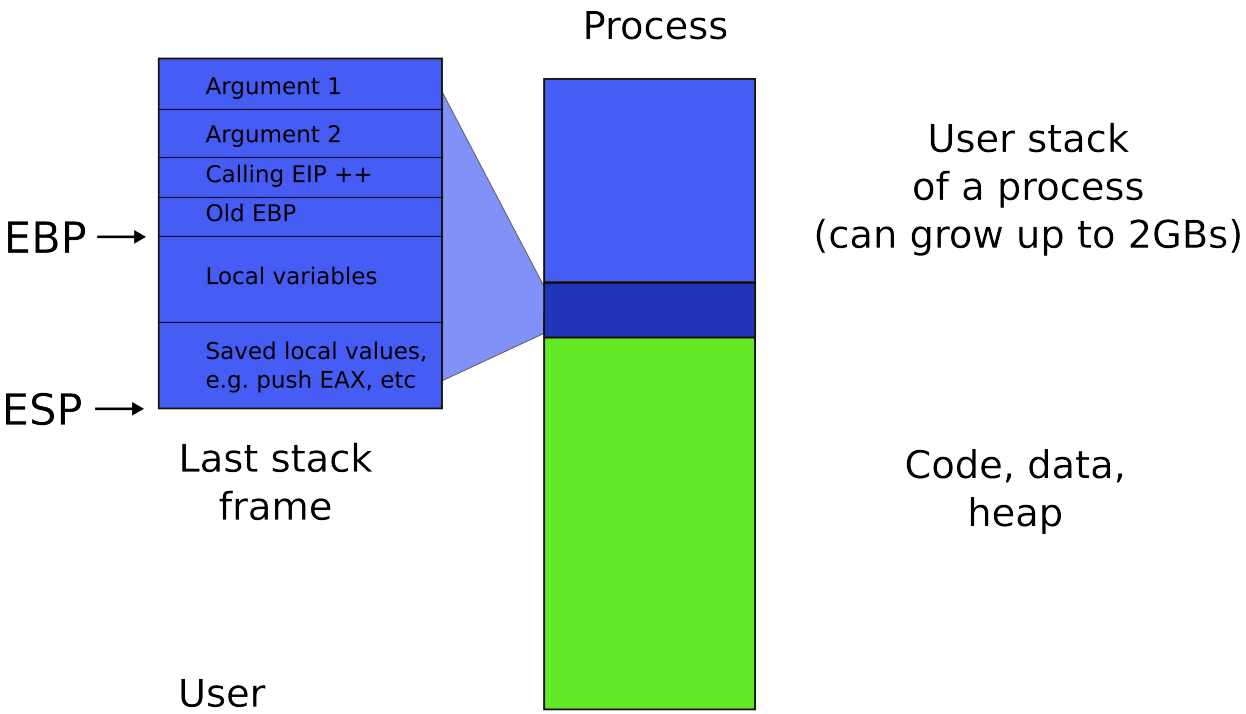
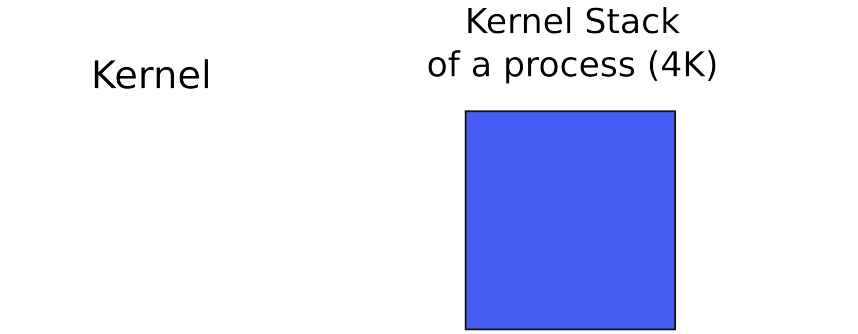
ICS143A: Principles of Operating Systems

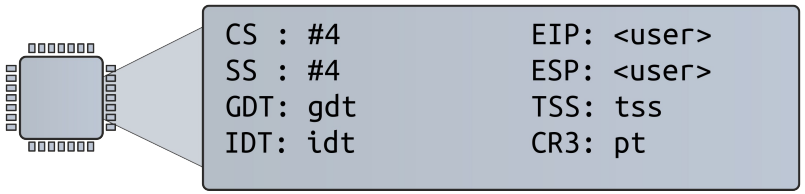
Lecture 14: System calls & the first process

Anton Burtsev
November, 2017

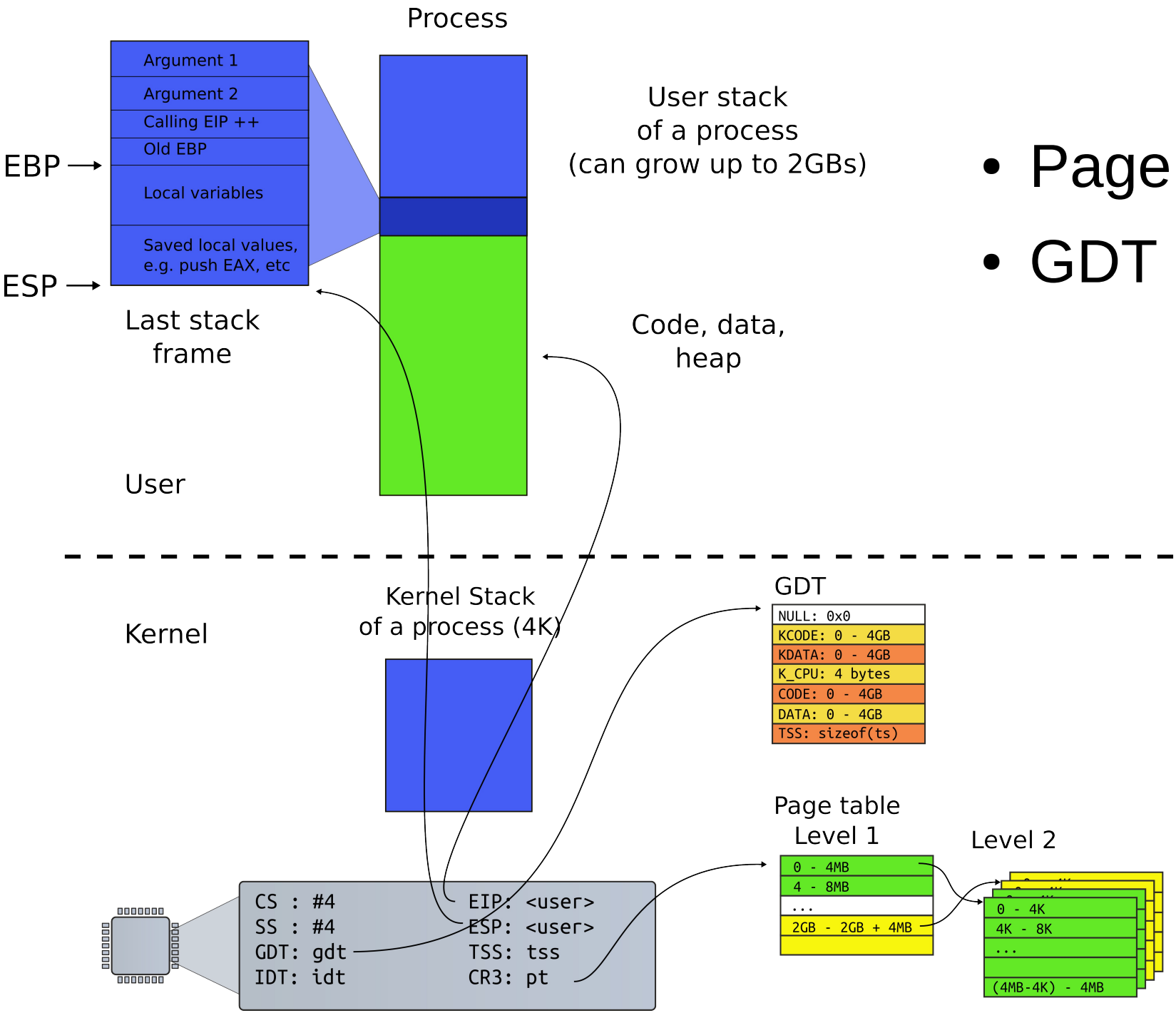
System calls



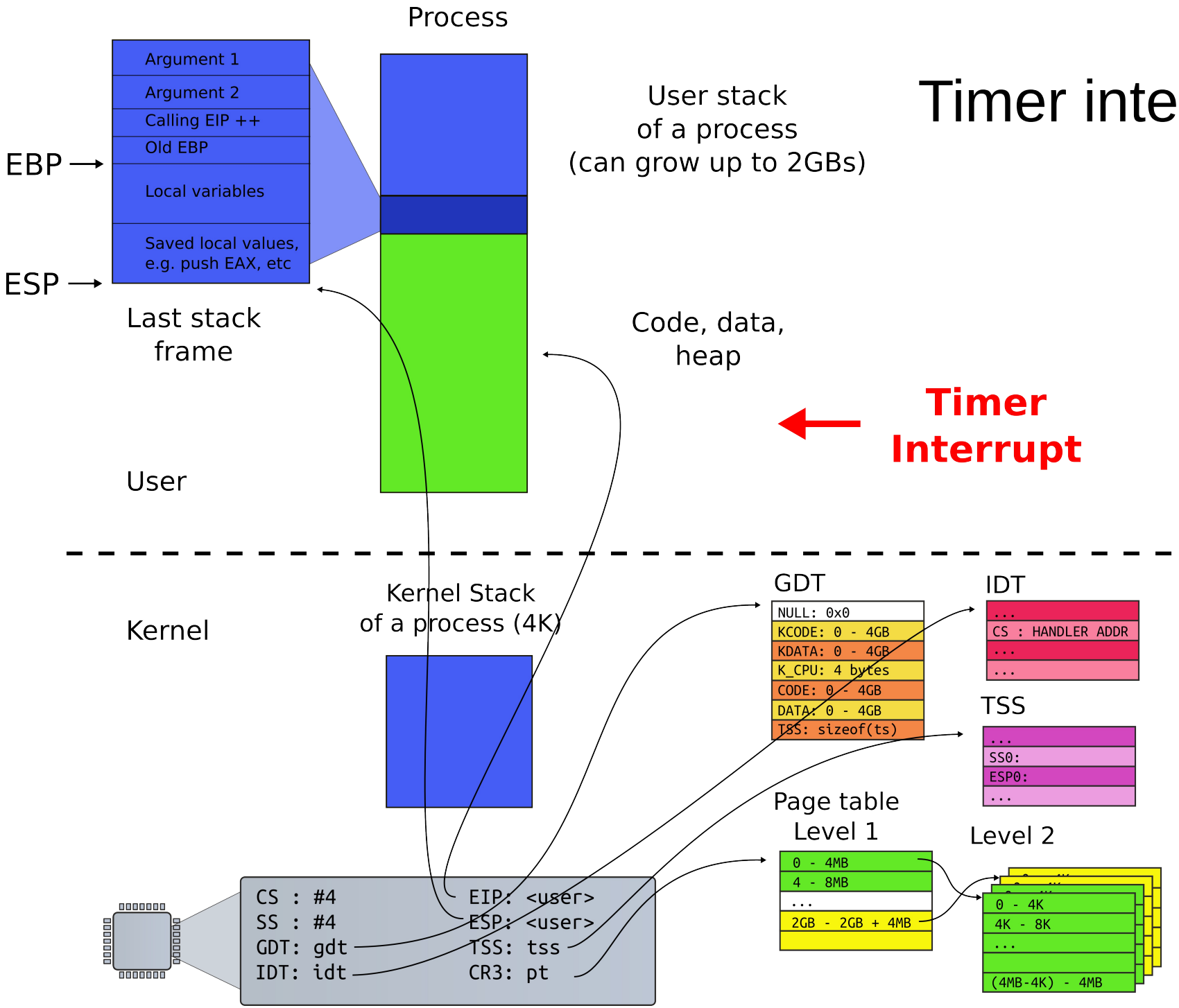




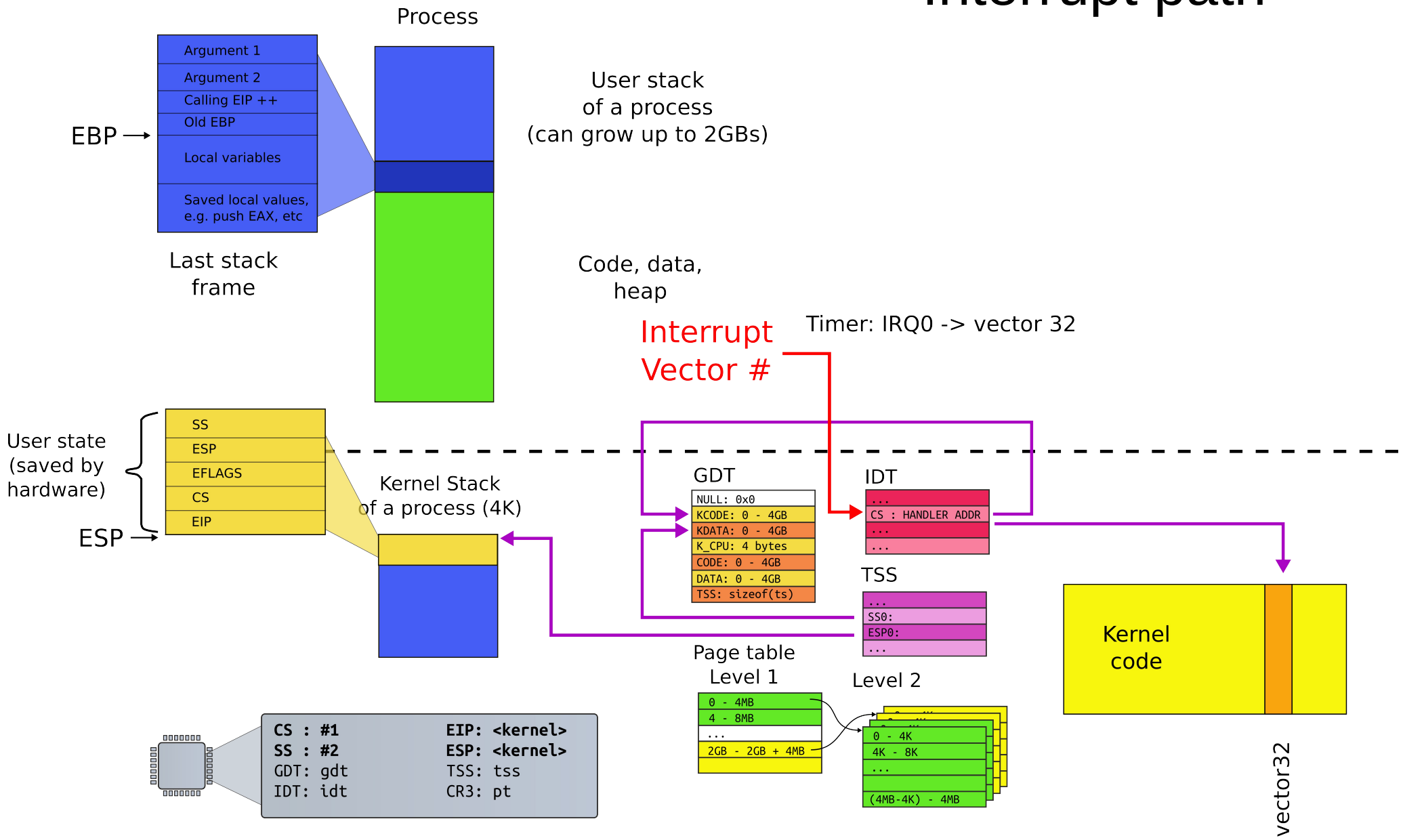
- User mode
- Two stacks
  - Kernel and user
  - Kernel stack is empty



- Page table
- GDT


Process
Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc
EBP
ESP
Last stack frame
User stack of a process (can grow up to 2GBs)
Code, data, heap
User
Timer Interrupt
Kernel
Kernel Stack of a process (4K)
GDT
NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 4 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(ts)
IDT
...
CS : HANDLER ADDR
...
...
TSS
...
SS0:
ESP0:
...
Page table Level 1
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB
Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
CS : #4
SS : #4
GDT: gdt
IDT: idt
EIP: <user>
ESP: <user>
TSS: tss
CR3: pt

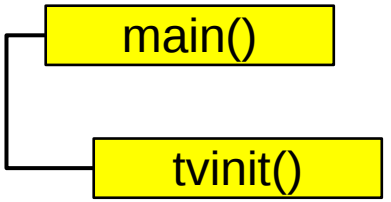
Interrupt path



Initialize IDT

- tvinit() is called from main()

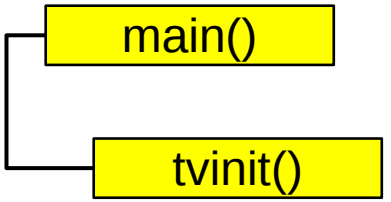
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



Initialize IDT

- tvinit() is called from main()

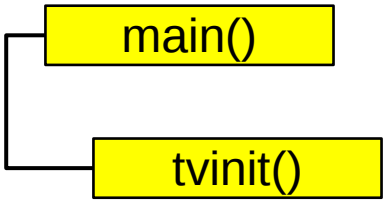
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



Initialize IDT

- A couple of important details

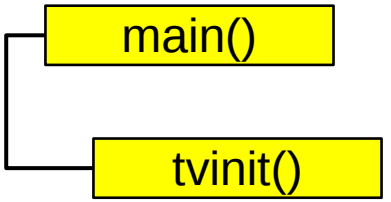
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



Initialize IDT

- Only int T_SYSCALL can be called from user-level

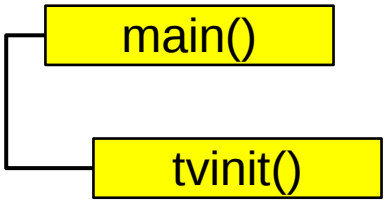
```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



Initialize IDT

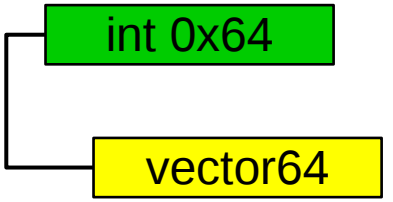
- Syscall is a “trap”
- i.e., doesn't disable interrupts

```
3316 void
3317 tvinit(void)
3318 {
3319   int i;
3320
3321   for(i = 0; i < 256; i++)
3322     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3323   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3,
                vectors[T_SYSCALL], DPL_USER);
3324
3325   initlock(&tickslock, "time");
3326 }
```



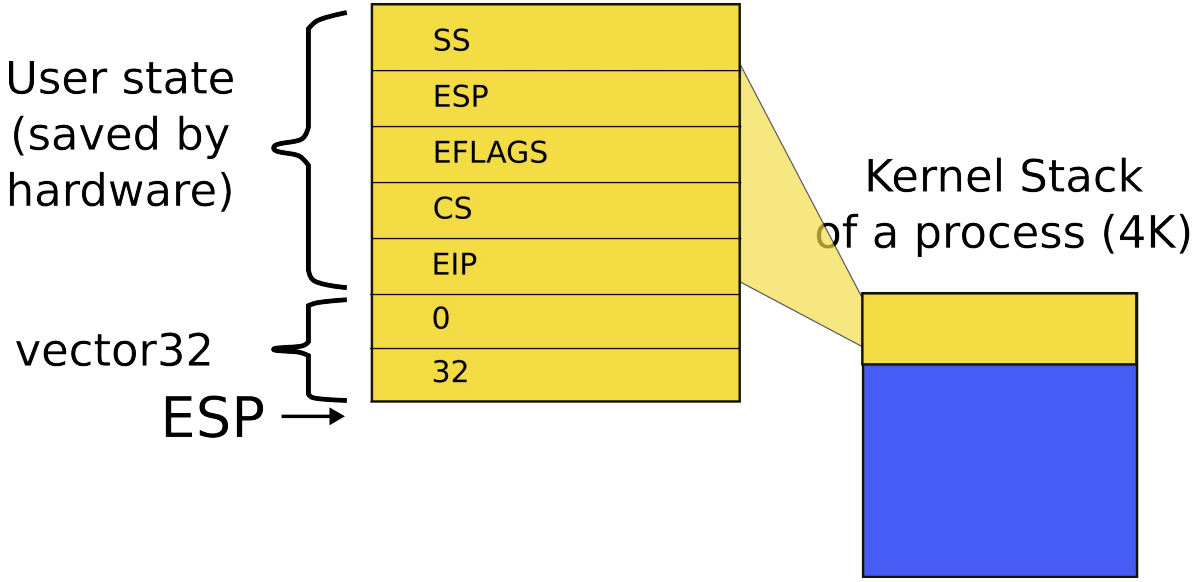
Where does IDT (entry 64) point to?

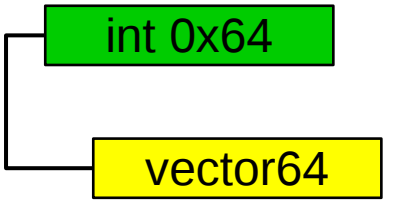
```
vector64:
  pushl $0     // error code
  pushl $64    // vector #
  jmp alltraps
```



- Automatically generated
- From vectors.pl
  - vector.S

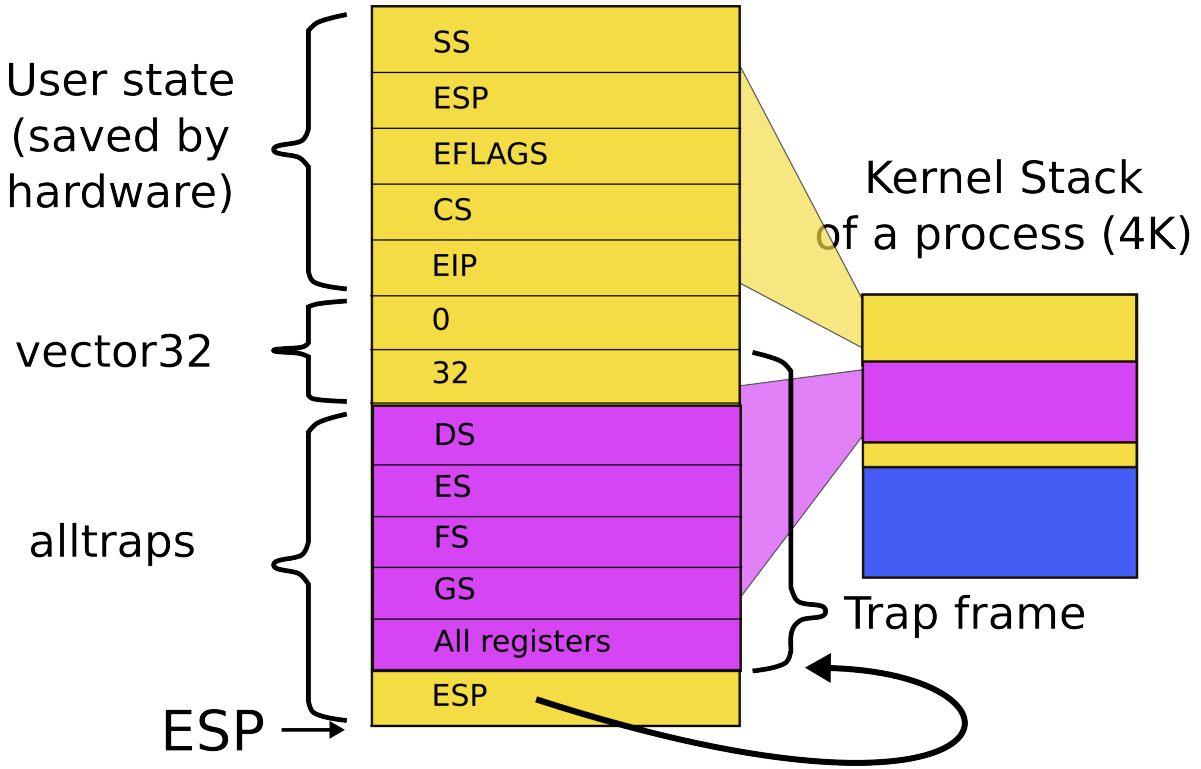
Kernel stack after interrupt



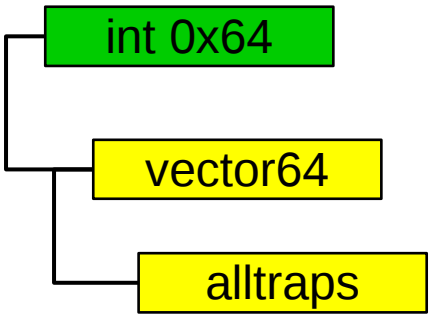


Call stack: vector32()

Kernel stack after interrupt



Call stack: vector32()
alltraps()

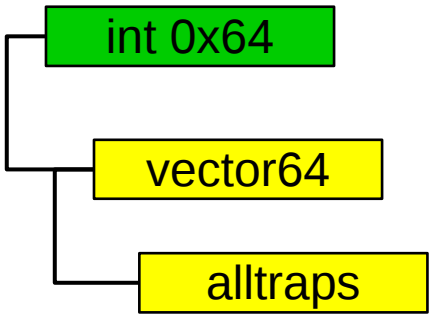


Syscall number

- System call number is passed in the %eax register
  - To distinguish which syscall to invoke,
    - e.g., sys_read, sys_exec, etc.
- alltrap() saves it along with all other registers

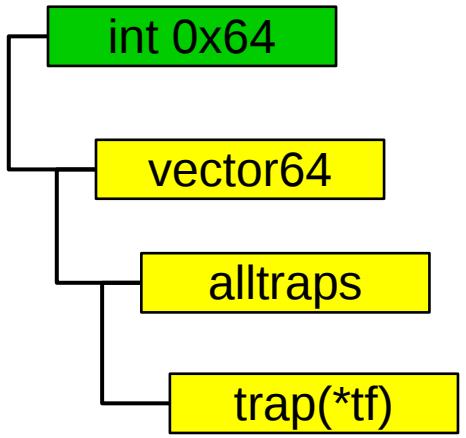
alltraps()

```
3254 alltraps:
3255 # Build trap frame.
3256 pushl %ds
3257 pushl %es
3258 pushl %fs
3259 pushl %gs
3260 pushal
3261
3262 # Set up data and per-cpu segments.
3263 movw $(SEG_KDATA<<3), %ax
3264 movw %ax, %ds
3265 movw %ax, %es
3266 movw $(SEG_KCPU<<3), %ax
3267 movw %ax, %fs
3268 movw %ax, %gs
3269
3270 # Call trap(tf), where tf=%esp
3271 pushl %esp
3272 call trap
```



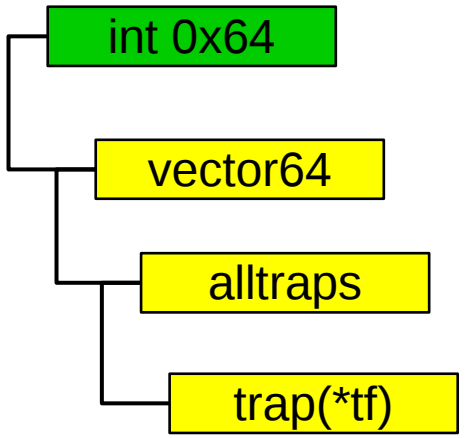
```
3351 trap(struct trapframe *tf)
3352 {
...
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
3365     if(cpu->id == 0){
3366     acquire(&tickslock);
3367     ticks++;
3368     wakeup(&ticks);
3369     release(&tickslock);
3370   }
3372   break;
...
3423   if(proc && proc->state == RUNNING
          && tf->trapno == T_IRQ0+IRQ_TIMER)
3424     yield();
```

Remember we went to trap() to handle timer interrup



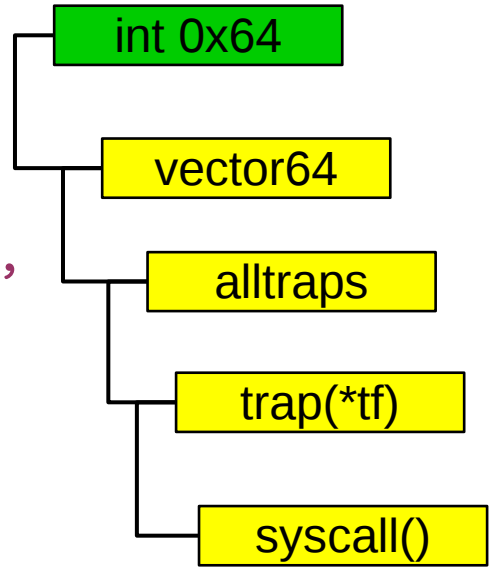
```
3351 trap(struct trapframe *tf)
3352 {
3353   if(tf->trapno == T_SYSCALL){
3354     if(proc->killed)
3355       exit();
3356     proc->tf = tf;
3357     syscall();
3358     if(proc->killed)
3359       exit();
3360     return;
3361   }
3362
3363   switch(tf->trapno){
3364   case T_IRQ0 + IRQ_TIMER:
```

Same for syscalls



syscall(): get the number from the trap frame

```
3625 syscall(void)
3626 {
3627   int num;
3628
3629   num = proc->tf->eax;
3630   if(num > 0 && num < NELEM(syscalls) && syscalls[num])
       {
3631     proc->tf->eax = syscalls[num]();
3632   } else {
3633     cprintf("%d %s: unknown sys call %d\n",
3634     proc->pid, proc->name, num);
3635     proc->tf->eax = -1;
3636   }
3637 }
```



syscall(): process a syscall from the table

```
3625 syscall(void)
3626 {
3627   int num;
3628
3629   num = proc->tf->eax;
3630   if(num > 0 && num < NELEM(syscalls) && syscalls[num])
       {
3631     proc->tf->eax = syscalls[num]();
3632   } else {
3633     cprintf("%d %s: unknown sys call %d\n",
3634     proc->pid, proc->name, num);
3635     proc->tf->eax = -1;
3636   }
3637 }
```

System call table

```
3600 static int (*syscalls[])(void) = {
3601   [SYS_fork] sys_fork,
3602   [SYS_exit] sys_exit,
3603   [SYS_wait] sys_wait,
3604   [SYS_pipe] sys_pipe,
3605   [SYS_read] sys_read,
3606   [SYS_kill] sys_kill,
3607   [SYS_exec] sys_exec,
3608   [SYS_fstat] sys_fstat,
3609   [SYS_chdir] sys_chdir,
3610   [SYS_dup] sys_dup,
3611   [SYS_getpid] sys_getpid,
3612   [SYS_sbrk] sys_sbrk,
3613   [SYS_sleep] sys_sleep,
3614   [SYS_uptime] sys_uptime,
3615   [SYS_open] sys_open,
3616   [SYS_write] sys_write,
3617   [SYS_mknod] sys_mknod,
3618   [SYS_unlink] sys_unlink,
3619   [SYS_link] sys_link,
3620   [SYS_mkdir] sys_mkdir,
3621   [SYS_close] sys_close,
3622 };
```

What do you think is the first system call xv6 executes?

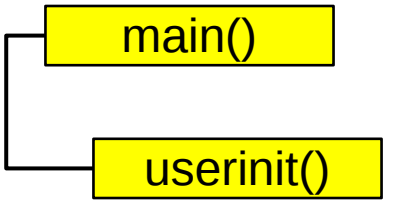
```
1317 main(void)
1318 {
1319   kinit1(end, P2V(4*1024*1024)); // phys page allocator
1320   kvmalloc(); // kernel page table
1321   mpinit(); // detect other processors
...
1323   seginit(); // segment descriptors
...
1330   tvinit(); // trap vectors
...
1338   userinit(); // first user process
1339   mpmain(); // finish this processor's setup
1340 }
```

main()

Userinit() – create first process

- Allocate process structure
  - Information about the process

```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[],
                  _binary_initcode_size[];
...
2509   p = allocproc();
2510   initproc = p;
2511   if((p->pgdir = setupkvm()) == 0)
2512     panic("userinit: out of memory?");
2513   inituvm(p->pgdir, _binary_initcode_start,
              (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```

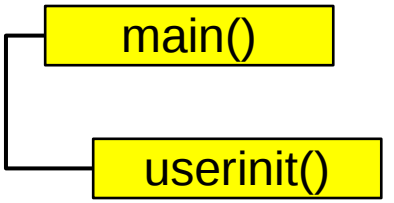


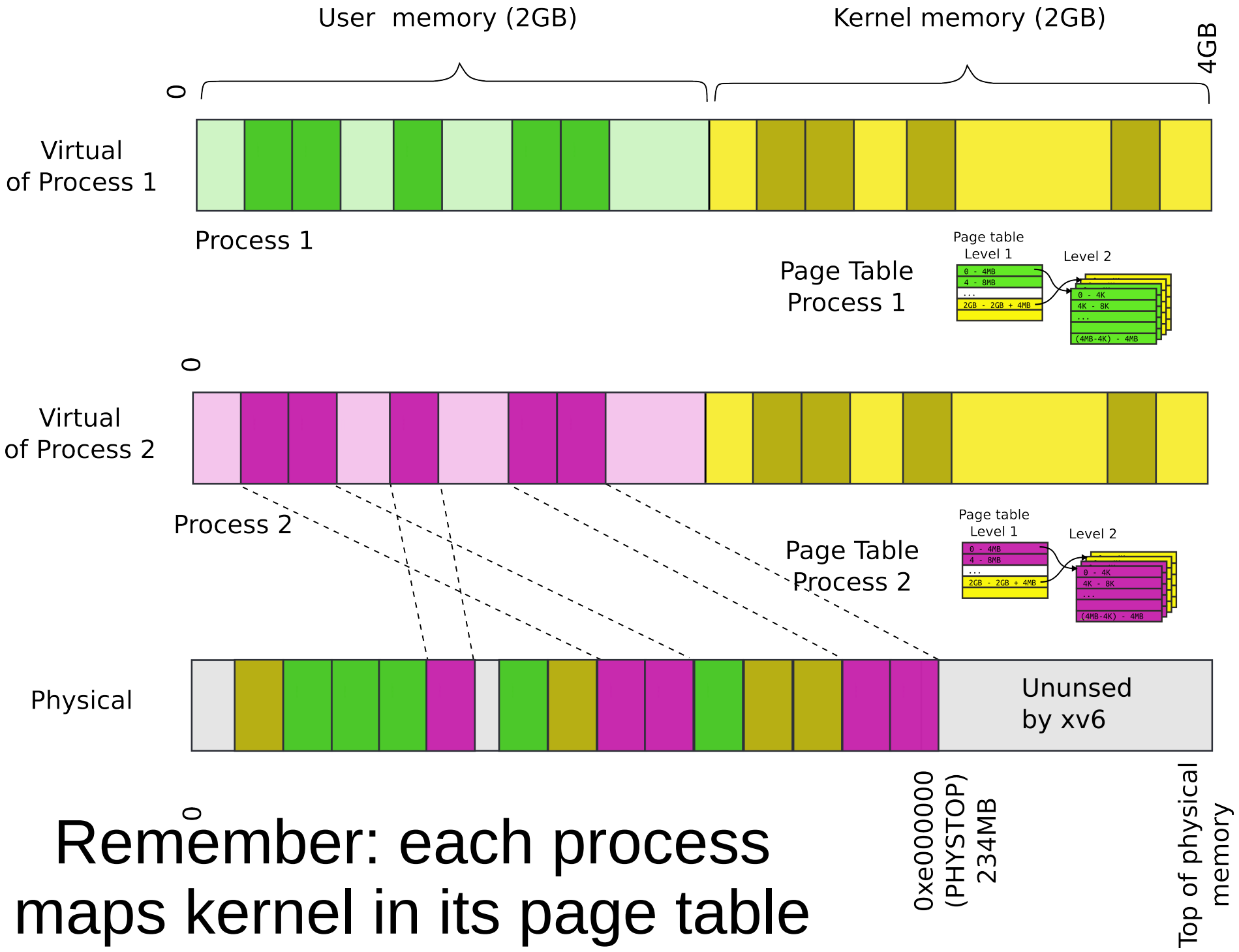
```
2103 struct proc {
2104   uint sz; // Size of process memory (bytes)
2105   pde_t* pgdir; // Page table
2106   char *kstack; // Bottom of kernel stack for this process
2107   enum procstate state; // Process state
2108   volatile int pid; // Process ID
2109   struct proc *parent; // Parent process
2110   struct trapframe *tf; // Trap frame for current syscall
2111   struct context *context; // swtch() here to run
2112   void *chan; // If non-zero, sleeping on chan
2113   int killed; // If non-zero, have been killed
2114   struct file *ofile[NOFILE]; // Open files
2115   struct inode *cwd; // Current directory
2116   char name[16]; // Process name (debugging)
2117 };
```

Userinit() – create first process

- Allocate process structure
  - Information about the process
- **Create a page table**
  - **Map only kernel space**

```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[],
                  _binary_initcode_size[];
...
2509   p = allocproc();
2510   initproc = p;
2511    if((p->pgdir = setupkvm()) == 0)
2512      panic("userinit: out of memory?");
2513   inituvm(p->pgdir, _binary_initcode_start,
              (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```

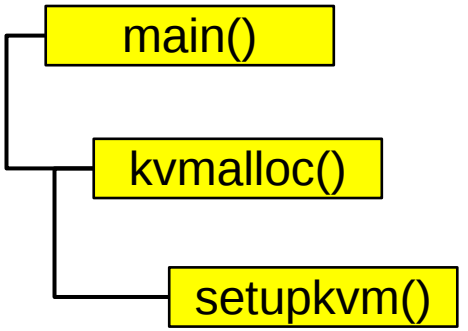




Remember: each process maps kernel in its page table

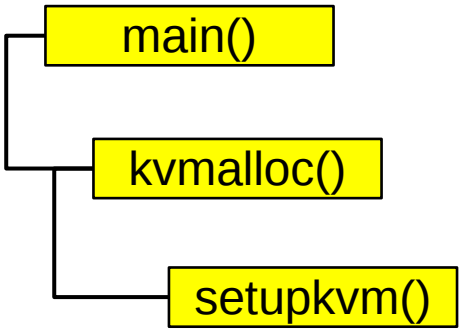
Recap: Allocate page table directory

```
1836 pde_t*
1837 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```



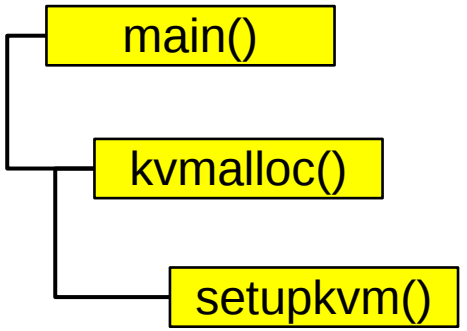
Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```

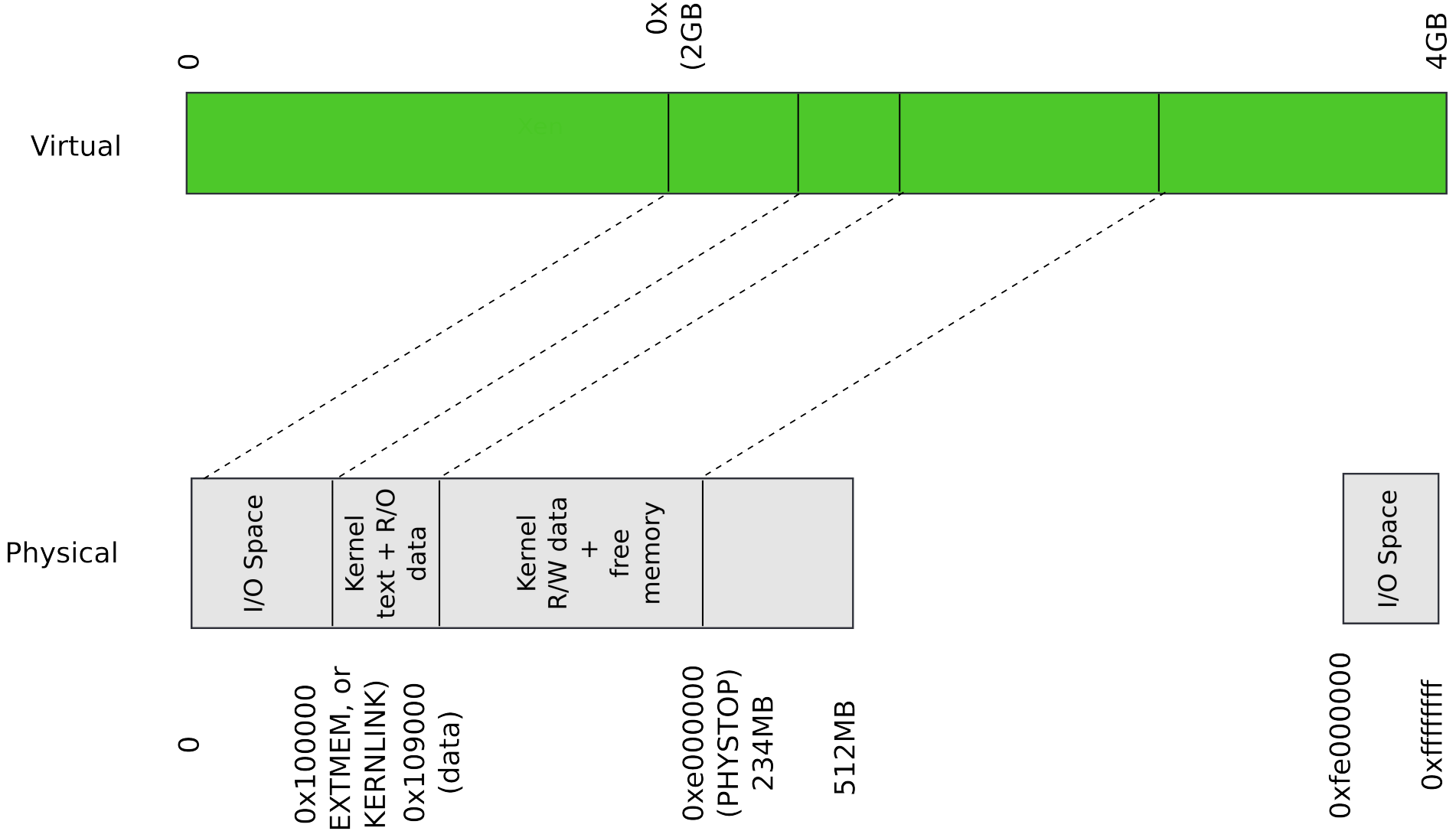


Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```

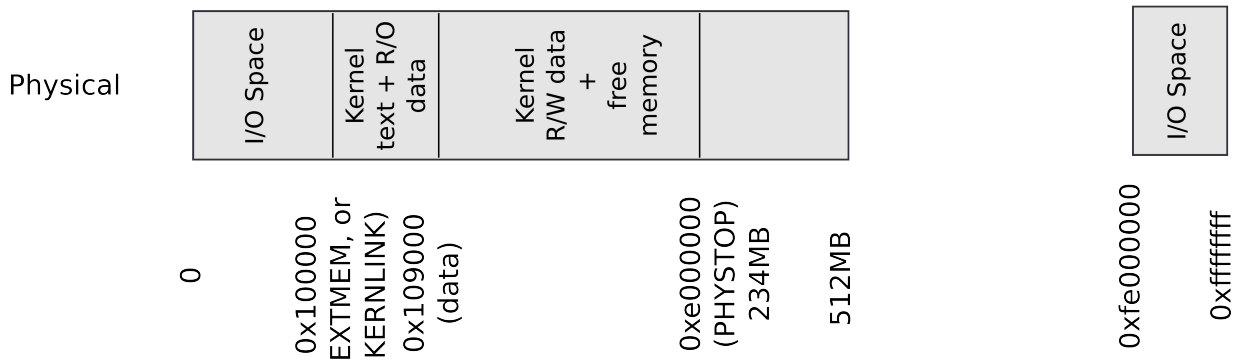


Recap: Kernel map



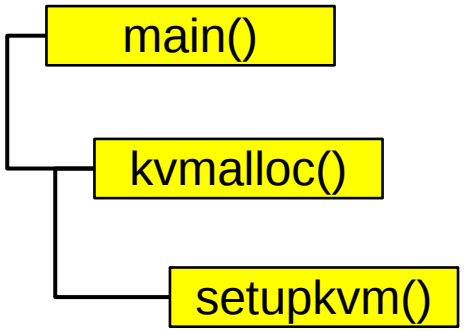
Recap: Kmap – kernel map

```
1823 static struct kmap {
1824     void *virt;
1825     uint phys_start;
1826     uint phys_end;
1827     int perm;
1828 } kmap[] = {
1829     { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
1830     { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},//text+rodata
1831     { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern
data+memory
1832     { (void*)DEVSPACE, DEVSPACE, 0, PTE_W}, // more devices
1833 };
```



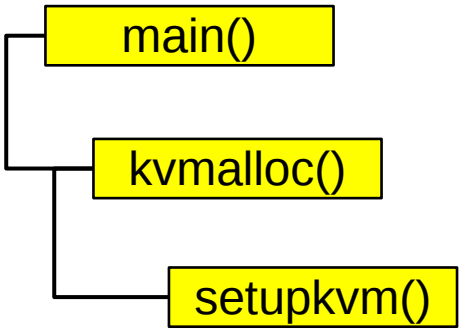
Recap: Iterate in a loop: remap physical pages

```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```



Recap: Remap physical pages

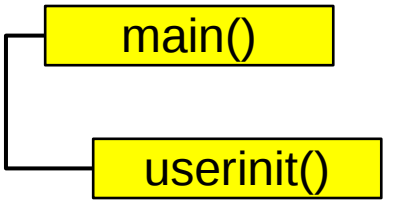
```
1836 pde_t*
1887 setupkvm(void)
1838 {
1839   pde_t *pgdir;
1840   struct kmap *k;
1841
1842   if((pgdir = (pde_t*)kalloc()) == 0)
1843     return 0;
1844   memset(pgdir, 0, PGSIZE);
...
1847   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1848     if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1849                 (uint)k->phys_start, k->perm) < 0)
1850       return 0;
1851   return pgdir;
1852 }
```



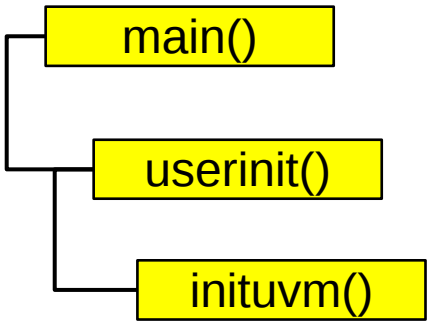
Userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- **Allocate a page for the user init code**
  - **Map this page**

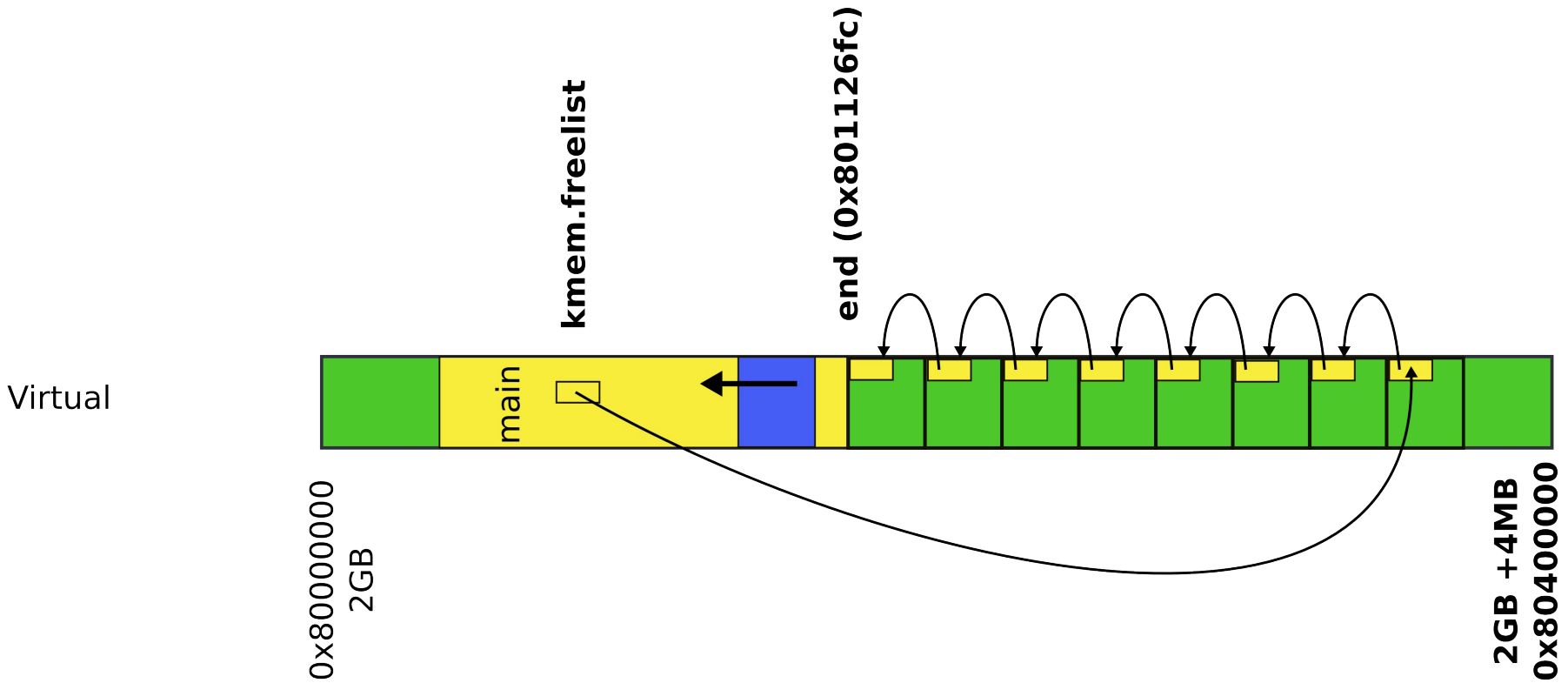
```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[],
                  _binary_initcode_size[];
...
2509   p = allocproc();
2510   initproc = p;
2511   if((p->pgdir = setupkvm()) == 0)
2512     panic("userinit: out of memory?");
2513   inituvm(p->pgdir, _binary_initcode_start,
                (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```

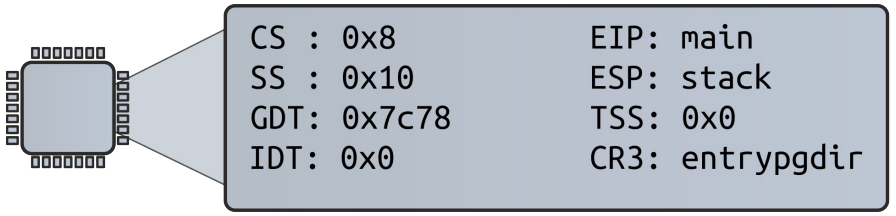


```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, V2P(mem),
                PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
```



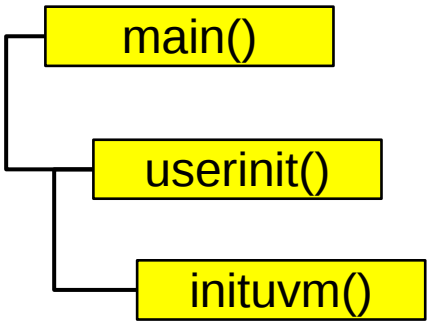
Recap: kalloc() – allocate page



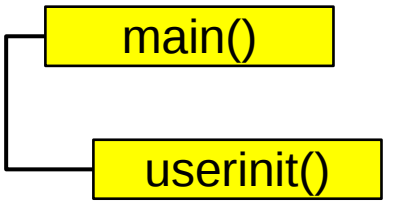


Protected Mode

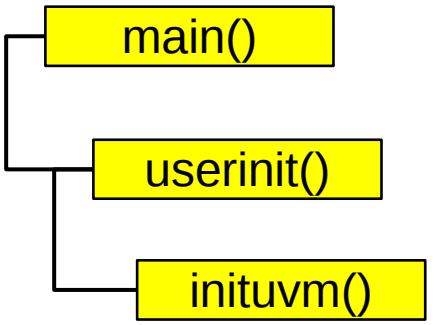
```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, V2P(mem),
                PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
```



```
2502 userinit(void)
2503 {
2504   struct proc *p;
2505   extern char _binary_initcode_start[],
                  _binary_initcode_size[];
...
2509   p = allocproc();
2510   initproc = p;
2511   if((p->pgdir = setupkvm()) == 0)
2512     panic("userinit: out of memory?");
2513   inituvm(p->pgdir, _binary_initcode_start,
                (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
...
2530 }
```



```
1903 inituvm(pde_t *pgdir, char *init, uint sz)
1904 {
1905   char *mem;
1906
1907   if(sz >= PGSIZE)
1908     panic("inituvm: more than a page");
1909   mem = kalloc();
1910   memset(mem, 0, PGSIZE);
1911   mappages(pgdir, 0, PGSIZE, V2P(mem),
                PTE_W|PTE_U);
1912   memmove(mem, init, sz);
1913 }
```



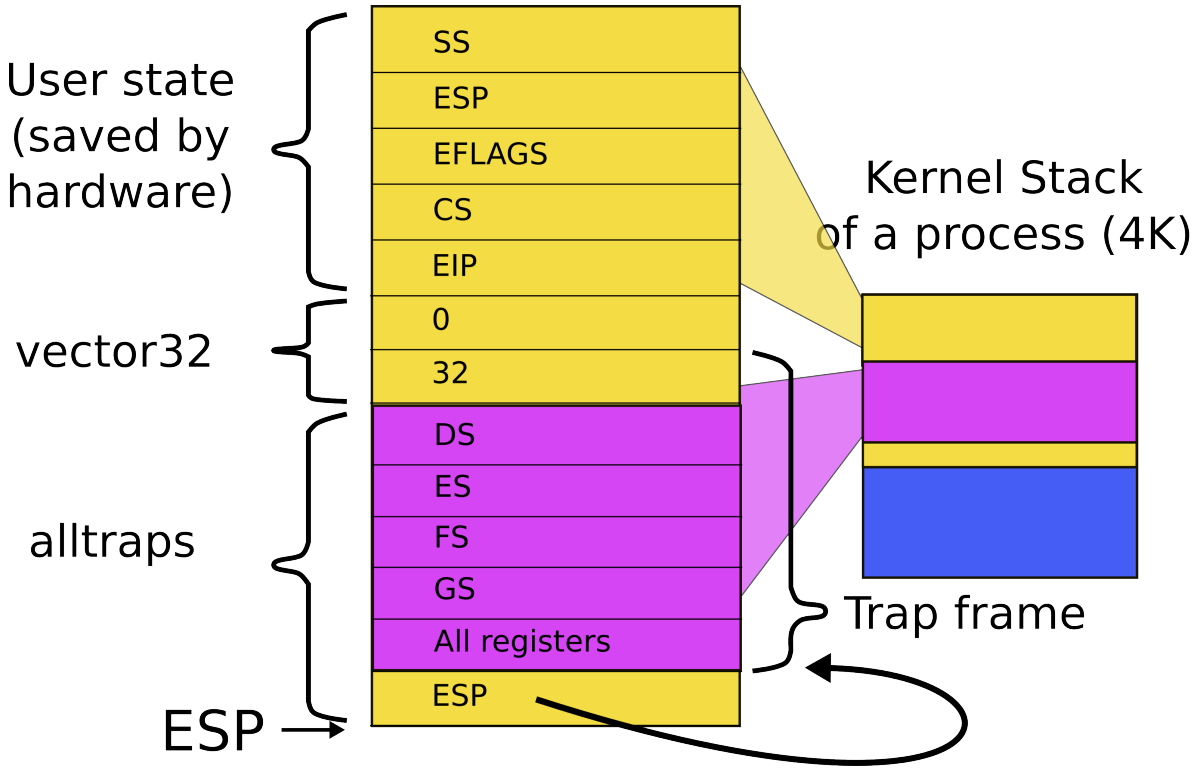
```
8409 start:
8410   pushl $argv
8411   pushl $init
8412   pushl $0 // where caller pc would be
8413   movl $SYS_exec, %eax
8414   int $T_SYSCALL
8415
...
8422 # char init[] = "/init\0";
8423 init:
8424   .string "/init\0"
8425
8426 # char *argv[] = { init, 0 };
8427 .p2align 2
8428 argv:
8429   .long init
8430   .long 0
```

initcode.S: call exec(“/init”, argv);

userinit() – create first process

- Allocate process structure
  - Information about the process
- Create a page table
  - Map only kernel space
- Allocate a page for the user init code
  - Map this page
- **Configure trap frame for “iret”**

Kernel stack after interrupt/syscall



Call stack: vector32()
alltraps()

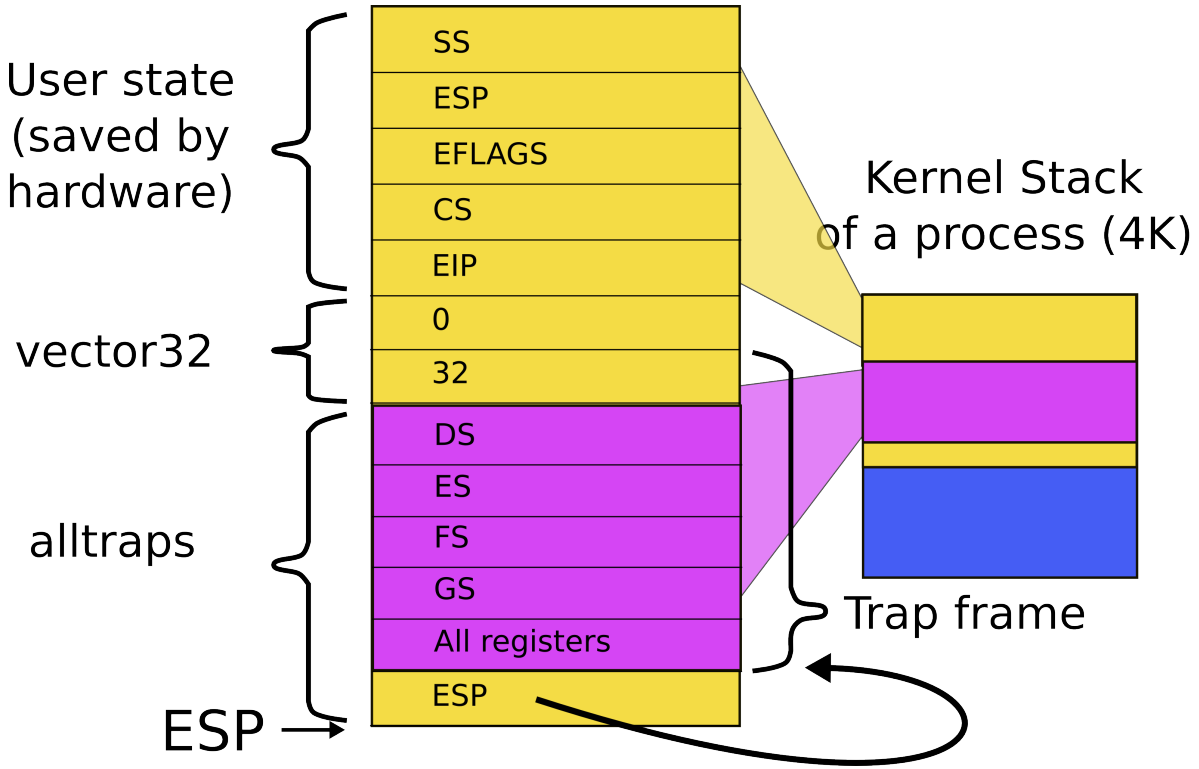
```
2103 struct proc {
2104   uint sz; // Size of process memory (bytes)
2105   pde_t* pgdir; // Page table
2106   char *kstack; // Bottom of kernel stack for this process
2107   enum procstate state; // Process state
2108   volatile int pid; // Process ID
2109   struct proc *parent; // Parent process
2110   struct trapframe *tf; // Trap frame
2111   struct context *context; // swtch() here to run
2112   void *chan; // If non-zero, sleeping on chan
2113   int killed; // If non-zero, have been killed
2114   struct file *ofile[NOFILE]; // Open files
2115   struct inode *cwd; // Current directory
2116   char name[16]; // Process name (debugging)
2117 };
```

```
2456 allocproc(void)
2457 {
...
2470   // Allocate kernel stack.
2471   if((p->kstack = kalloc()) == 0){
2472     p->state = UNUSED;
2473     return 0;
2474   }
2475   sp = p->kstack + KSTACKSIZE;
2476
2477   // Leave room for trap frame.
2478   sp -= sizeof *p->tf;
2479   p->tf = (struct trapframe*)sp;
2480
...
2492 }
```

Trap frame is on the kernel stack of the process

```
2502 userinit(void)
2503 {
...
2513   inituvm(p->pgdir, _binary_initcode_start,
                (int)_binary_initcode_size);
2514   p->sz = PGSIZE;
2515   memset(p->tf, 0, sizeof(*p->tf));
2516   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2517   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2518   p->tf->es = p->tf->ds;
2519   p->tf->ss = p->tf->ds;
2520   p->tf->eflags = FL_IF;
2521   p->tf->esp = PGSIZE;
2522   p->tf->eip = 0; // beginning of initcode.S
...
2530 }
```

Kernel stack after interrupt/syscall

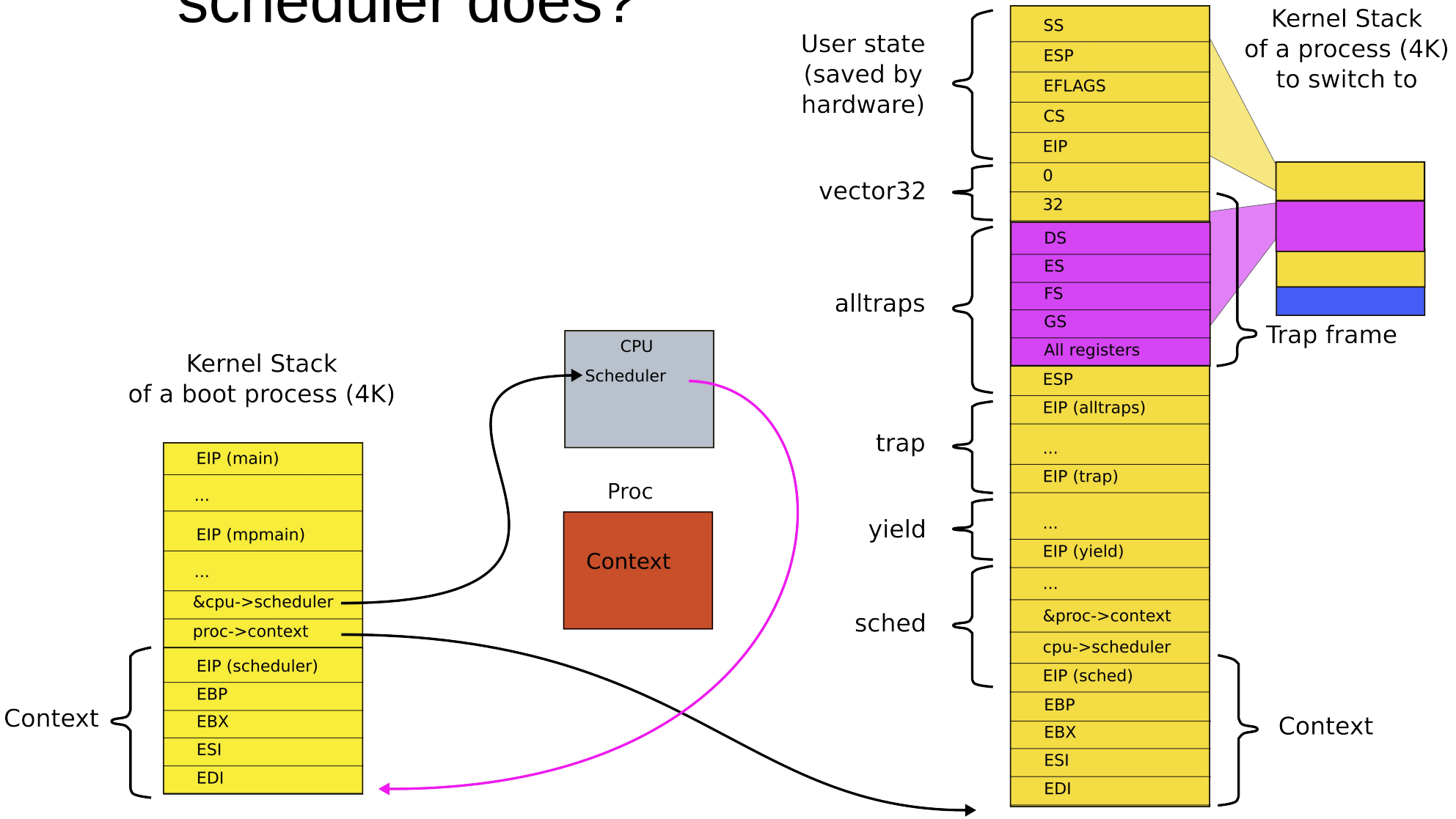


Call stack: vector32()
alltraps()

```
2502 userinit(void)
2503 {
...
2515   memset(p->tf, 0, sizeof(*p->tf));
2516   p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
2517   p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
2518   p->tf->es = p->tf->ds;
2519   p->tf->ss = p->tf->ds;
2520   p->tf->eflags = FL_IF;
2521   p->tf->esp = PGSIZE;
2522   p->tf->eip = 0; // beginning of initcode.S
2523
2524   safestrcpy(p->name, "initcode", sizeof(p->name));
2525   p->cwd = namei("/");
2526
2527   p->state = RUNNABLE;
...
2530 }
```

Wait, we mapped process memory, created trap frame, but it doesn't really run...

Remember what scheduler does?



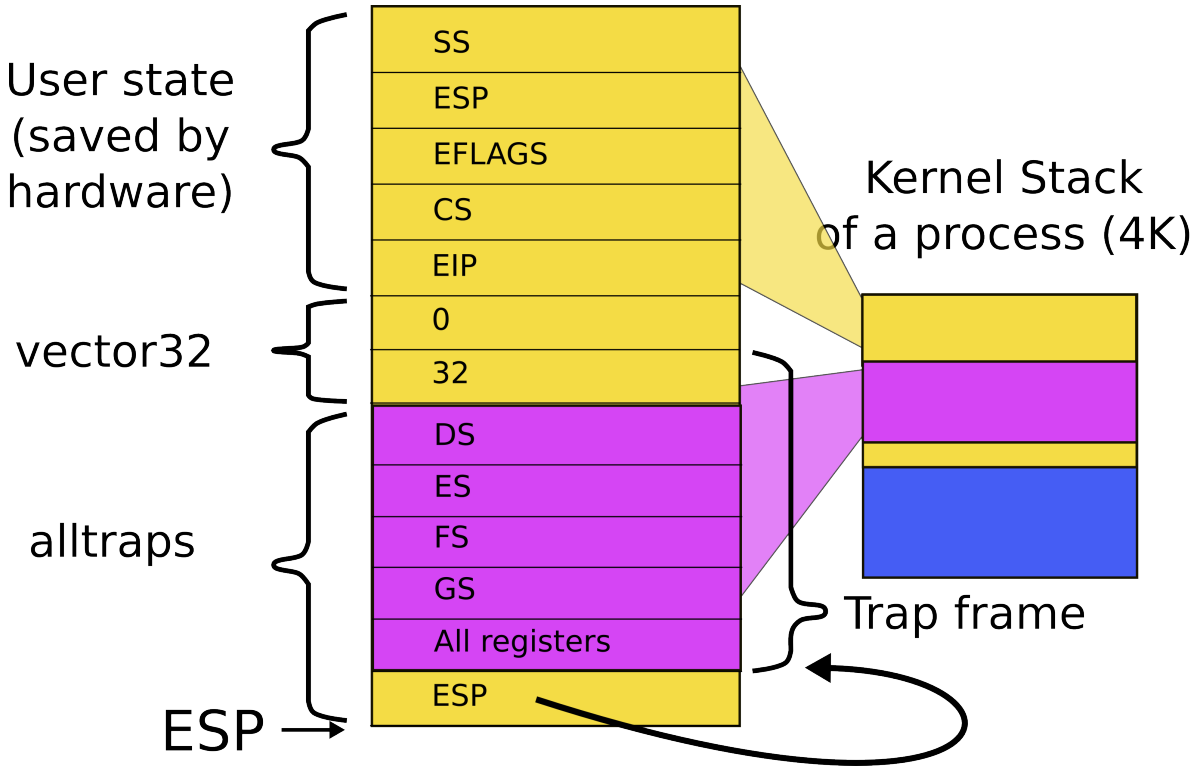
```
2456 allocproc(void)
2457 {
...
2477   // Leave room for trap frame.
2478   sp -= sizeof *p->tf;
2479   p->tf = (struct trapframe*)sp;
2480
2481   // Set up new context to start executing at forkret,
2482   // which returns to trapret.
2483   sp -= 4;
2484   *(uint*)sp = (uint)trapret;
2485
2486   sp -= sizeof *p->context;
2487   p->context = (struct context*)sp;
2488   memset(p->context, 0, sizeof *p->context);
2489   p->context->eip = (uint)forkret;
...
2492 }
```

Trap frame is on the kernel stack of the process

forkret(): just returns

```
2788 forkret(void)
2789 {
...
2803   // Return to "caller",
         actually trapret (see
         allocproc).
2804 }
```

Kernel stack after interrupt/syscall



Call stack: vector32()
alltraps()

```
3276 .globl trapret
3277 trapret:
3278   popal
3279   popl %gs
3280   popl %fs
3281   popl %es
3282   popl %ds
3283   addl $0x8, %esp # trapno and
                       errcode
3284   iret
```

forkret(): just returns

/init starts /sh

```
8510 main(void)
8511 {
...
8514   if(open("console", O_RDWR) < 0){
8515     mknod("console", 1, 1);
8516     open("console", O_RDWR);
8517   }
8518   dup(0); // stdout
8519   dup(0); // stderr
8520
8521   for(;;){
8522     printf(1, "init: starting sh\n");
8523     pid = fork();
8524     if(pid < 0){
8525       printf(1, "init: fork failed\n");
8526       exit();
8527     }
8528     if(pid == 0){
8529       exec("sh", argv);
8530       printf(1, "init: exec sh failed\n");
8531       exit();
8532     }
8533     while((wpid=wait()) >= 0 && wpid != pid)
8534       printf(1, "zombie!\n");
8535   }
8536 }
```

Back to the first system call: exec()

Summary

- We've finally learned how the first process came to life
- We're half way into system calls

Thank you

sys_exec()

```
6225 sys_exec(void)
6226 {
6227   char *path, *argv[MAXARG];
6228   int i;
6229   uint uargv, uarg;
6230
6231   if(argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0){
6232     return -1;
6233   }
6234   memset(argv, 0, sizeof(argv));
6235   for(i=0;; i++){
6236     if(i >= NELEM(argv))
6237       return -1;
6238     if(fetchint(uargv+4*i, (int*)&uarg) < 0)
6239       return -1;
6240     if(uarg == 0){
6241       argv[i] = 0;
6242       break;
6243     }
6244     if(fetchstr(uarg, &argv[i]) < 0)
6245       return -1;
6246   }
6247   return exec(path, argv);
6248 }
```

```
6225 sys_exec(void)
6226 {
6227   char *path, *argv[MAXARG];
6228   int i;
6229   uint uargv, uarg;
6230
6231   if(argstr(0, &path) < 0
         || argint(1, (int*)&uargv) < 0){
6232     return -1;
6233   }
...
6247   return exec(path, argv);
6248 }
```

sys_exec()