EE488: Unix Kernel Design
Fall 2022

Booting

Youjip Won

KAIST EE

Start the kernel.

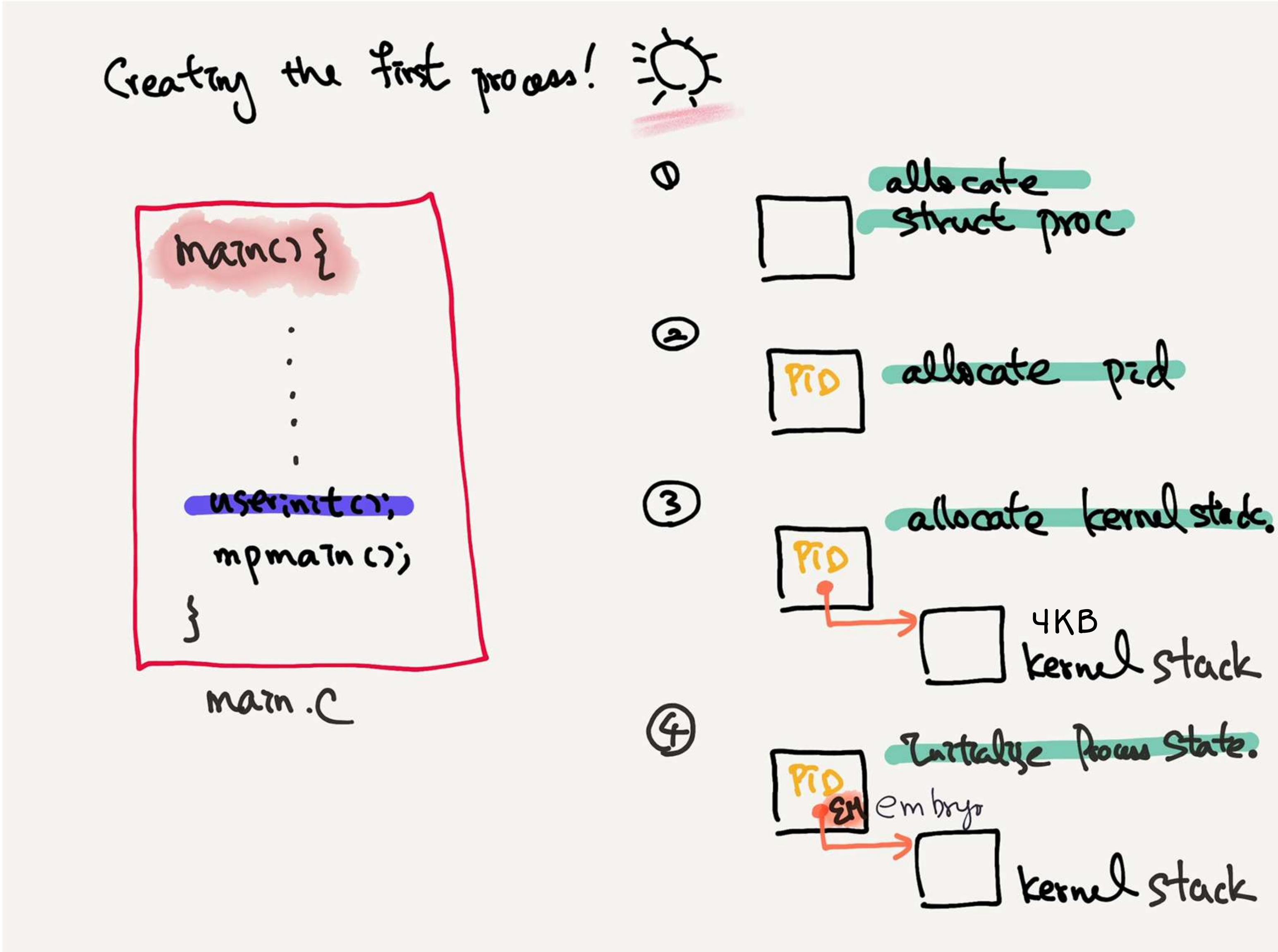
main.c

```
int main(void){
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();        // kernel page table
  mpinit();          // detect other processors
  lapicinit();       // interrupt controller
  seginit();         // segment descriptors
  picinit();         // disable pic
  ioapicinit();      // another interrupt controller
  consoleinit();     // console hardware
  uartinit();        // serial port
  pinit();           // process table
  tvinit();          // trap vectors
  binit();           // buffer cache
  fileinit();        // file table
  ideinit();         // disk
  startothers();     // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();        // first user process
  mpmain();          // finish this processor's setup
}
```

Creating the first process

- Allocate process structure.
  - Scan the proc table in the kernel address space and find the free entry.
- Initialize the process structure.
  - Set the state of the proc structure to EMBRYO.
  - Allocate process ID.
  - Allocate kernel stack.

Creating the first process



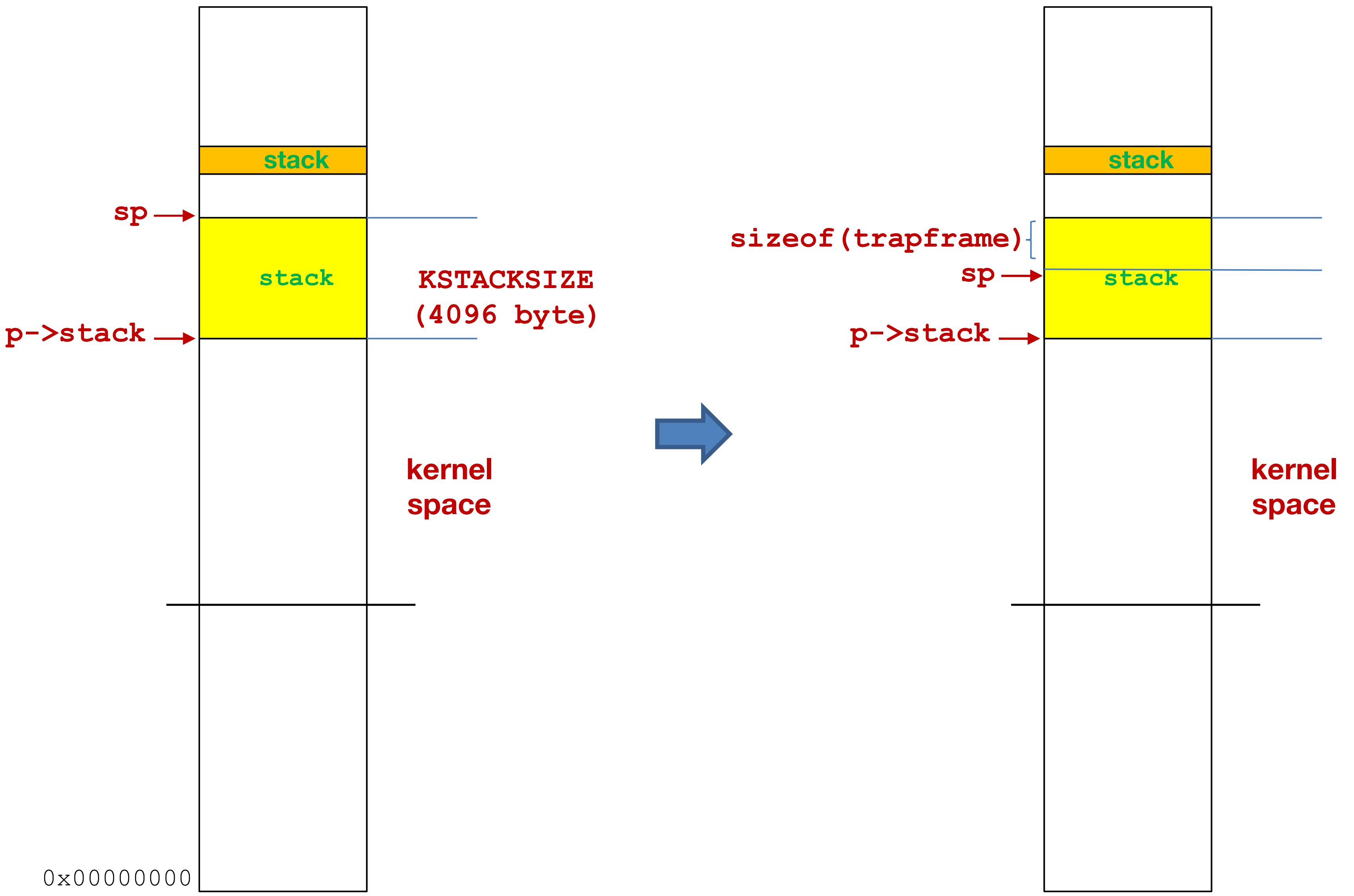
Creating the first process

- Allocate process structure.
  - Scan the proc table in the kernel address space and find the free entry.
- Initialize the process structure.
  - Set the state of the proc structure to EMBRYO.
  - Allocate process ID.
  - Allocate kernel stack.

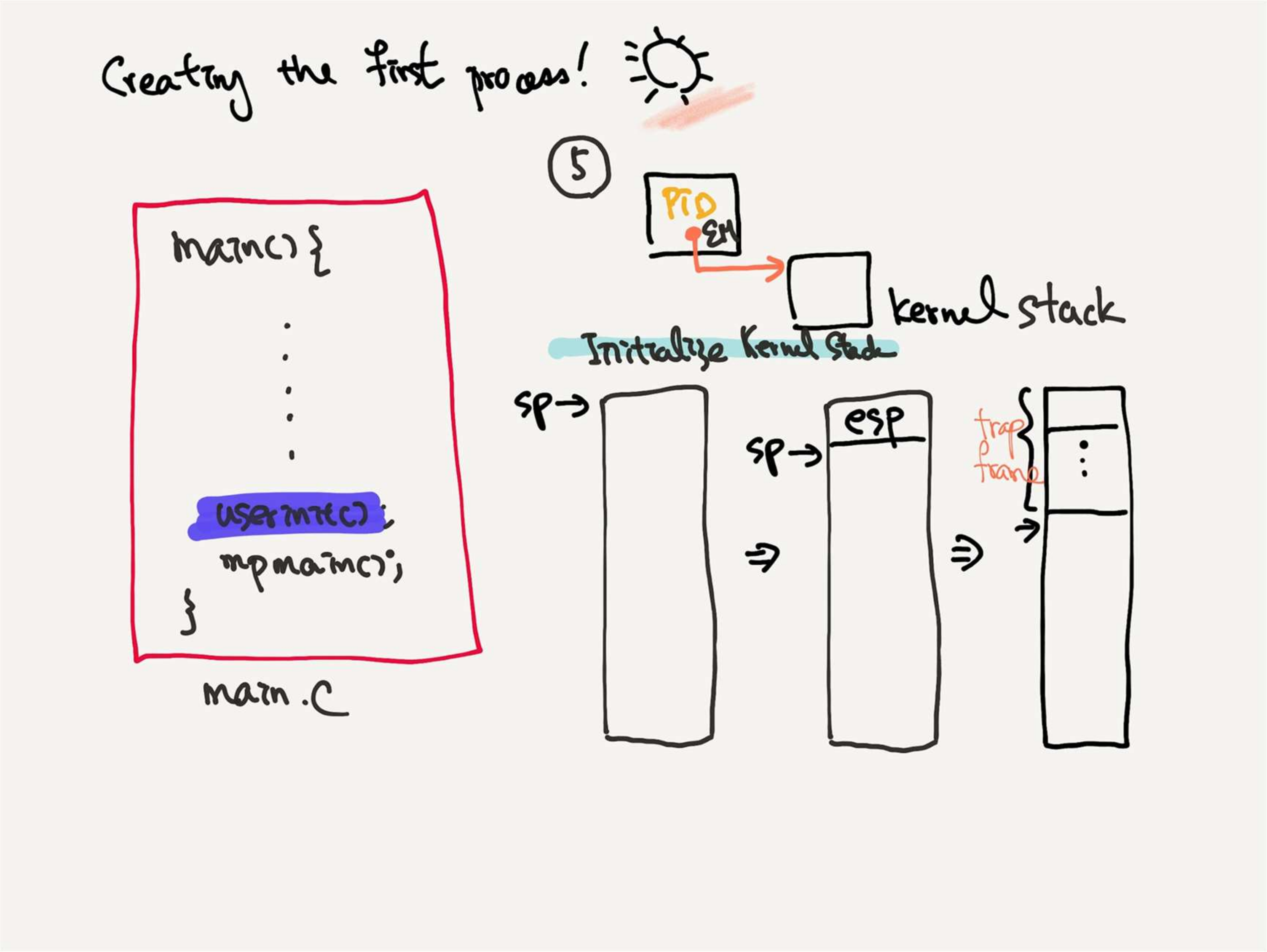
```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == UNUSED)
      goto found;
found:
  p->state = EMBRYO;
  p->pid = nextpid++;

  if((p->kstack = kalloc()) == 0){
  sp = p->kstack + KSTACKSIZE;
```

allocproc()

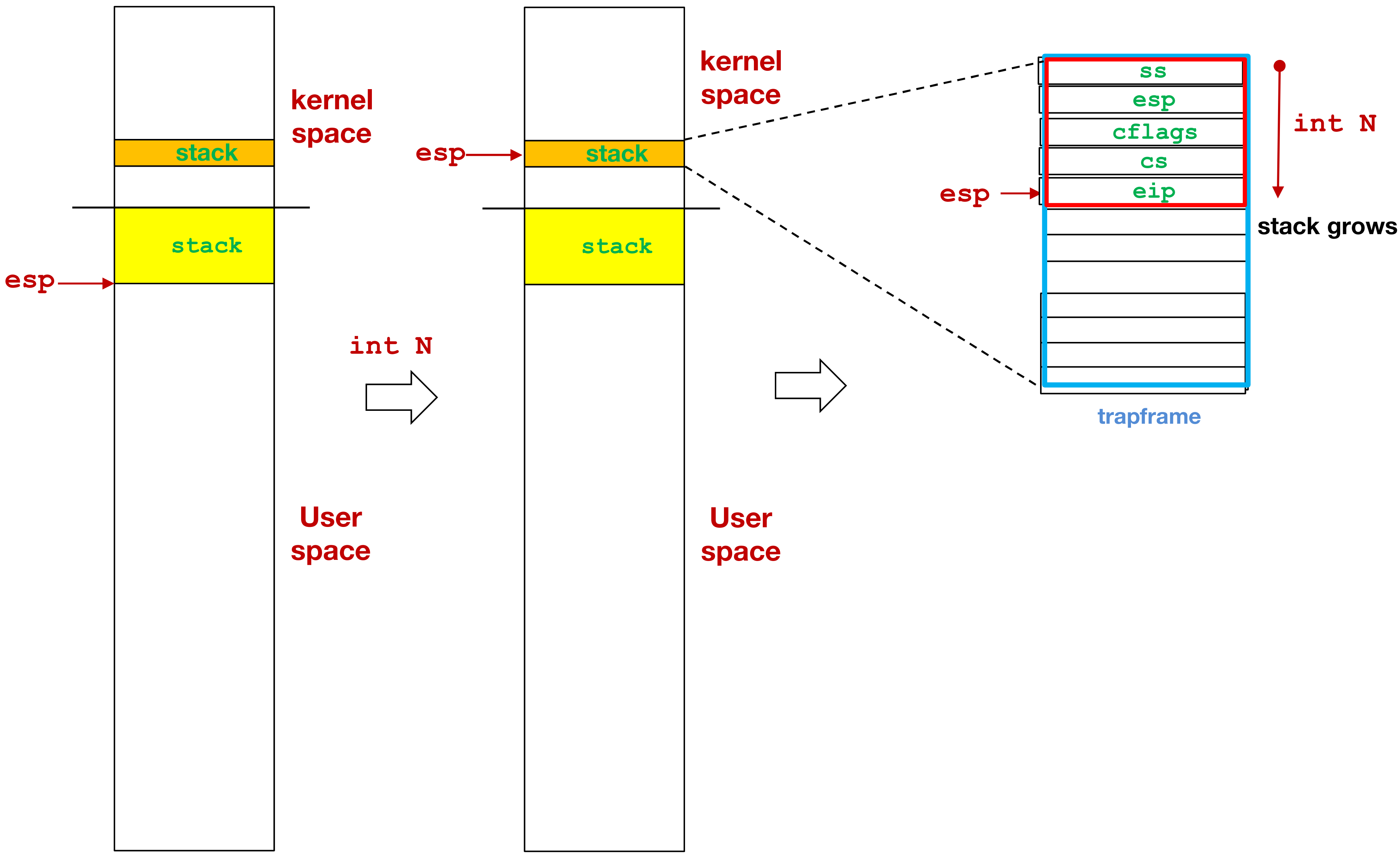
Allocating stack for the first process



Creating the first process (Cont.)



Getting into and out of kernel



struct trapframe

```
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
  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;
};
```

← `trapframe`: a data structure that stores the registers of the user process

← It is in the kernel stack.

Stack grows.

struct trapframe

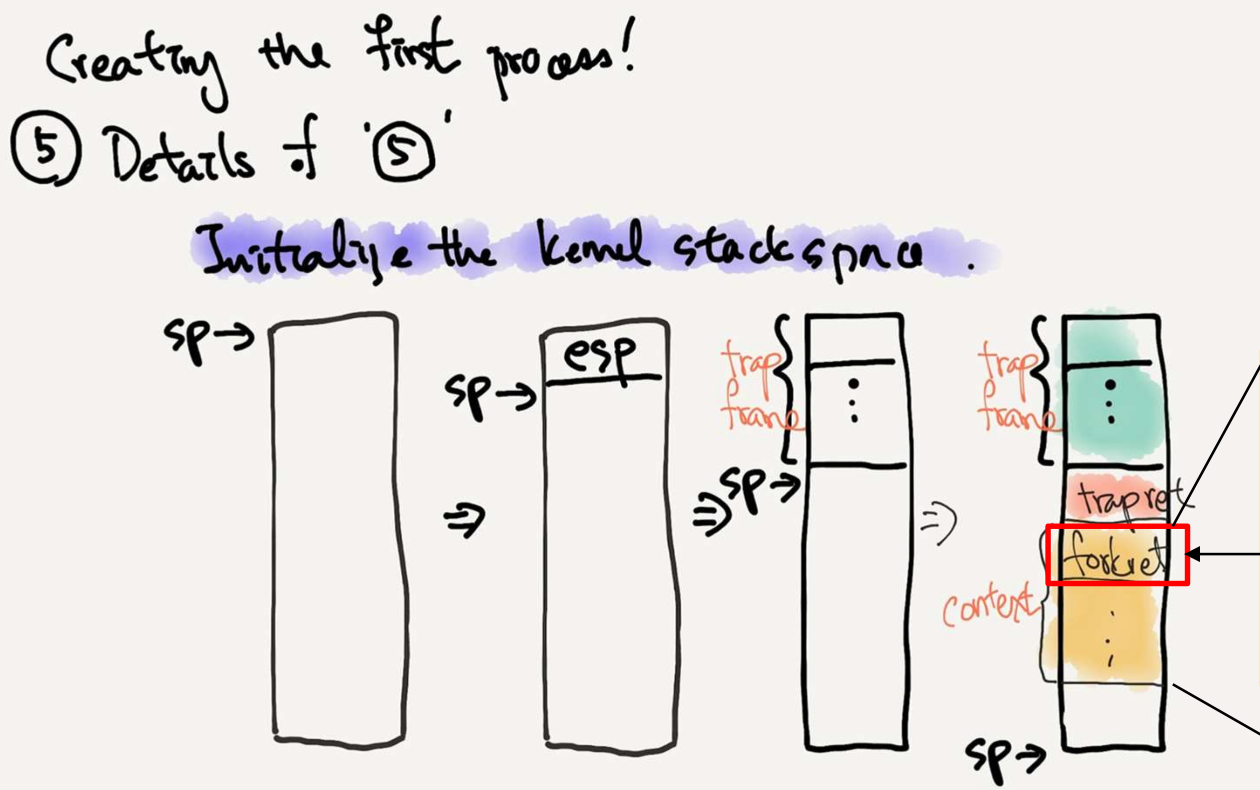
```
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  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;
};
```

Stack grows.

Pushed and Popped by Software (`alltraps`,`trapret`)

Pushed and Popped by Hardware

Creating the first process (Cont.)

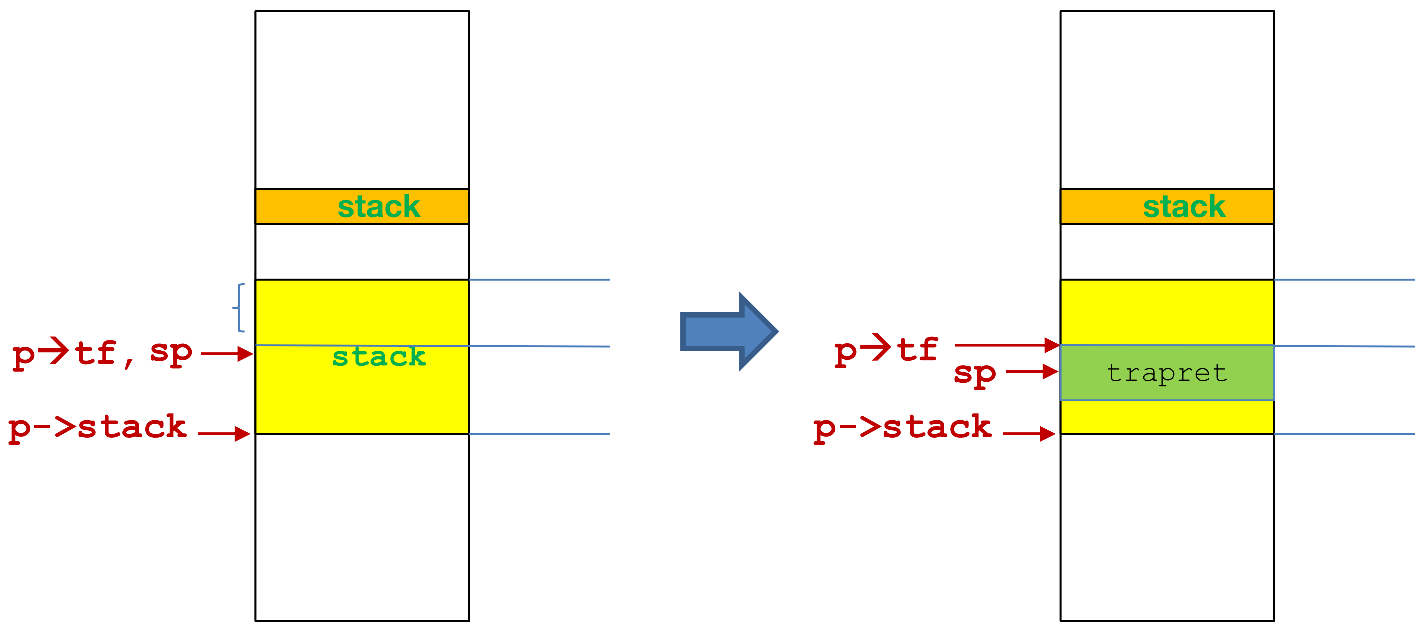



```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

1. Allocate a space for `struct trapframe`.
2. Store the function to execute to pop the frame contents.
3. Allocate a space for `struct context`.
4. Set the `%eip` to execute to `function forkret`.

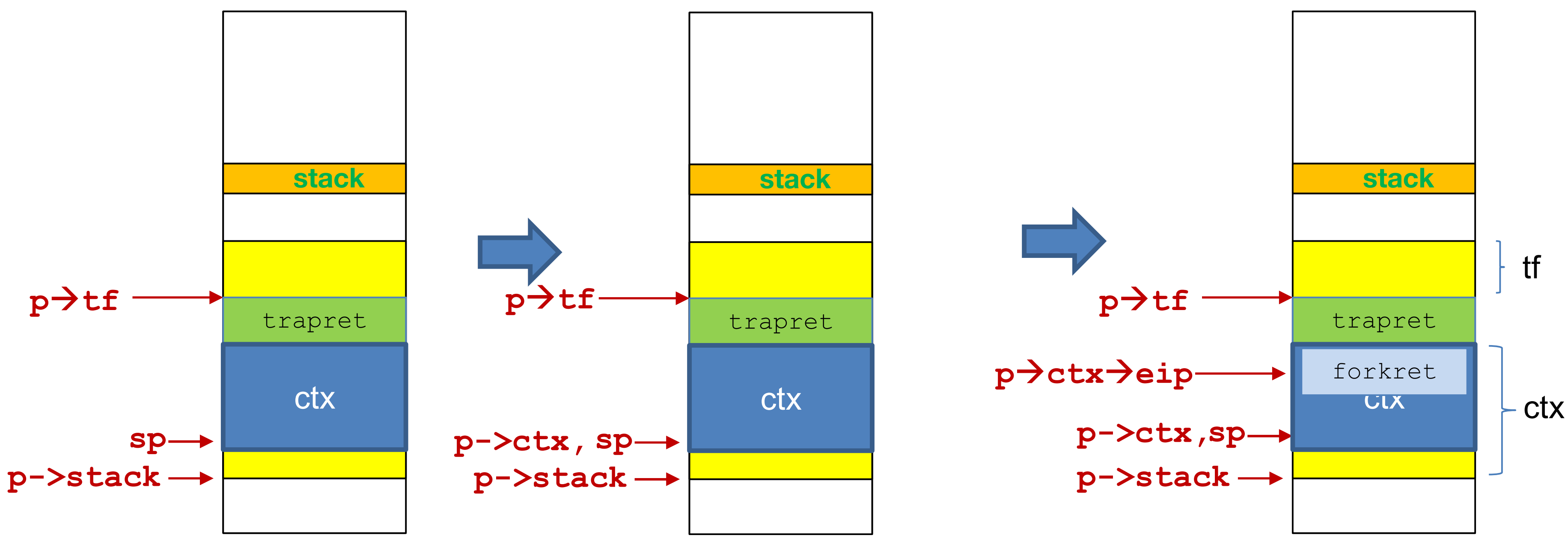
Allocating stack for the first process

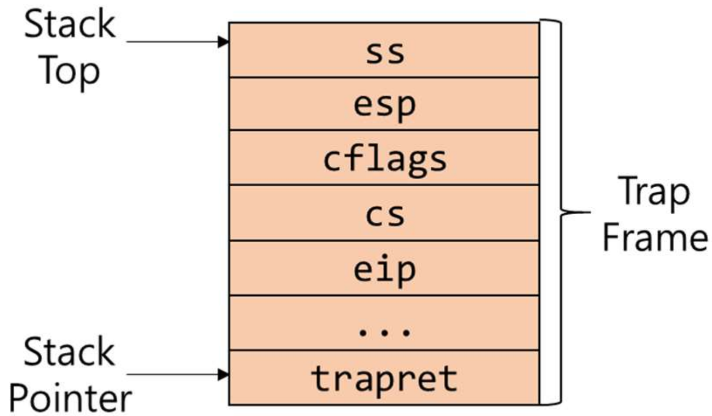
1. Allocate a space for `struct trapframe`.
2. Store the function to execute to pop the frame contents.
3. Allocate a space for `struct context`.
4. Set the `%eip` to execute to `function forkret`.



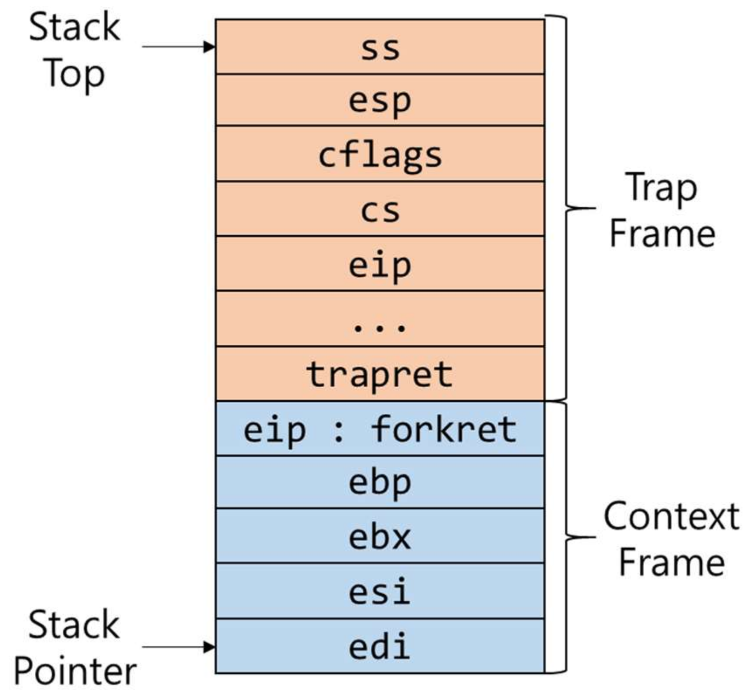
Allocating stack for the first process

1. Allocate a space for `struct trapframe`.
2. Store the function to execute to pop the frame contents.
3. Allocate a space for `struct context`.
4. Set the `%eip` to execute to `function forkret`.





(a) Line 102 ~ 108



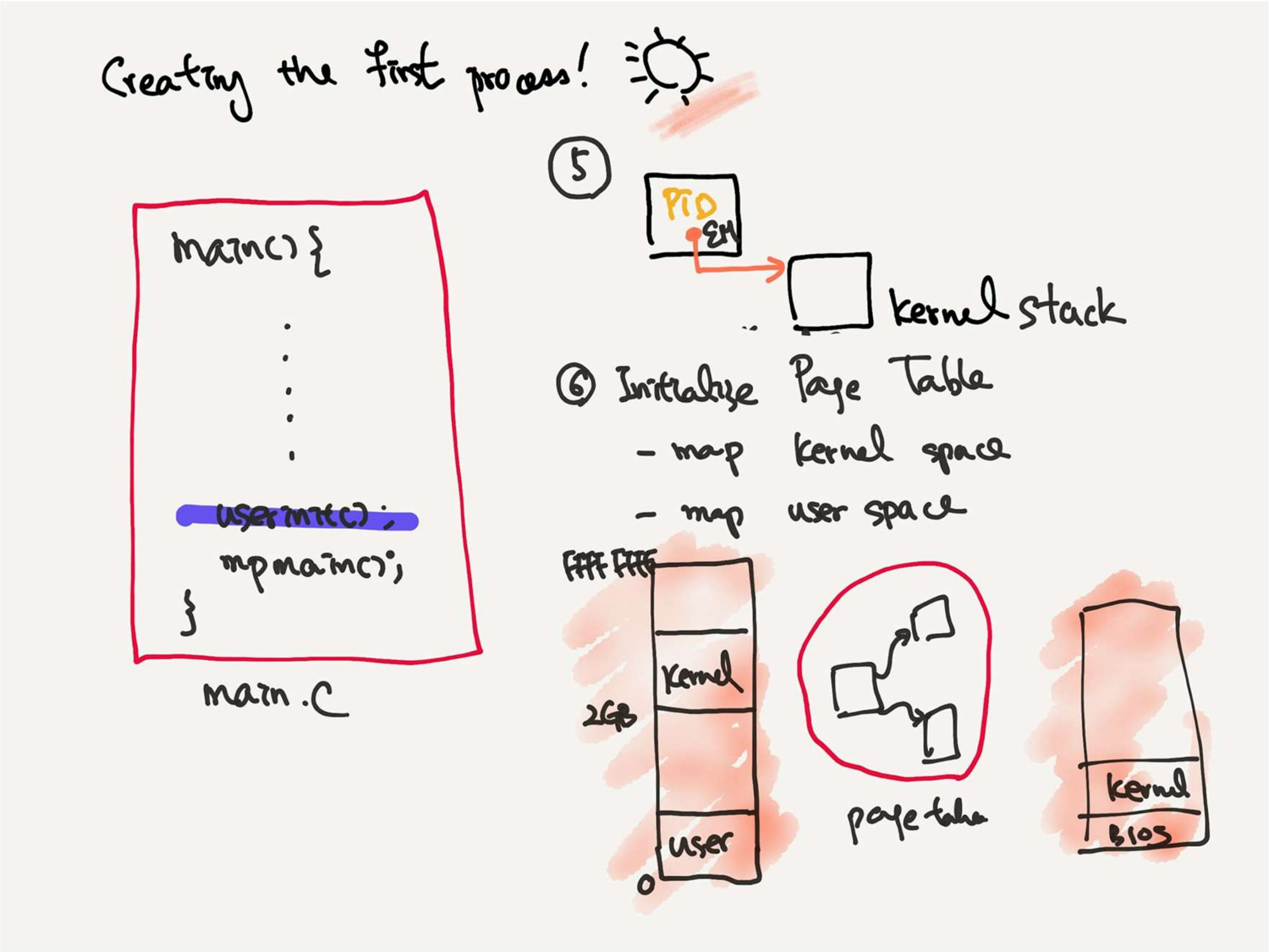
(b) Line 110 ~ 113

Creating the first process

1. Allocate a space for `struct trapframe`.
2. Store the function to execute to pop the frame contents.
3. Allocate a space for `struct context`.
4. Set the `%eip` to execute to `function forkret`.

```
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;
```

Creating the first process (Cont.)



Initialize address space

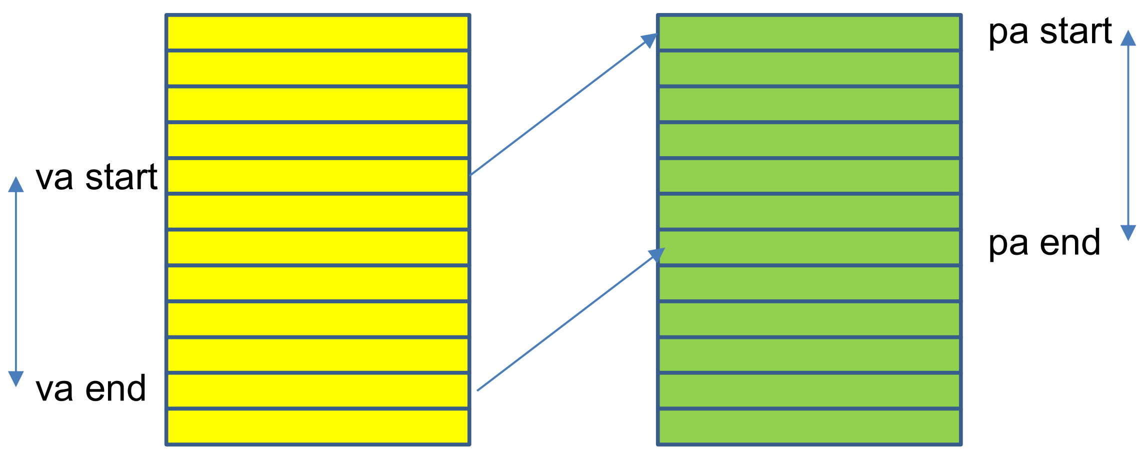
- Setup Page Table for `init` process (`setupkvm()`)
  - Map Virtual to Physical address as ELF binary describes.
- Load `init` binary to memory (`inituvm()`).

```
void
userinit(void)
{

//deleted for space saving

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);

// deleted for space saving

}
```

Initialize address space

- Setup Page Table for `init` process (`setupkvm()`)
  - Set up page table entry according to `kmap` structure described at ELF binary.

```
pde_t *pgdir; struct kmap *k;
if((pgdir = (pde_t*)kalloc()) == 0)
for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
  if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
return pgdir;
```



Initialize address space

- Setup Page Table for `init` process (`setupkvm()`)
  - Set up page table entry according to `kmap` structure described at ELF binary.

```
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
};
```

Creating the first process (Cont.)

Creating the first process!

```
main(){
    .
    .
    .
    .
    .
    userinit();
    mpmain();
}
```

main.c

⑦ copy initcode.S binary to the user address space.

⑧ fill up the values into the trap frame.

⑨ Set the state of the process RUNNABLE.!

Now, we are ready to run the first process.

Load the code for `init` and set up trap frame

```
void
userinit(void)
{

//deleted for space saving

  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
// allocate one page to pgdir 0, and copy initcode.S.
  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

 // deleted for space saving
}
```

Set the process state to runnable

```
void
userinit(void)
{

   //deleted for space saving

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);
 }
}
```

Running the first process

Running the first process!

- how to switch page table
- how to switch context
- how to get out of the kernel and start executing the user process.

```
main() {
    ⋮
    userinit();
    mpmain();
}
```

main.c

0. schedule();
1. exec("init.S");
2. init.S call exec("sh");

Running the first process (Cont.)

Running the first process!

```
main() {
    .
    .
    .
    userinit();
    mpmain();
}
```

main.c

① ~~schedule();~~

- look for a process with RUNNABLE.
- There is only one : initcode.
- change page table (switchuvm). to target process.
- change state to RUNNING.
- change context (swtch).

⇒ kernel starts executing 'forkret'.

Think why!

Running the first process (Cont.)

1. Find a RUNNABLE process
2. Change Page Table for target process
3. Change Context

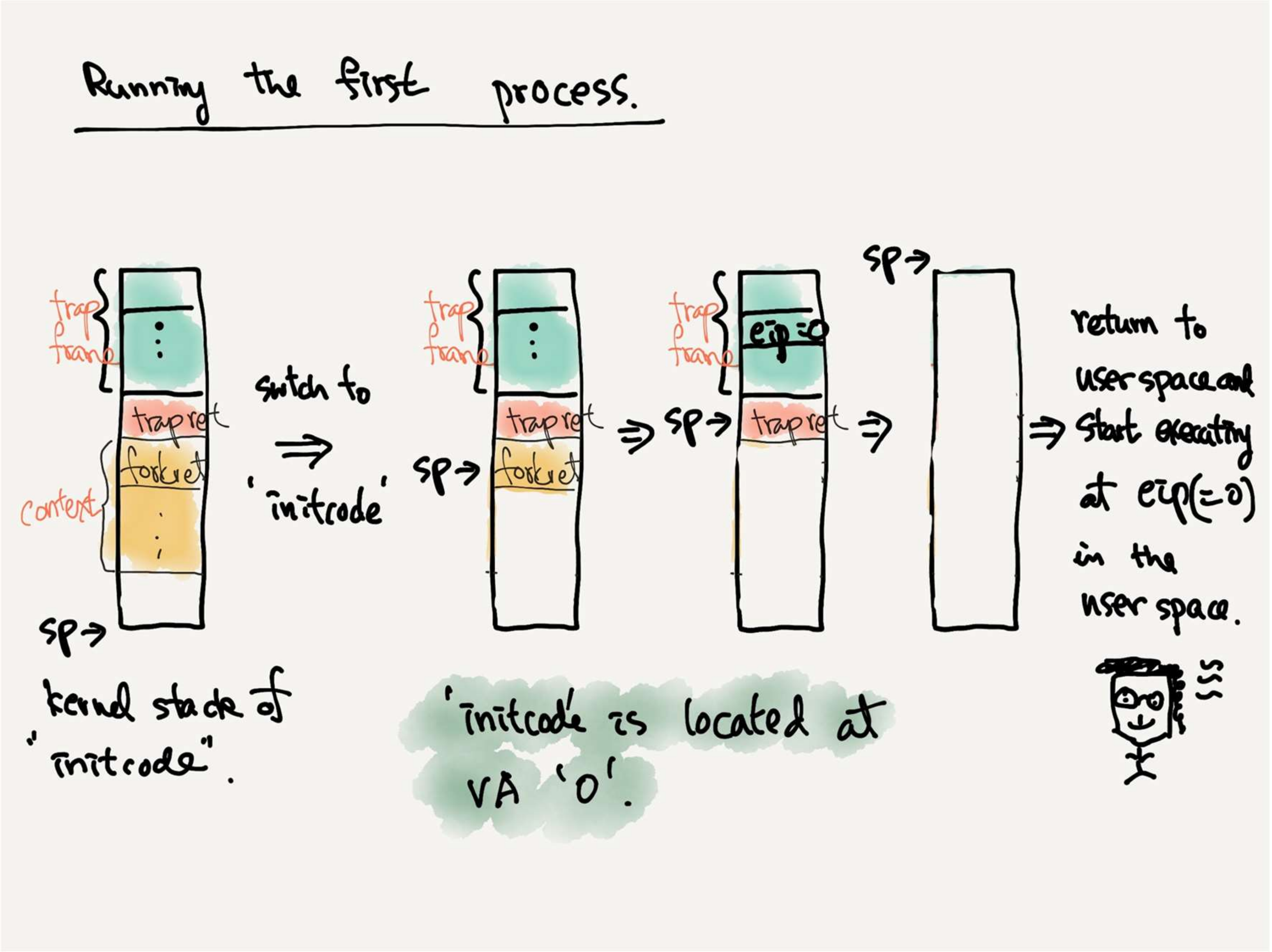
```
 static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();        // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();      // start running processes
}
```

Running the first process (Cont.)

1. Find a RUNNABLE process
2. Change Page Table for target process
3. Change Context

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

  c->proc = p;
  switchuvm(p);
  p->state = RUNNING;

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

Running the first process (Cont.)



Running the first process (Cont.)

- Switch context from current one to the new one.

```
swtch:
  movl 4(%esp), %eax /* &(c->scheduler) */
  movl 8(%esp), %edx /* p->context  */

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

  # Switch stacks
  movl %esp, (%eax) /* Save Current Stack Pointer */
  movl %edx, %esp   /* Switch Stack */

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret                /* Control Flow to forkret */
```

/init

What does 'initcode' do?

It executes '/init' binary.

```
# Initial process execs /init.
# This code runs in user space.

#include "syscall.h"
#include "traps.h"


# exec(init, argv)
.globl start
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

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

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

'/init' is mother of all processes in Unix!

/init (Cont.)

What does 'initcode' do?

The first system call : exec

- page table : done
- process structure : done
- Now : let's exec
  - replaces memory and registers
  - keep process id, proc structure parent process and file descriptors

`/init` (Cont.)

Executing '/init'

① push parameters

- $argv
- $init
- $0

goes to user stack

② setup system call #.

- %eax ← SYSCALL_EXEC

③ trap

- T_SYSCALL

exec never returns!

init.c

```
char *argv[] = { "sh", 0 };

int main(void){
  int pid, wpid;

 // open the console device if necessary and
 // set it as file descriptor 0,1,2.

  if(open("console", O_RDWR) < 0){
    mknod("console", 1, 1);
    open("console", O_RDWR);
  }
  dup(0);   // stdout
  dup(0);   // stderr

  // to be continued on the next page.
```

`/init` (Cont.)

In '/init',

- open consde device.
- open fd 0,1 and 2.
- fork and exec 'shell'.

Now, the system is up!!!

init.c

```
// from the previous page

  for(;;){
    printf(1, "init: starting sh\n");
    pid = fork();
    if(pid < 0){
      printf(1, "init: fork failed\n");
      exit();
    }
    if(pid == 0){ // child process runs shell.
      exec("sh", argv);
      printf(1, "init: exec sh failed\n");
      exit();
    }
    while((wpid=wait()) >= 0 && wpid != pid)  // handled orphaned zombies.
      printf(1, "zombie!\n");
  }
}
```

Summary

- Now we are done with booting.
- Booting contains everything.
  - Address mode
  - Process creation
  - Context switch
  - Mode switch
  - Scheduling
- We will deal with these one by one.

appendix

Quiz: about booting

- Sample quiz
  - In the course of booting, the program counter starts to execute in the lower address space and then at some point switches to the kernel region. Locate the instruction where $eip is changed to the kernel address space.
  - entry() sets up the initial page table with 4MByte page size before it calls main(). It initializes the first entry and 512^th entry. What happens if entry() does not initialize the first entry in the page table? Modify the source code. Try to boot it. Examine what happens.
  - Please provide the cause about what happens to booting.

Assembler directives

`.code16`	create 16 bit code from now on.
`.code32`	create 32 bit code from now on.
`.global [symbol]`	make the [symbol] visible outside.
`.word [value]`	create a memory space for [word] and set its value to [value]. For 16 bit address, 16 bit space is allocated. for 32 bit address, 32bit address is created.
`.long [value]`	create 32bit data.
`.string [str]`	create character string.
`.text`	the following segment belongs to text section.
`.data`	the following segment belongs to data section.
`.comm [symbol], [length]`	In bss section, create the space with [length] size and [symbol] name.

what is function entry point?

test.c

```
#include <stdio.h>

int add(int p1, int p2)
{
   return p1+p2;
}

int main(int argc, char *argv[])
{
   printf("Please type two numbers\n");

   int a, b;
   scanf("%d %d", &a, &b);

   int res = add(a, b);
   printf("Result of adding : %d\n", res);
}
```

entry point

- compile the file for debugging purpose
  - `$ gcc -g -o test test.c`
- run GDB
  - `$ gdb test`
  - `(gdb) disas main`
  - `(gdb) disas add`

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000400601 <+0>:     push   %rbp
   0x0000000000400602 <+1>:     mov    %rsp,%rbp
   0x0000000000400605 <+4>:     sub    $0x20,%rsp
   0x0000000000400609 <+8>:     mov    %edi,-0x14(%rbp)
   0x000000000040060c <+11>:    mov    %rsi,-0x20(%rbp)
   0x0000000000400610 <+15>:    mov    $0x4006e4,%edi
   0x0000000000400615 <+20>:    callq  0x4004b0 <puts@plt>
   0x000000000040061a <+25>:    lea    -0x8(%rbp),%rdx
   0x000000000040061e <+29>:    lea    -0xc(%rbp),%rax
   0x0000000000400622 <+33>:    mov    %rax,%rsi
   0x0000000000400625 <+36>:    mov    $0x4006fc,%edi
   0x000000000040062a <+41>:    mov    $0x0,%eax
   0x000000000040062f <+46>:    callq  0x4004f0 <__isoc99_scanf@plt>
   0x0000000000400634 <+51>:    mov    -0x8(%rbp),%edx
   0x0000000000400637 <+54>:    mov    -0xc(%rbp),%eax
   0x000000000040063a <+57>:    mov    %edx,%esi
   0x000000000040063c <+59>:    mov    %eax,%edi
   0x000000000040063e <+61>:    callq  0x4005ed <add>
   0x0000000000400643 <+66>:    mov    %eax,-0x4(%rbp)
   0x0000000000400646 <+69>:    mov    -0x4(%rbp),%eax
   0x0000000000400649 <+72>:    mov    %eax,%esi
   0x000000000040064b <+74>:    mov    $0x400702,%edi
   0x0000000000400650 <+79>:    mov    $0x0,%eax
   0x0000000000400655 <+84>:    callq  0x4004c0 <printf@plt>
   0x000000000040065a <+89>:    leaveq
   0x000000000040065b <+90>:    retq
End of assembler dump.
```

```
(gdb) disas add
Dump of assembler code for function add:
   0x00000000004005ed <+0>:     push   %rbp
   0x00000000004005ee <+1>:     mov    %rsp,%rbp
   0x00000000004005f1 <+4>:     mov    %edi,-0x4(%rbp)
   0x00000000004005f4 <+7>:     mov    %esi,-0x8(%rbp)
   0x00000000004005f7 <+10>:    mov    -0x8(%rbp),%eax
   0x00000000004005fa <+13>:    mov    -0x4(%rbp),%edx
   0x00000000004005fd <+16>:    add    %edx,%eax
   0x00000000004005ff <+18>:    pop    %rbp
   0x0000000000400600 <+19>:    retq
End of assembler dump.
```

finding the address of a variable

- run GDB
  - `$ gdb test`
  - `(gdb) b add`
  - `(gdb) run`
  - `(gdb) p &p1`

```
Breakpoint 1, add (p1=5, p2=3) at test.c:5
5               return p1+p2;
(gdb) p &p1
$1 = (int *) 0x7fffffffe3ac
(gdb) p &p2
$2 = (int *) 0x7fffffffe3a8
```

Summary

- overview of booting

- creating the first address space
- creating the first process
- running the first process

- do not forget: preview and review