KBD
  - Kbdintr()
- COM1
  - Uatrintr()
- If Timer
  - Call yield() -- calls sched()
- If process was killed (how is that done?
  - Call exit()!

when trap() returns

- #Back in alltraps

```
  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
```

-

- Stack had (trapframe)
  - ss, esp,eflags, cs, eip, 0 (for error code), 64, ds, es, fs, gs, eax, ecx, edx, ebx, oesp, ebp, esi, edi, esp
- add $4 %esp
  - esp
- popal
  - eax, ecx, edx, ebx, oesp, ebp, esi, edi
- Then gs, fs, es, ds
- add $0x8, %esp
  - 0 (for error code), 64
- iret
  - ss, esp,eflags, cs, eip,

understanding fork()

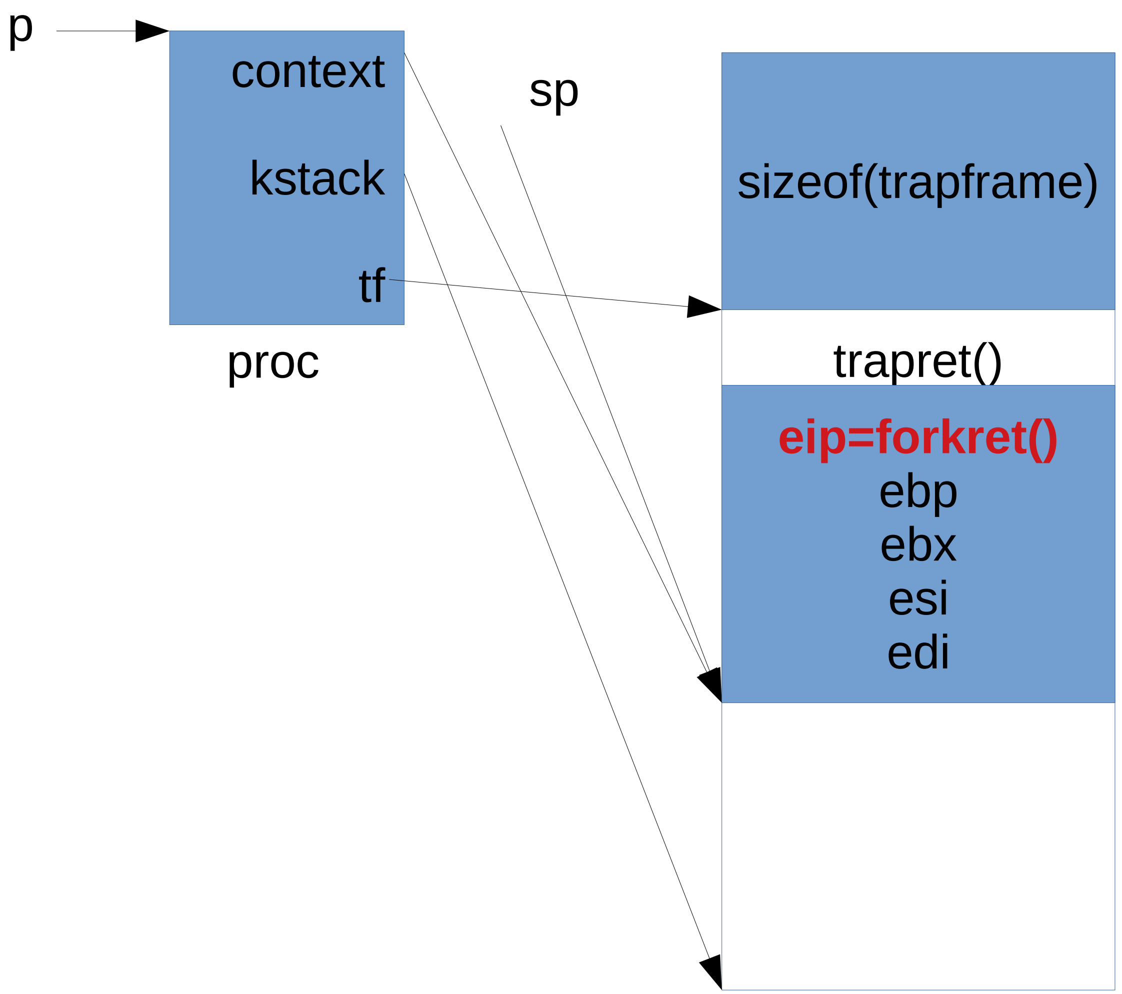
- **What should fork do?**
  - **Create a copy of the existing process**
  - **child is same as parent, except pid, parent-child relation, return value (pid or 0)**
  - **Please go through every member of struct proc, understand it's meaning to appreciate what fork() should do**
  - **create a struct proc, and**
    - **duplicate pages, page directory, sz, state,trapframe,context, ofile (and files!), cwd, name**
    - **modify: pid, parent, trapframe, state**

understanding fork()

```
int
sys_fork(void)
{
  return fork();
}
```

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

after allocproc()
-- we studied this -- same as creation of first process



understanding fork()

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

- copy the pages, page tables, page directory
  - no copy on write here!
  - Rewind if operation of copyuvm() fails
- copy size
- set parent of child
- copy trapframe (structure is copied)

understanding fork()->copyuvm()

```
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
  pde_t *d;  pte_t *pte;  uint pa, i, flags;
  char *mem;
  if((d = setupkvm()) == 0)
    return 0;
  for(i = 0; i < sz; i += PGSIZE){
    if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
      panic("copyuvm: pte should exist");
    if(!(*pte & PTE_P))
      panic("copyuvm: page not present");
    pa = PTE_ADDR(*pte);
    flags = PTE_FLAGS(*pte);
    if((mem = kalloc()) == 0)
      goto bad;
    memmove(mem, (char*)P2V(pa), PGSIZE);
    if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
      kfree(mem);
      goto bad;
    }
  }
  return d;
bad:
  freevm(d);
  return 0;
}
```

- Map kernel pages
- for every page in parent’s VM address space
  - allocate a PTE for child
  - set flags
  - copy data
  - map pages in child’s page directory/tables

understanding fork()

```
np->tf->eax = 0;
for(i = 0; i < NOFILE; i++)
  if(curproc->ofile[i])
    np->ofile[i] = filedup(curproc->ofile[i]);
np->cwd = idup(curproc->cwd);
safestrcpy(np->name, curproc->name, sizeof(curproc->name));
pid = np->pid;
acquire(&ptable.lock);
np->state = RUNNABLE;
release(&ptable.lock);
```

- set return value of child to 0
  - eax contains return value, it’s on TF
- copy each struct file
- copy current working dir inode
- copy name
- set pid of child
- set child “RUNNABLE”

exec() - different prototype

- **int exec(char*, char**);**
  - **usage: to print README and test.txt using “cat”**

```
int main(int argc, char *argv[])
{
   char *cmd = "/cat";
   char *argstr[4] = { "/cat", "README",
"test.txt", 0};
   exec(cmd, argstr);
}
```

note: to really run this code in xv6, you need to make changes to Makefile. First, add this program to UPROGS, then write a file test.txt using Linux, and add ‘test.txt’ to list of files in ‘mkfs’ target in Makefile

sys_exec()

```
int
sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;
  uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))
      return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;
      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- argstr(n,), argint(n,)
  - Fetch the n'th argument from *process stack* using p->tf->esp + offset
  - Again: revise calling conventions
  - 0'th argument: name of executable file
  - 1st Argument: address of the array of arguments
    - store in *uargv*

sys_exec()

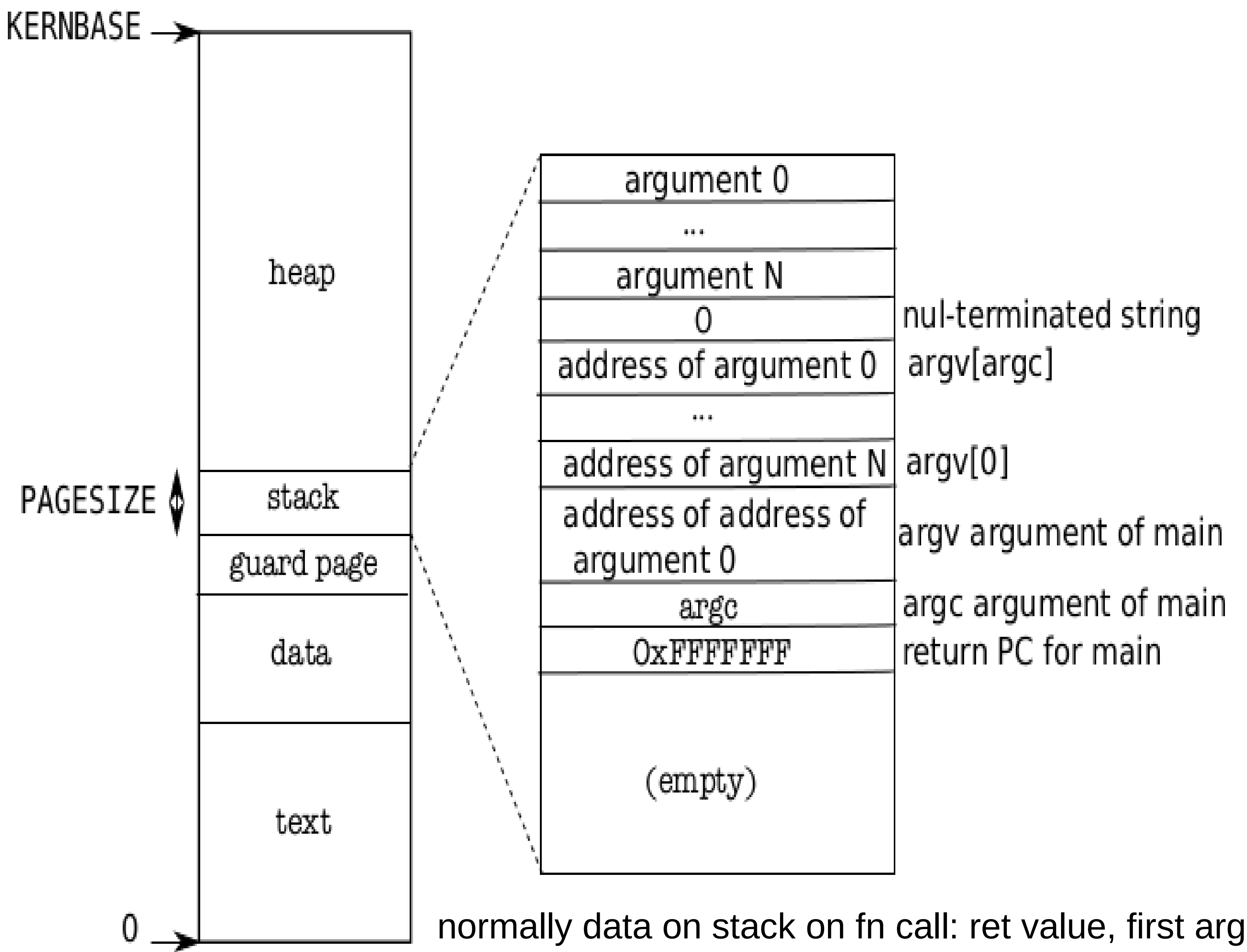
```
int sys_exec(void)
{
  char *path, *argv[MAXARG];
  int i;   uint uargv, uarg;
  if(argstr(0, &path) < 0 || argint(1, 
(int*)&uargv) < 0){
    return -1;
  }
  memset(argv, 0, sizeof(argv));
  for(i=0;; i++){
    if(i >= NELEM(argv))     return -1;
    if(fetchint(uargv+4*i, (int*)&uarg) < 0)
      return -1;
    if(uarg == 0){
      argv[i] = 0;      break;
    }
    if(fetchstr(uarg, &argv[i]) < 0)
      return -1;
  }
  return exec(path, argv);
}
```

- the local array argv[] (allocated on kernel stack, obviously) set to 0
- fetch every next argument from array of arguments
  - Sets the address of argument in argv[1]
- call exec
  - beware: mistake to assume that this exec() is the exec() called from user code! NO!

What should exec() do?

- Remember, it came from fork()
  - so proc & within it tf, context, kstack, pgdir-tables-pages, all exist.
  - Code, stack pages exist, and mappings exist through proc->pgdir
- Hence
  - read the ELF executable file (argv[0])
  - create a new page dir – create mappings for kernel and user code+data; copy data from ELF to these pages (later discard old pagedir)
  - Copy the argv onto the user stack – so that when new process starts it has it's main(argc, argv[]) built
  - set values of other fields in proc to start program correctly

User stack after call to exec() is over



normally data on stack on fn call: ret value, first arg, second arg, ...

main(int argc, char *argv[])

argv[] is address of array of string; string itself is an adress. Hence 2 levels of indirection on stack

exec()

```
int
exec(char *path, char **argv)
{
...
  uint argc, sz, sp,
ustack[3+MAXARG+1];
...
  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
```

- **ustack**
  - **used to build the arguments to be pushed on user-stack**
- **namei**
  - **get the inode of the executable file**

exec()

```
// Check ELF header
if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
  goto bad;
if(elf.magic != ELF_MAGIC)
  goto bad;

if((pgdir = setupkvm()) == 0)
  goto bad;
```

- readi
  - read ELF header
- setupkvm()
  - creating a *new* page directory and mapping kernel pages

exec()

```
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) <
0)
    goto bad;
}
```

- Read ELF program headers from ELF file
- Map the code/data into pagedir-pagetable-pages
- Copy data from ELF file into the pages allocated

exec()

```
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

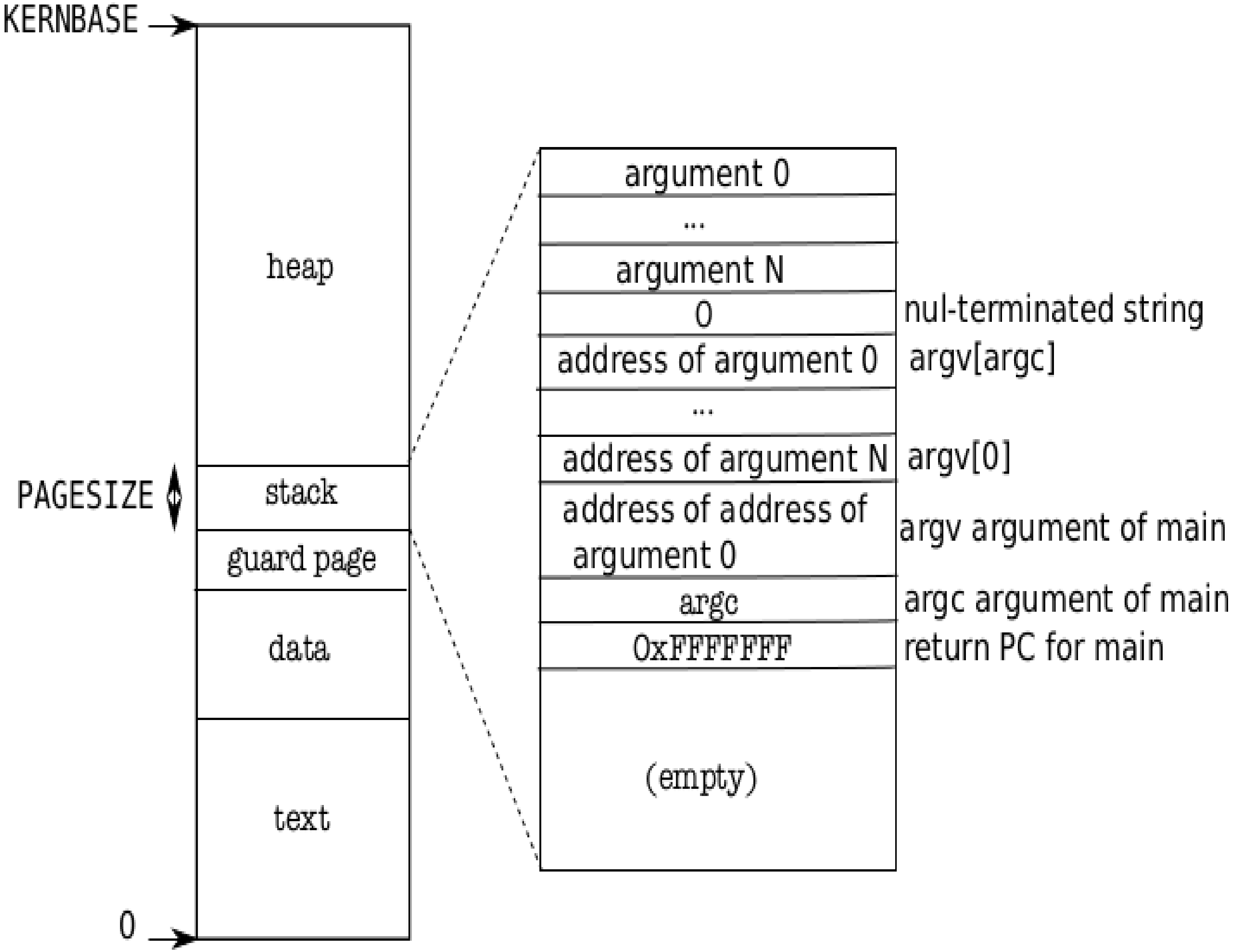
- Allocate 2 pages on top of proc->sz
- One page for stack
- one page for guard page
- Clear the valid flag on guard page

exec()

```
  // Push argument strings, prepare rest of stack
in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc],
strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;
  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer
  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;
```

- For each entry in argv[]
  - copy it on user-stack
  - remember it's location on user stack in ustack
- add extra entries (to be copied to user stack) to ustack
- copy argc, argv pointer
- take sp to bottom
- copy ustack to user stack

This is what the code on earlier slide did



exec()

```
// Save program name for debugging.
for(last=s=path; *s; s++)
  if(*s == '/')
    last = s+1;
safestrcpy(curproc->name, last,
sizeof(curproc->name));

// Commit to the user image.
oldpgdir = curproc->pgdir;
curproc->pgdir = pgdir;
curproc->sz = sz;
curproc->tf->eip = elf.entry;  // main
curproc->tf->esp = sp;
switchuvm(curproc);
freevm(oldpgdir);
return 0;
```

- copy name of new process in proc->name
- change to new page directory
- change new size
- tf->eip will be used when we return from exec() to jump to user code. Set to to first instruction of code, given by elf.entry
- Set user stack pointer to “sp” (bottom of stack of arguments)
- Update TSS, change CR3 to newpagedir
- free old page dir

return 0 from exec()?

- We know exec() does not return !
- This was exec() function !
  - Returns to sys_exec()
- sys_exec() also returns , where?
  - Remember we are still in kernel code, running on kernel stack. p->kstack has the trapframe setup
  - There is context struct on stack. Why?
  - sys_exec() returns to trapret(), the trap frame will be popped !
    - with “iret” jump into new program !
  - New program is not old program , which could have accessed return value of sys_exec()