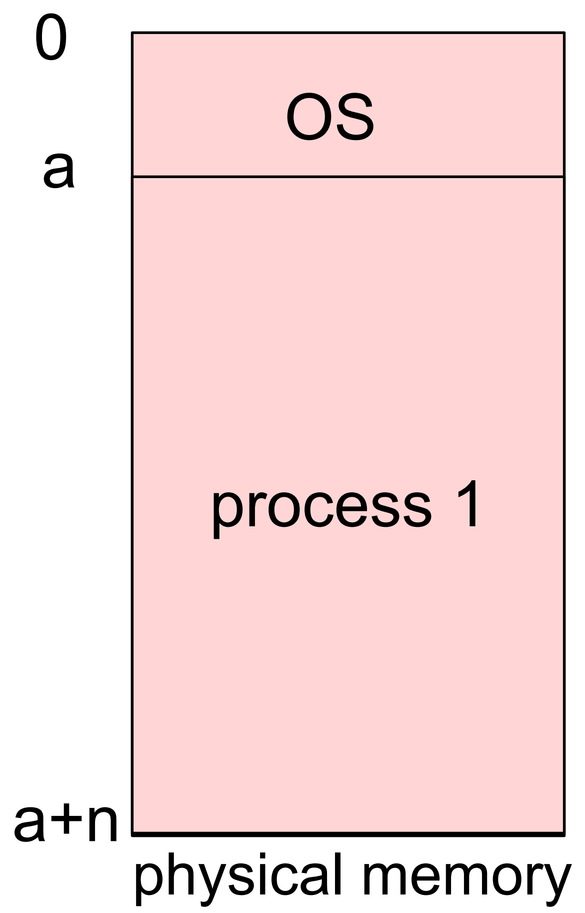
CS 134
Operating Systems

Feb 11, 2019

Virtual Memory

Memory

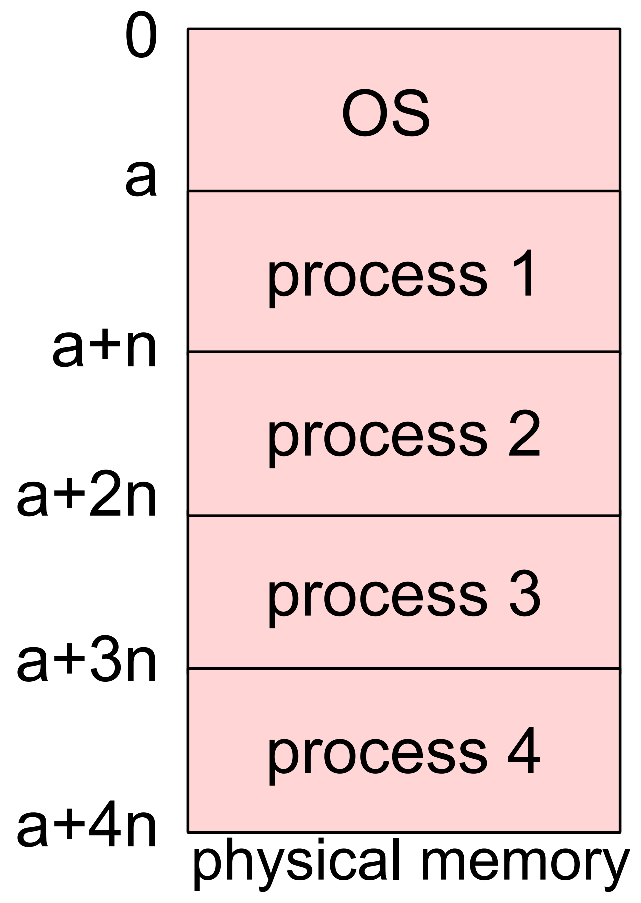
- For multiprogramming, must have multiple processes
- Each process must be in memory to execute
- Possibilities:
  - One process in memory: ***Swapping***: Swap it out to disk, swap in a new one



Memory

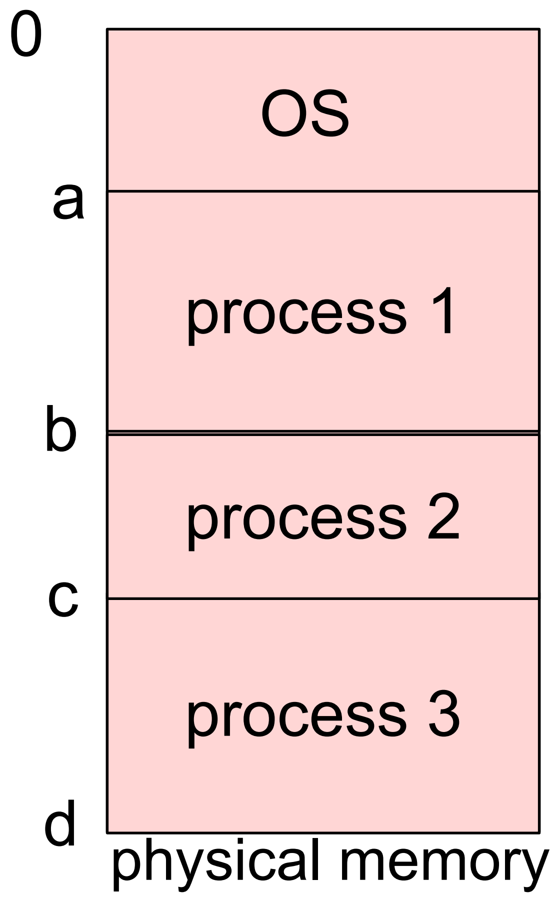
- For multiprogramming, must have multiple processes
- Each process must be in memory to execute
- Possibilities:
  - Multiple processes in memory: each in their own partition
    - Fixed-size equal partitions

Internal fragmentation: fragmented memory *within* allocated memory blocks



Memory

- For multiprogramming, must have multiple processes
- Each process must be in memory to execute
- Possibilities:
  - Multiple processes in memory: each in their own partition
    - Fixed-size *non*-equal partitions



External fragmentation: fragmented memory *between* allocated memory blocks

Address binding is the process of mapping from one address space to another address space.
Address binding is just address translation: bind physical address to each virtual/logical address.
Address binding can happen at different period:
Check https://www.geeksforgeeks.org/mapping-virtual-addresses-to-physical-addresses/

Address Binding

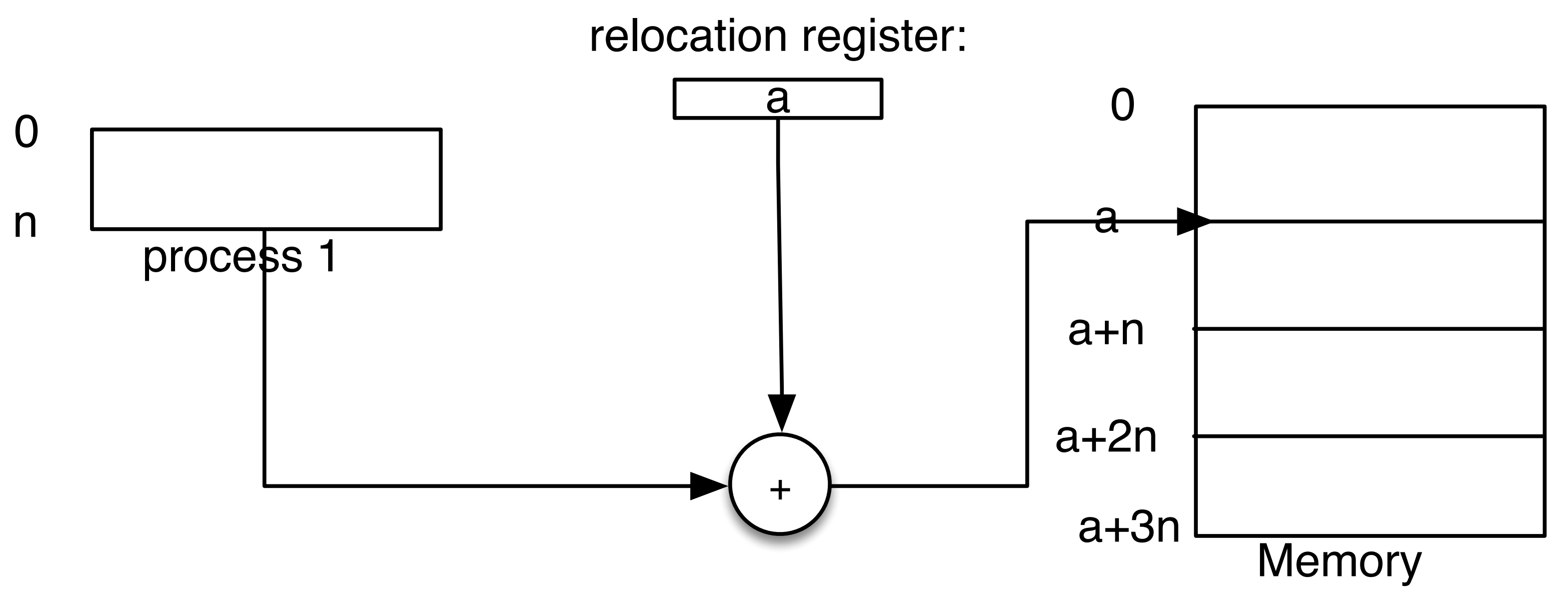
- When to map instructions and data references to actual memory locations
  - Link time

xv6, JOS, ELF format.

    - Absolute addressing. Works if well-known address at which user programs are loaded
  - Load time
    - Linker-loader relocates code as it loads into memory
    - Change all data references to take into account where loaded
    - Position-independent code
    - For PIE (position independent executables), subroutine calls are PC-relative. Static data references are indirected through a pointer to data location

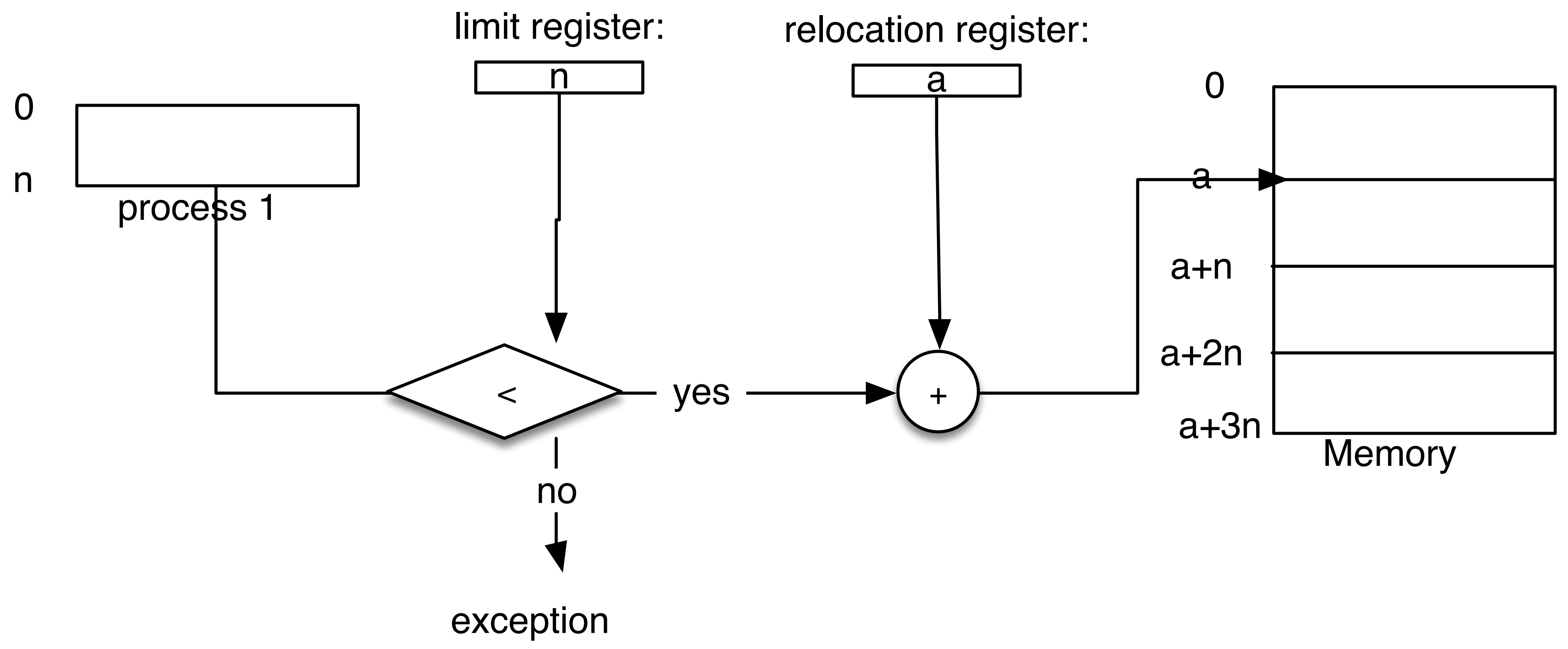
Address Binding

- When to map instructions and data references to actual memory locations
  - Execution time
    - Code uses *logical addresses* which are converted to *physical addresses*
    - Hardware: relocation register



Process Protection

- How to protect the memory of a process (or the OS) from other running processes?
  - Hardware solution: base (*aka* relocation) and limit registers

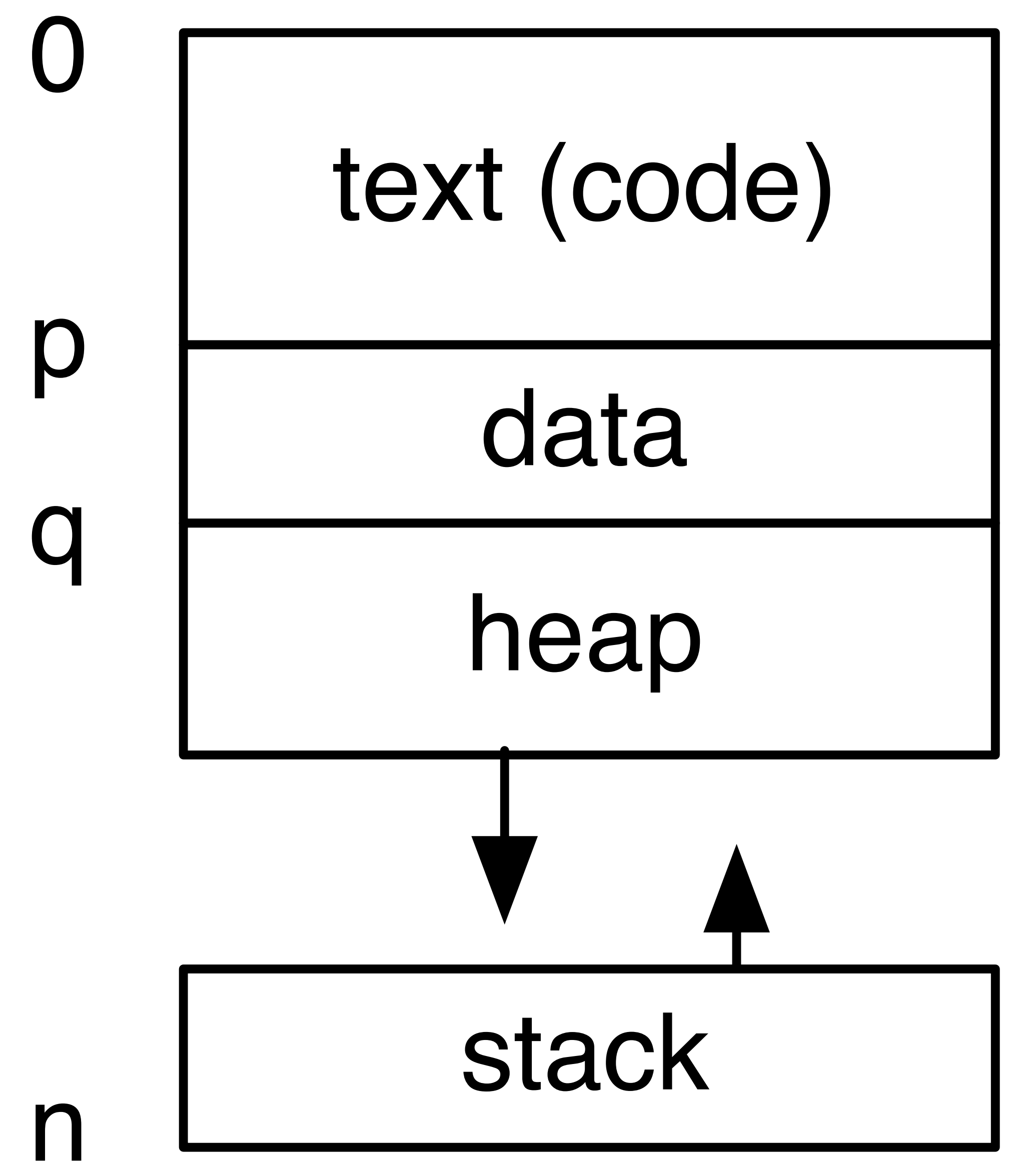


Disadvantage: can't easily share memory between processes

Process Protection

- How to protect the memory of a process (or the OS) from other running processes?
  - Software solution
    - Tagged data (e.g., Smalltalk/Lisp). No raw pointers
    - Virtual machine

Within a process address space

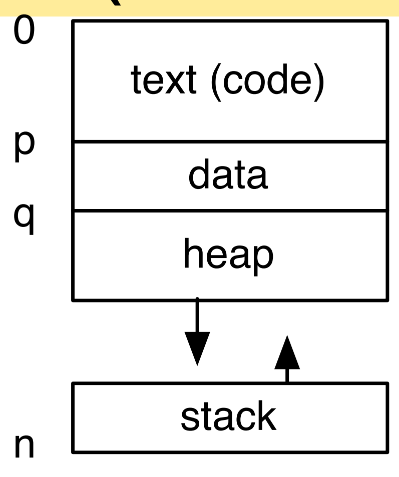


Fragmentation

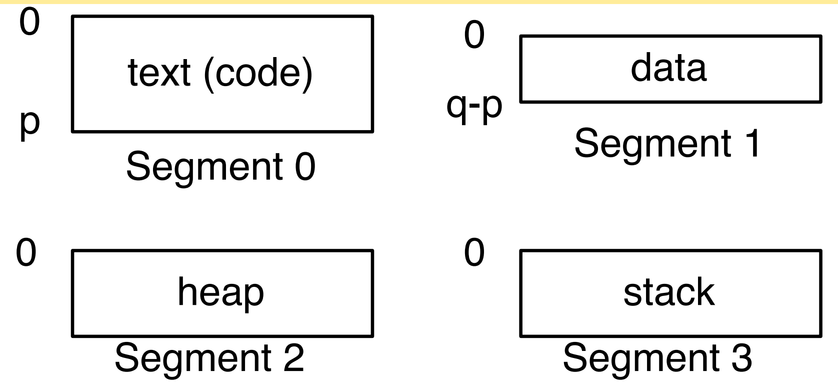
- External fragmentation
  - Space wasted *between* allocated memory blocks
  - Solutions
    - Compaction
      - Move blocks around dynamically to make free space contiguous. Only possible if relocation is dynamic at execution time
    - Non-contiguous
      - Don't require all of the memory of a process to be contiguous: Segmentation/Paging
- Internal fragmentation
  - Space wasted *within* allocated memory blocks
  - Solutions:
    - Don't use fixed-size blocks

Segmentation

- Provides multiple address spaces for a given process
  - Handy for separate data/stack/code
  - Good for sharing code/data between processes
  - Easy granularity to specify protection
    - no execute on stack!
  - Address is (segment num, offset within segment)
  - Need segment base/limit registers (1/segment)
  - Programmer (or compiler) must specify different segments



process with single segment



process with 4 segments

Segmentation

- Pros
  - Easy to share segments between processes
  - Segment-specific protection
  - No internal fragmentation
- Cons
  - Must still worry about external fragmentation: each segment must still have contiguous physical memory
  - Programmer/Compiler/Linker must be aware of different segments

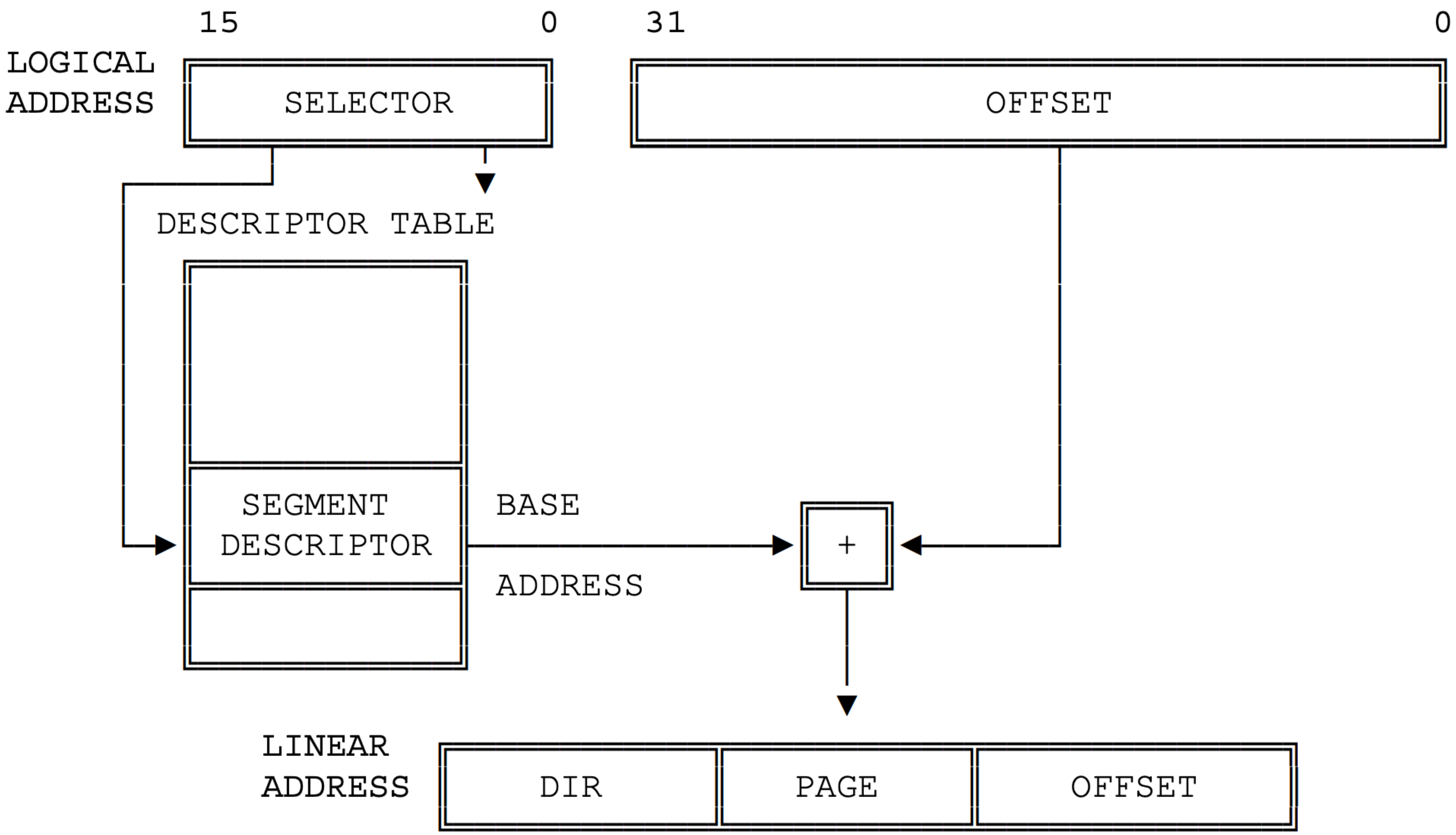
They are part of the context and should be swapped when switching between processes.

Segmentation on x86 (protected mode)

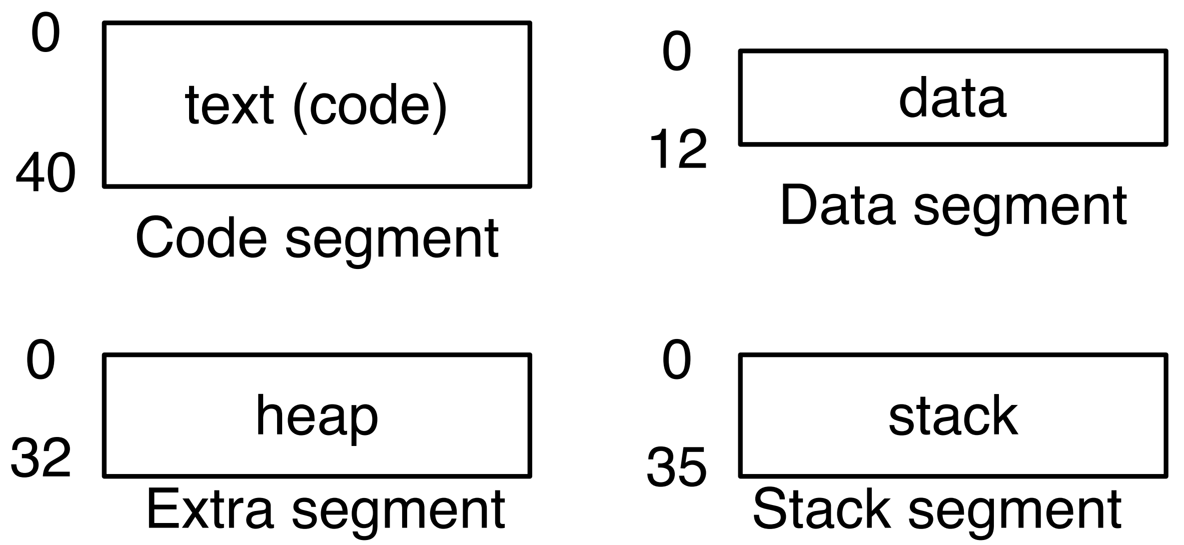
- There are four segments:
  - CS: code segment DS: data segment
  - SS: stack segment ES: extra segment
- Segment register contains:
  - Local/Global descriptor table bit
  - RPL: requested privilege level (2-bit)
  - an offset into a descriptor table
- Descriptor table entry contains:
  - 32-bit segment base
  - 20-bit segment limit (effectively 32-bit)
  - DPL: descriptor privilege level (2-bit)
  - Present bit

Segment registers does not contain the base register or limit register directly. Segment registers index into a descriptor table, each descriptor table entry contains the information.

Segmentation on x86 (protected mode)



Segmentation on x86, simplified example



The processor has some idea of which segment it should use. For example, when advancing the instruction pointer to point to the next instruction and fetch it, the processor knows that it should use the code segment. When encountering "push" or "pop", it knows that it should go to the stack segment.

Example code:

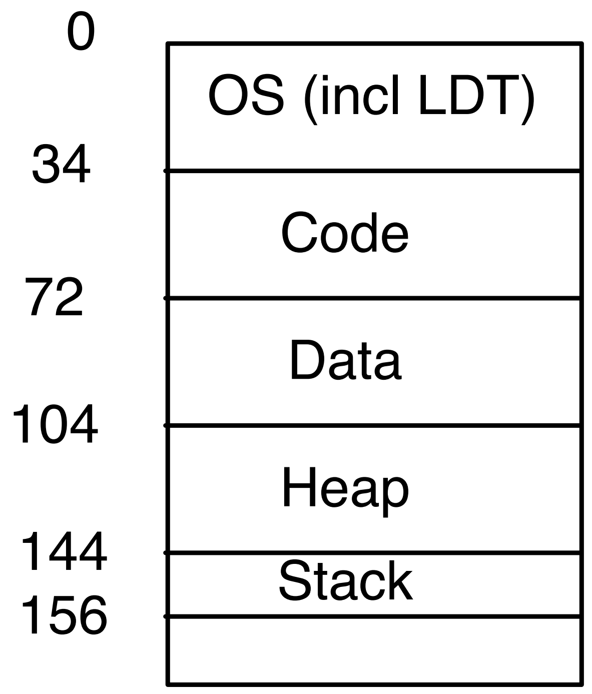
```
0x04: movl %(ebx), %eax
0x06: push %eax
0x08: add $0x2, %esp
```

Registers:

```
%CS: Local, CPL=3, offset=1
%DS: Local, CPL=3, offset=2
%SS: Local, CPL=3, offset=3
%ES: Local, CPL=3, offset=4
%SP: 12
%LDTR: 14
```

Local Descriptor table:

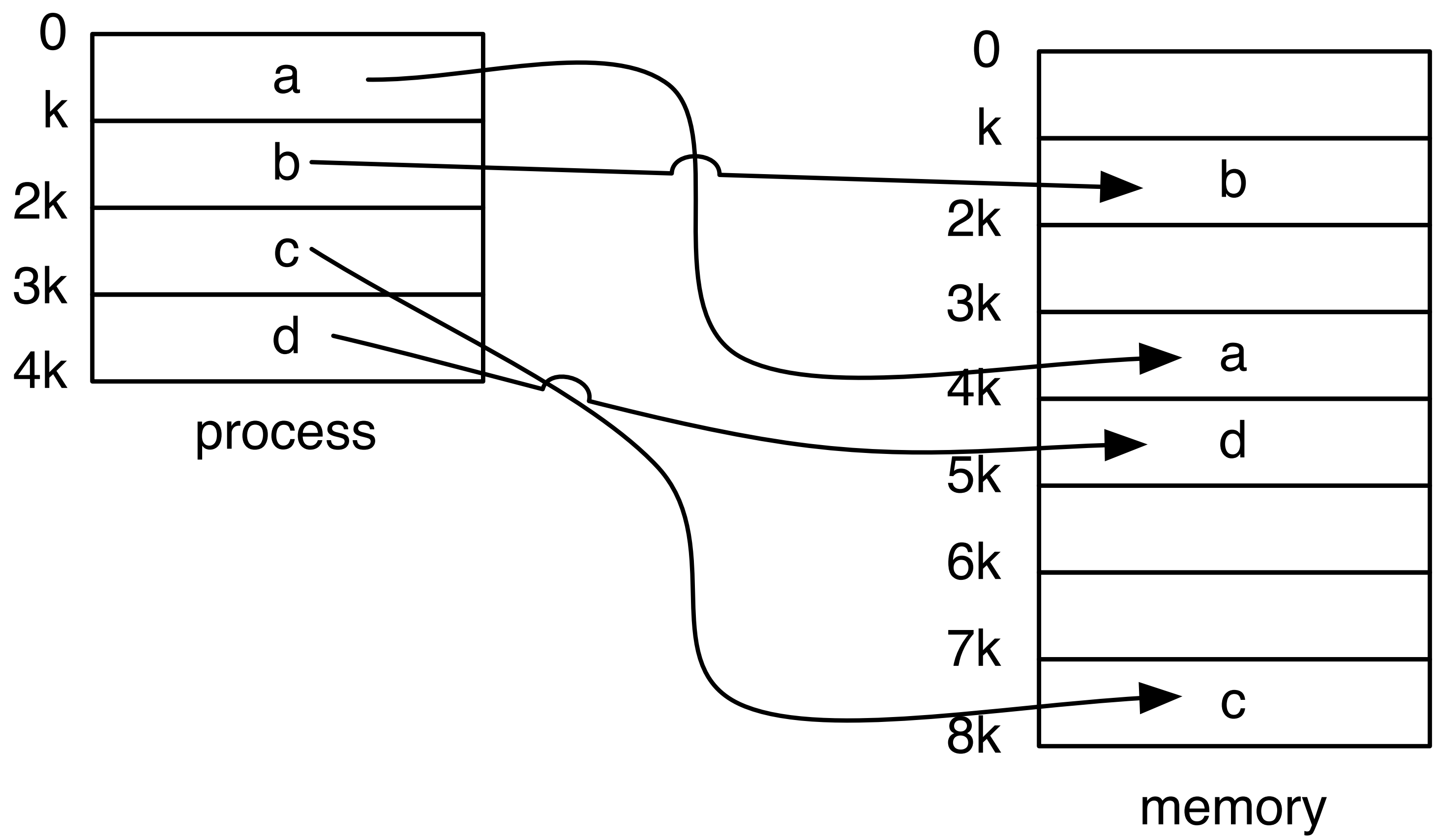
```
0: unused
1: Bounds=104, Limit=40,DPL=3,P
2: Bounds=144, Limit=12,DPL=3,P
3: Bounds= 72, Limit=32,DPL=3,P
4: Bounds= 34, Limit=38,DPL=3,P
```



Memory

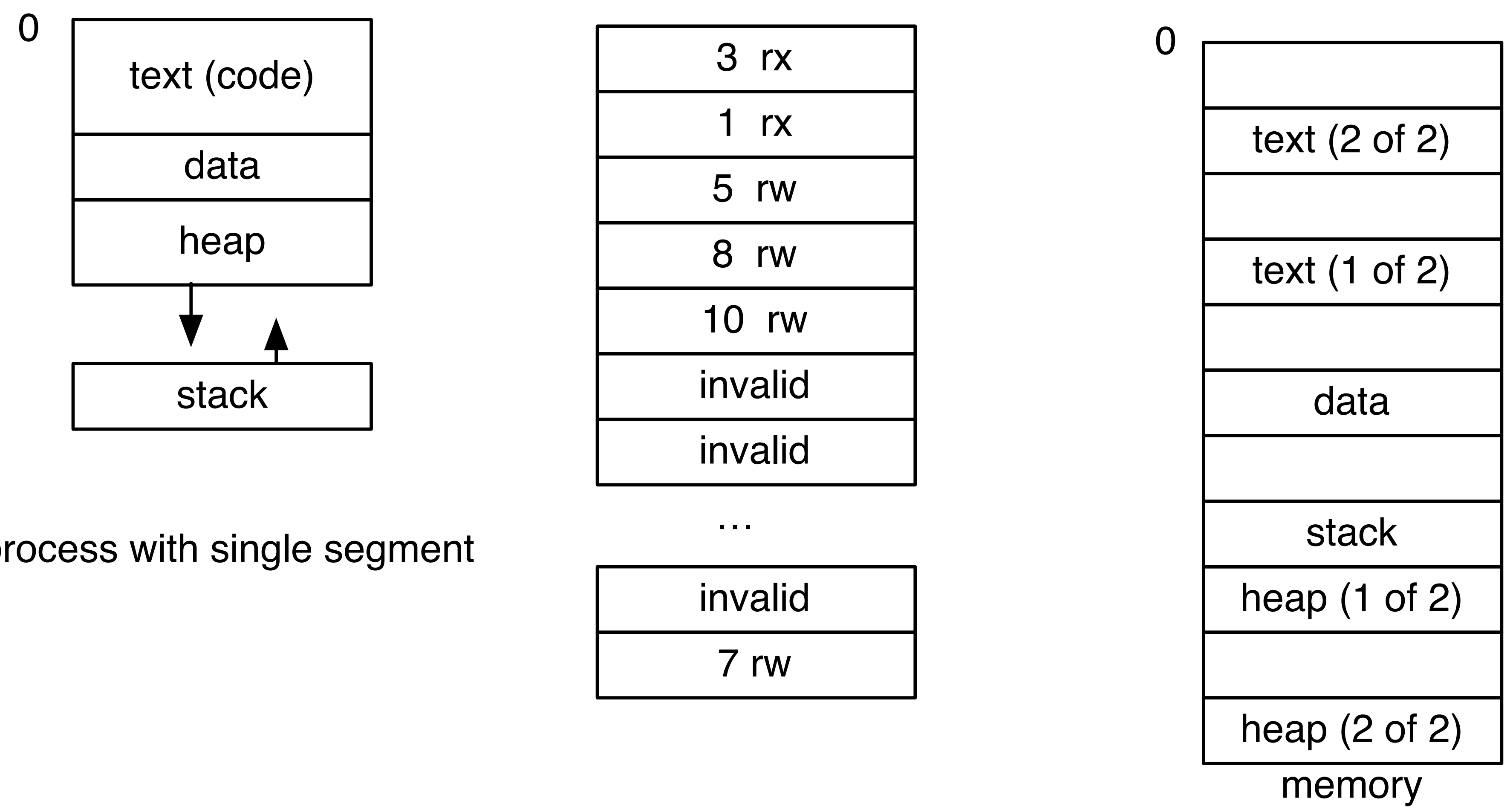
Paging

- Map contiguous virtual address space to non-contiguous physical address space
  - Idea:
    - break virtual address space into fixed-size *pages* (aka virtual pages)
    - break physical address space into fixed size *frames* (aka physical pages)



Implementing Paging

- Page table
  - index into the table is a *page number*
  - Each entry is a *page table entry*
    - frame number
    - Present bit (valid/not valid)
    - Permissions
    - Modified
    - …



Implementing Paging

- Done in hardware (MMU)
  - Page-table register points to page table (physical address)
    - Must be saved/restored on context-switch
    - Page table per process

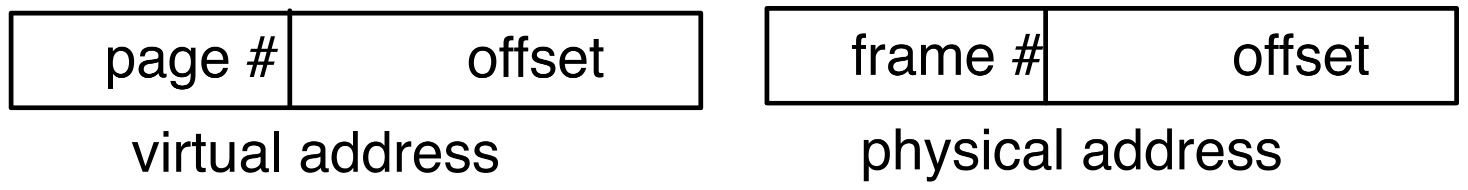
```
convertToPhysicalAddress(logicalAddress) {
  pageNumber = upper bits of logicalAddress
  if pageNumber out of range, generate exception (aka fault)
  if !pageTable[pageNumber].present generate page fault
  offset = lower bits of logicalAddress
  upper bits of physicalAddress=pageTable[pageNumber].frameNumber
  lower bits of physicalAddress = offset
```

Paging

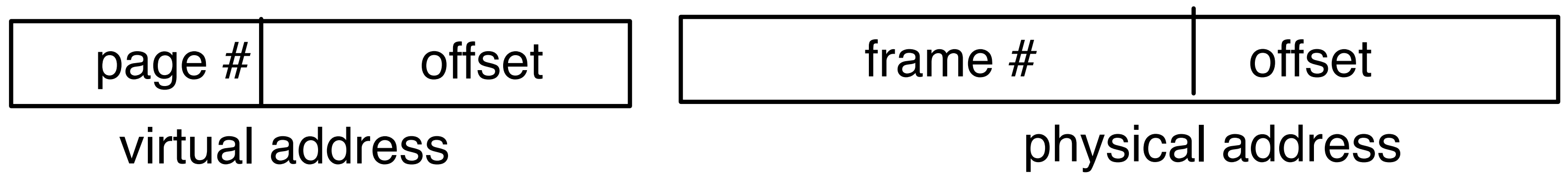
- Pros
  - No external fragmentation
    - Unallocated memory can be allocated to any process
  - Transparent to programmer/compiler/linker
  - Can put individual frames out to disk (VM)
- Cons
  - Translating from virtual to physical address requires an additional memory access
  - Unused pages in a process still require page table entries
  - Internal fragmentation
    - On average, 1/2 frame size per “segment”

Relationship between virtual/physical addresses

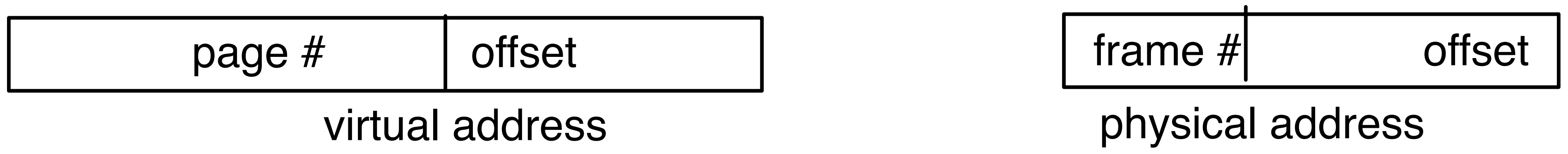
- Virtual address the same size as the physical address



- Virtual address smaller than physical address



- Virtual address bigger than physical address

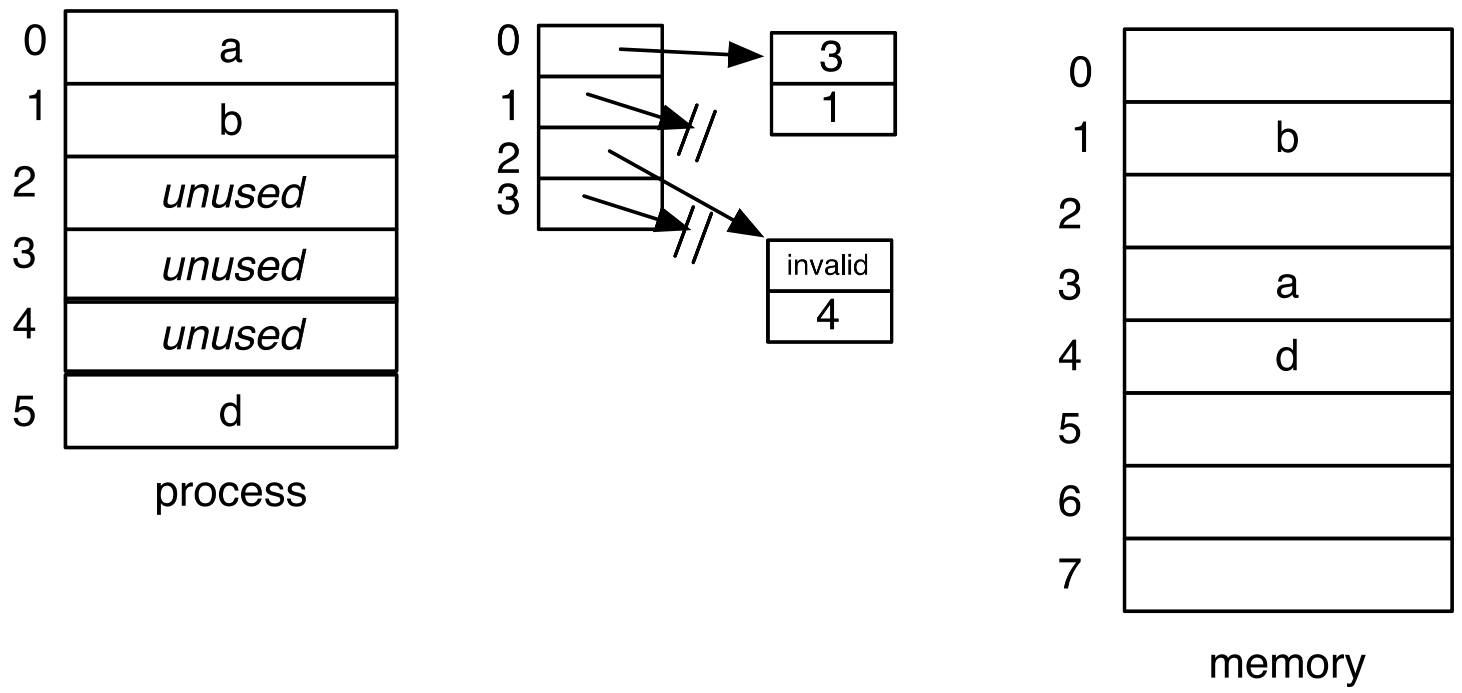


Dealing with holes in virtual address space

- With large address space (and small pages), page tables can be very large
  - 32-bit virtual addresses, 4K page: 2^{20} page table entries/process

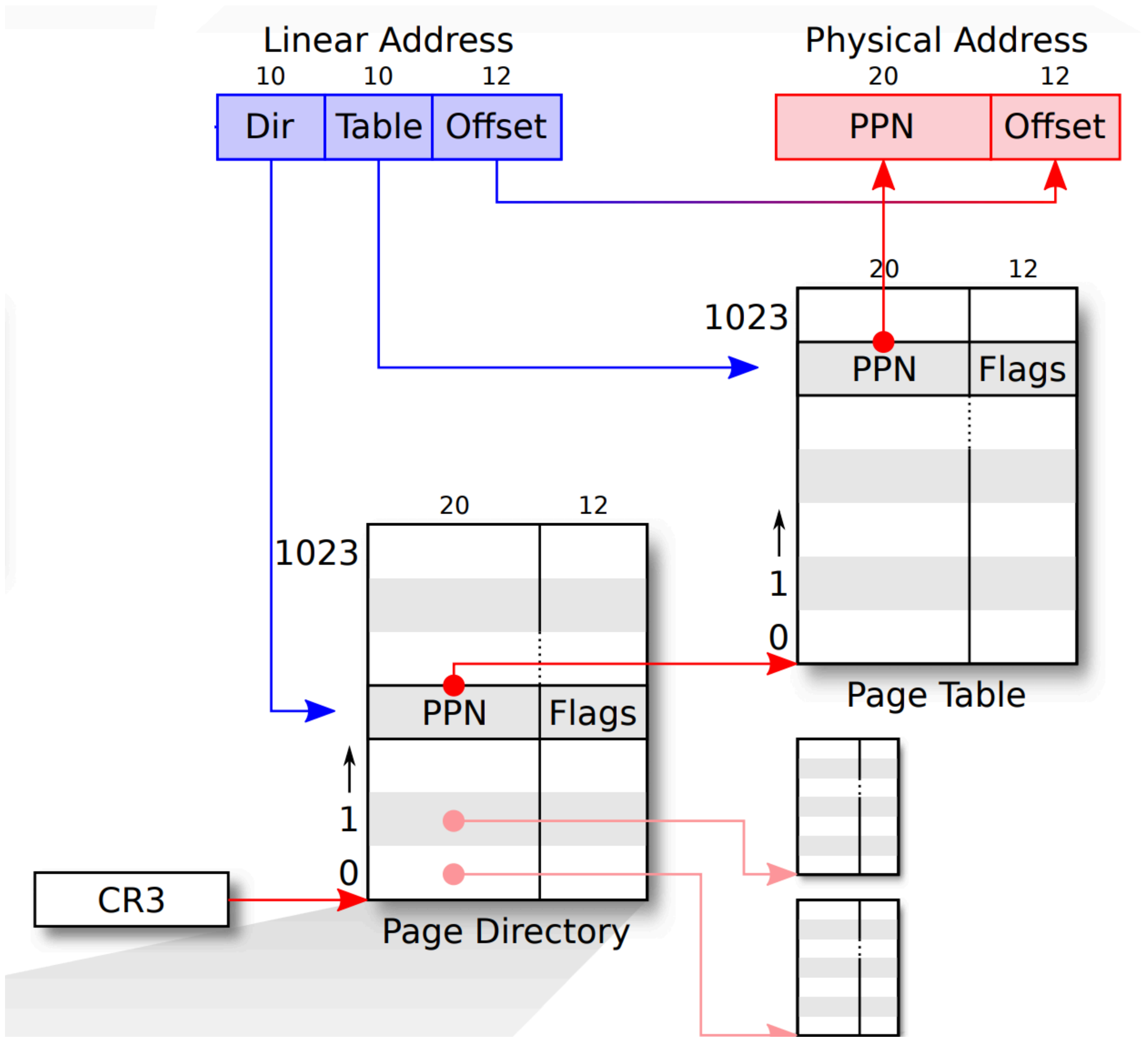
Dealing with holes in virtual address space

- Multilevel page table
  - Virtual address broken down into multiple page numbers: PT1 and PT2
  - PT1 is used as index into top-level PT to find second-level PT
  - PT2 is used as index into second-level PT to find PTE
  - If parts of address space are unused, top-level PT can show second-level PT not present

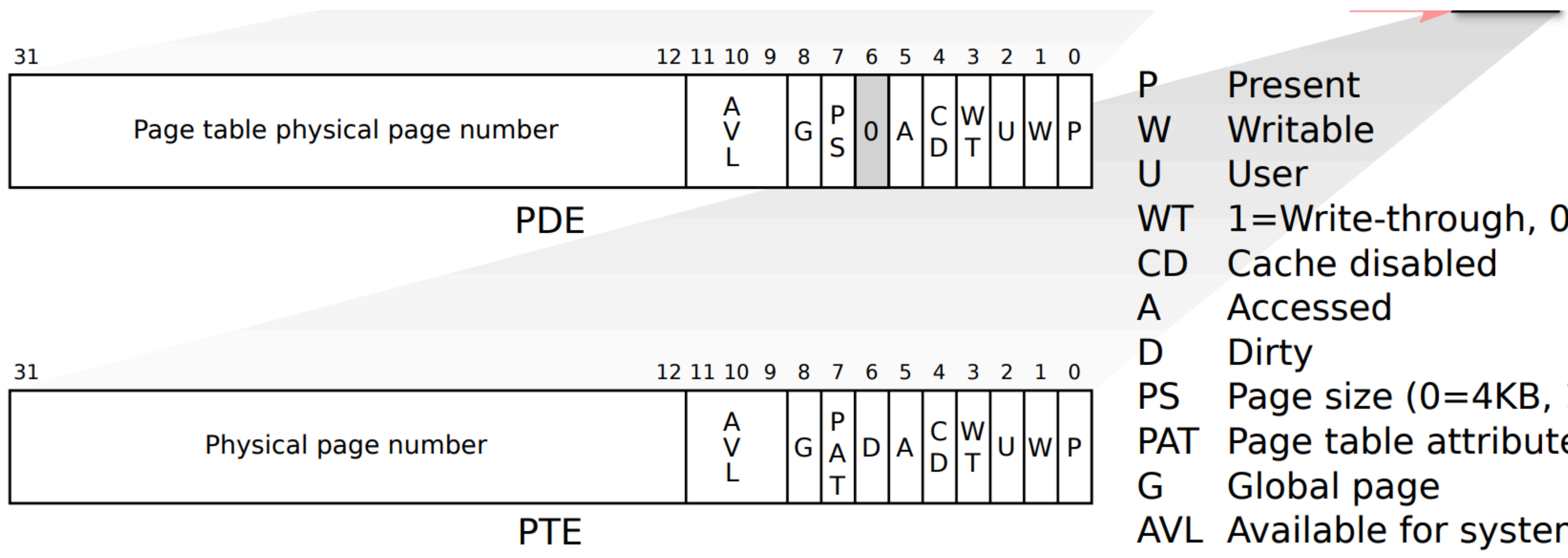


Paging on x86

- %CR3 contains pointer to first-level page table (*Page directory*).

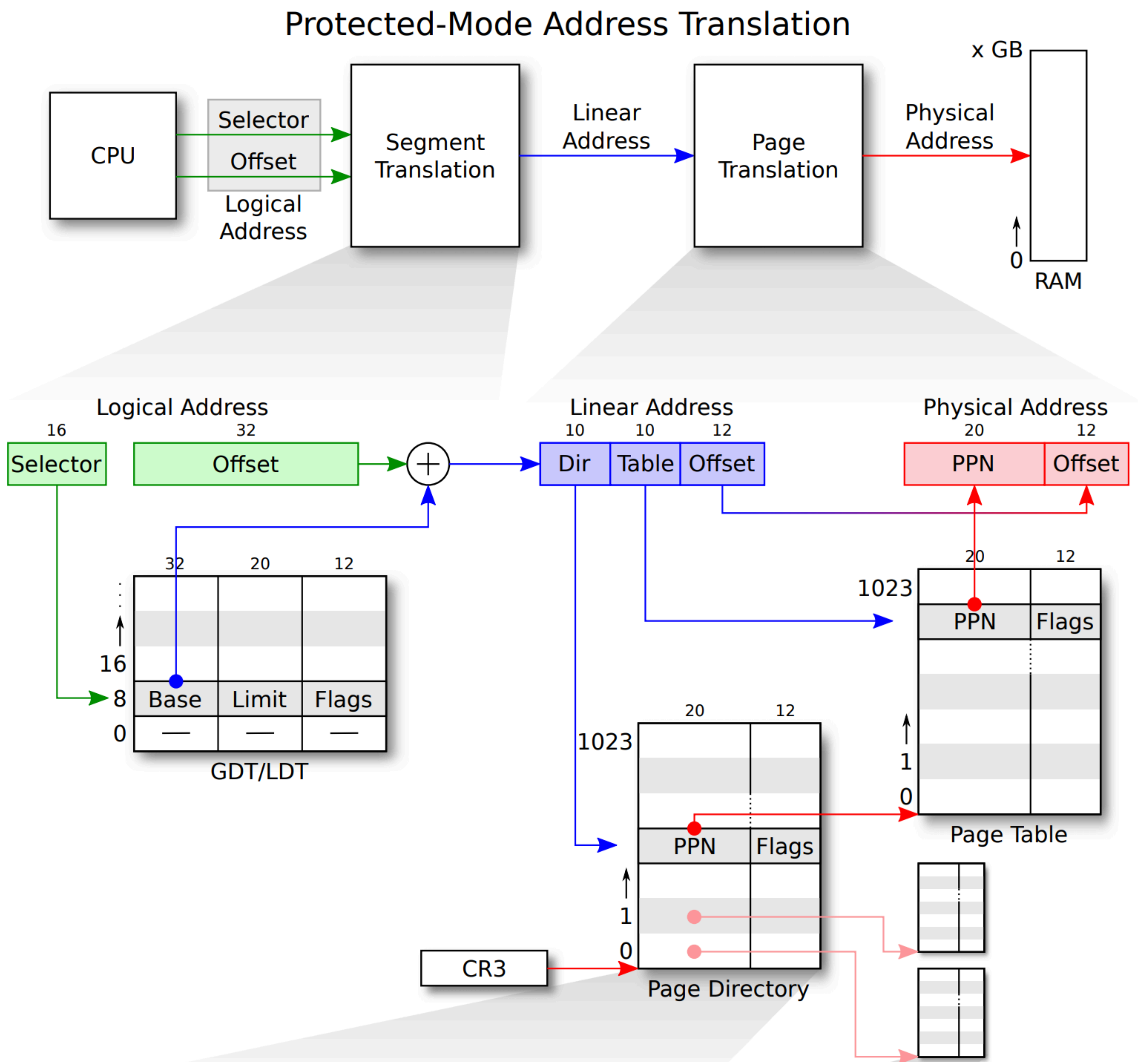


Page table entries on x86



P	Present
W	Writable
U	User
WT	1=Write-through, 0=Write-back
CD	Cache disabled
A	Accessed
D	Dirty
PS	Page size (0=4KB, 1=4MB)
PAT	Page table attribute index
G	Global page
AVL	Available for system use

Segmentation/Paging interaction on x86



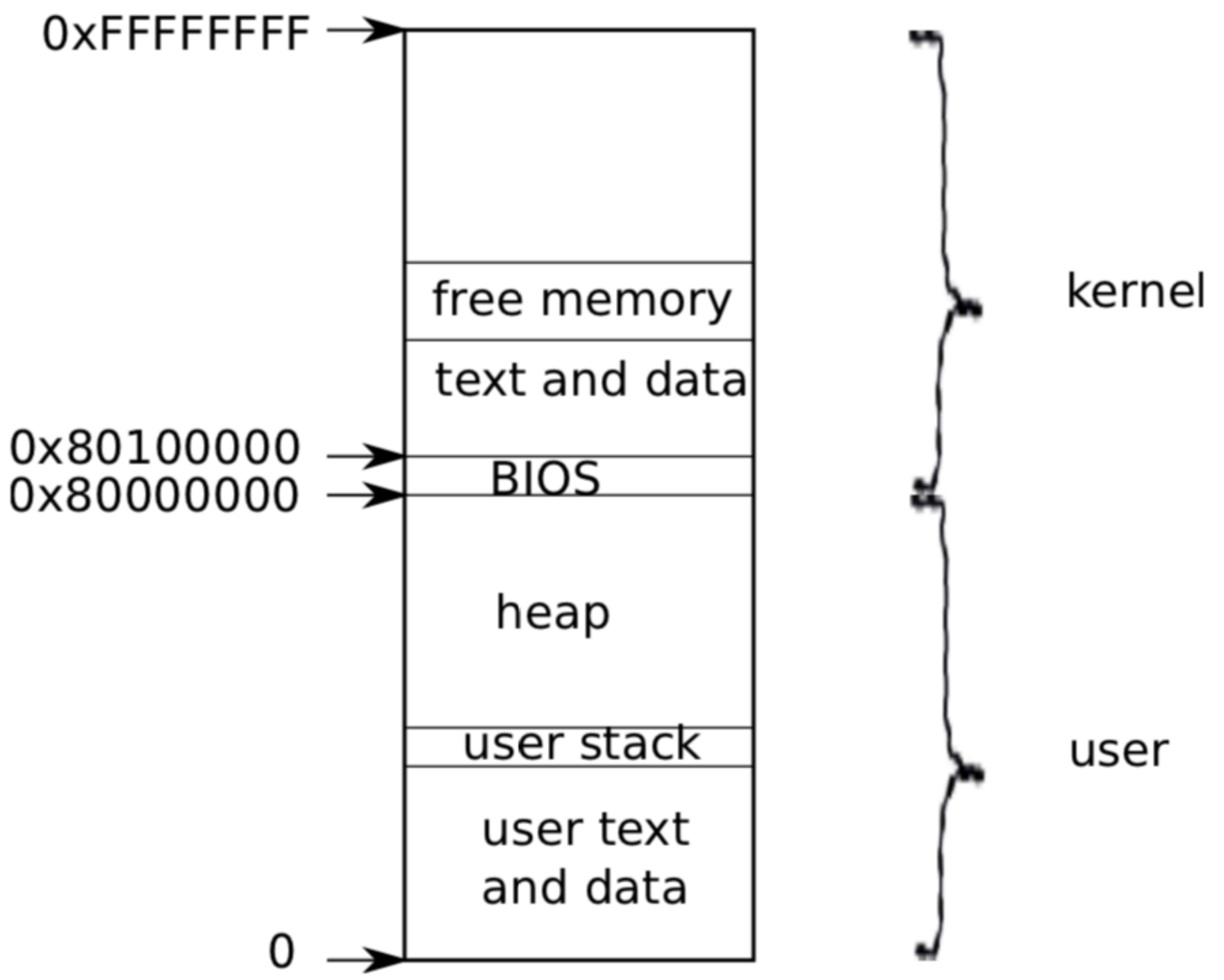
Exceptions

- What if the PDE or PTE doesn't have Present bit set?
  Or, what if trying to write and W bit not set?
  Or, what if trying access and U mode not set?
  - Page fault exception
  - Kernel can:
    - Kill process
    - Install/modify PTE and then resume the process
      - For example, after loading the page of memory from disk

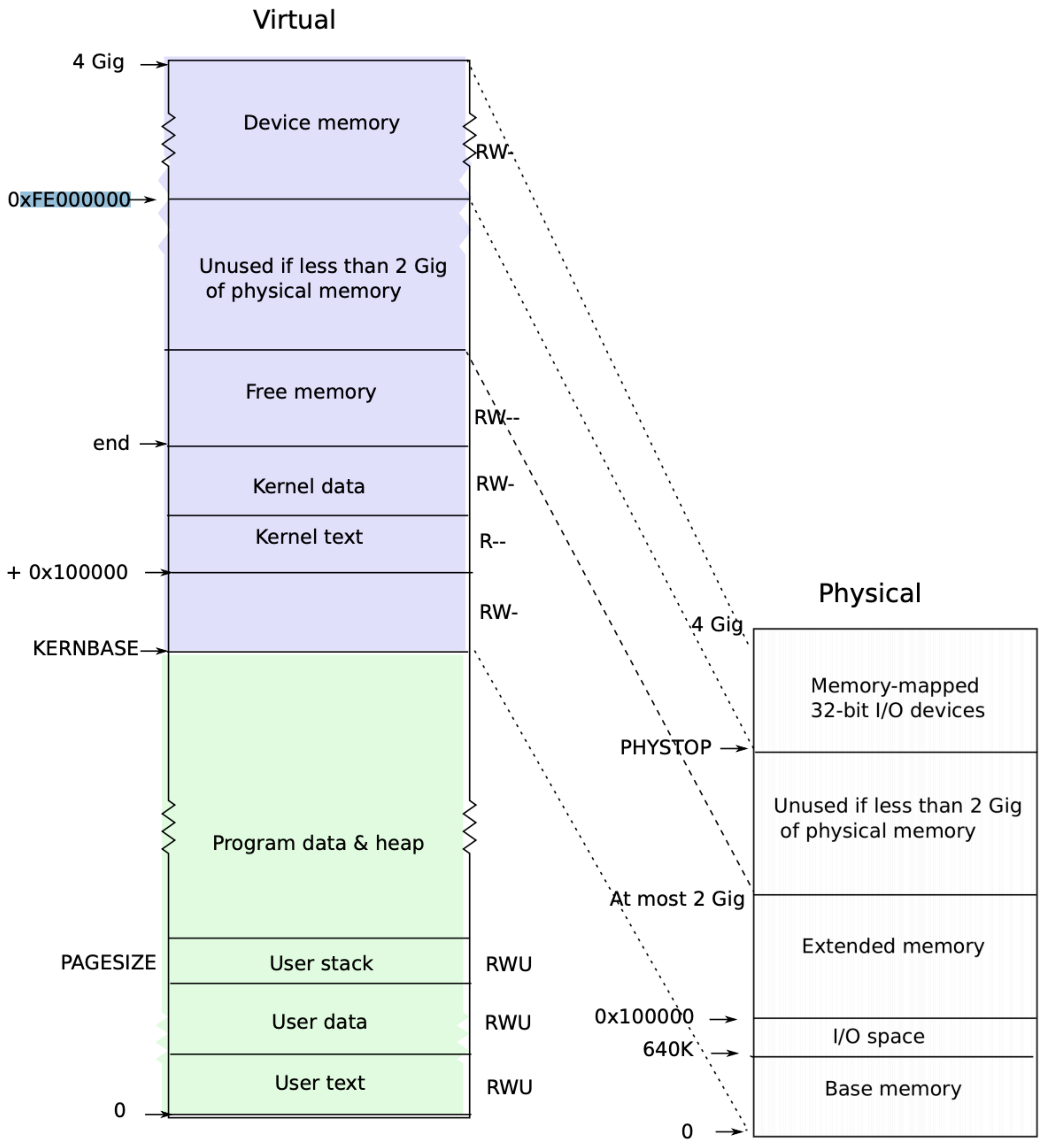
Why use virtual memory in kernel?

- Why does the kernel need a page table?
  - Hardware often makes it difficult to turn off paging
  - Would need to turn it off/on when entering/exiting a system call
  - Without page table, we'd have external memory fragmentation

xv6 address space



xv6 address space



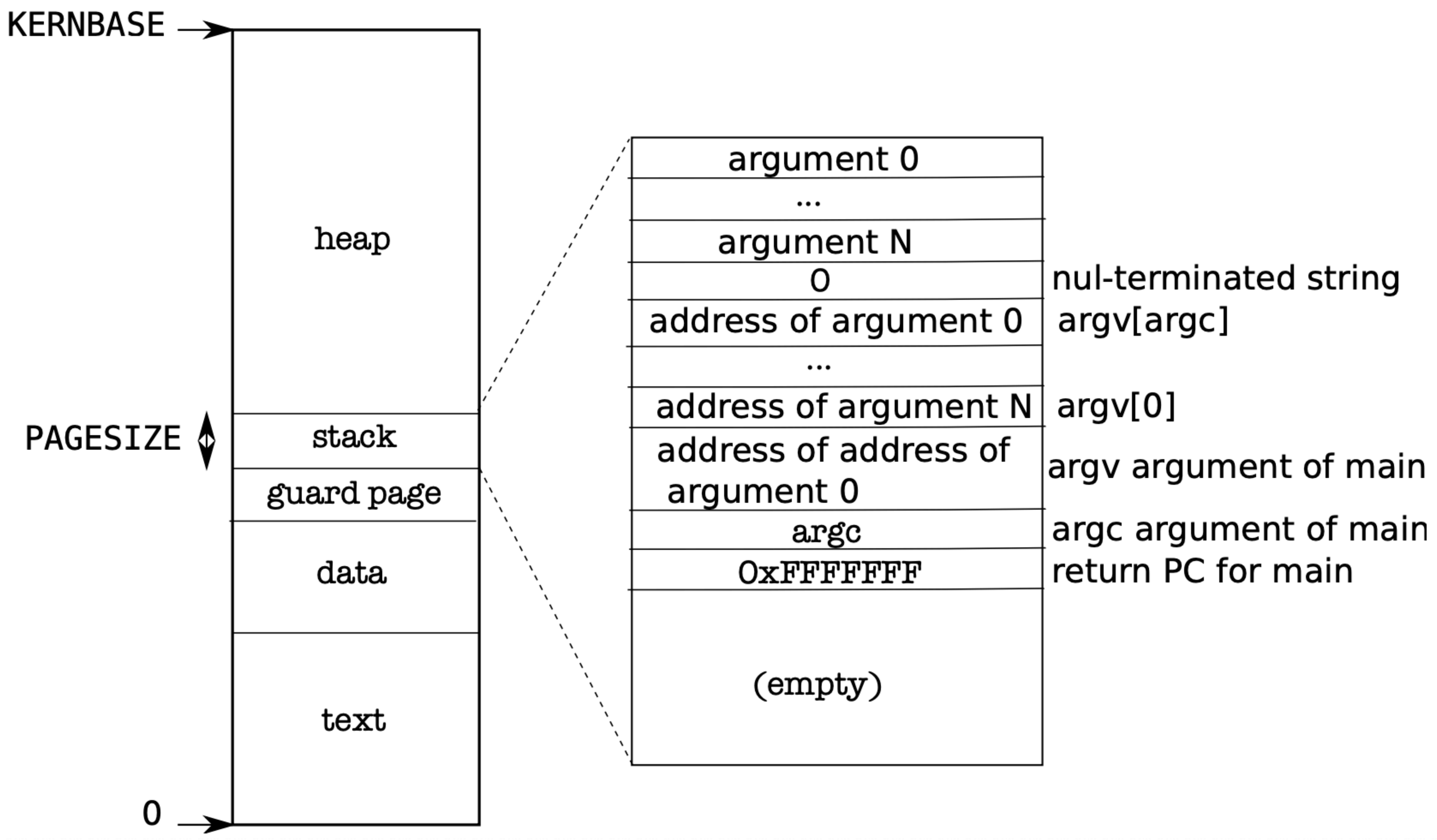
Processes on xv6

- Each has its own:
  - address space
  - page table (although second half of page dir and associated page tables could be shared)
- Kernel switches page tables (%CR3) when switching processes

Address space reasoning

- User virtual addresses start at 0
  - And go to 2GB, all contiguous. No external fragmentation. Pages can be mapped to any physical frame
- Kernel and user spaces are mapped:
  - No need to change page tables on a system call
- Kernel mapped at same place for all processes
  - Easier switching between processes
- Easy for kernel to R/W user memory
  - Just use user virtual addresses
- Easy for kernel to R/W physical memory
  - virtual addr = phys addr + 0x80000000

xv6: Memory layout of user process



xv6: Starting the first process

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

  p = allocproc();

  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
  …
}
```

proc.c

xv6: Starting the first process

```
void
user
{
  st
  ex


  p

  in
  if

  in

  p-
  me
  p-
  p-
  p-
  p-
  p-
  p-
  p-
  …
}
```

```
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
};

pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  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) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

vm.c

xv6: Starting the first process

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

  p = allocproc();

  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
  …
}
```

proc.c

xv6: Starting the first process

```
void
userinit(void)
{
  struct proc *p;
  extern char _binar
              _binar

  p = allocproc();

  initproc = p;
  if((p->pgdir = set
    panic("userinit:
  inituvm(p->pgdir, _
    (int)_binary_ini
  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
  …
}
```

proc.c

```
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, V2P(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```

vm.c

xv6: Starting the first process

```
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```

proc.c

xv6: Starting the first process

```
void
switchuvm(struct proc
{
  if(p == 0)
    panic("switchuvm:
  if(p->kstack == 0)
    panic("switchuvm:
  if(p->pgdir == 0)
    panic("switchuvm:

  pushcli();
  mycpu()->gdt[SEG_TS

  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = S
  mycpu()->ts.esp0 =
  // setting IOPL=0 i
  // forbids I/O inst
  mycpu()->ts.iomb =
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir))
  popcli();
}
```

before lcr3

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[80000-803ff]  PDE[200]     ----A--UWP
  …
  [80115-803ff]  PTE[115-3ff] --------WP 00115-003ff
[80400-8dfff]  PDE[201-237] ----A--UWP
  [80400-8dfff]  PTE[000-3ff] ---DA---WP 00400-0dfff
[fe000-febff]  PDE[3f8-3fa] -------UWP
  [fe000-febff]  PTE[000-3ff] --------WP fe000-febff
[fec00-fefff]  PDE[3fb]     ----A--UWP
  …
[ff000-fffff]  PDE[3fc-3ff] -------UWP
  [ff000-fffff]  PTE[000-3ff] --------WP ff000-fffff
```

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     -------UWP
  [00000-00000]  PTE[000]     -------UWP 0dfbd
[80000-803ff]  PDE[200]     -------UWP
  [80000-800ff]  PTE[000-0ff] --------WP 00000-000ff
  [80100-80106]  PTE[100-106] ---------P 00100-00106
  [80107-803ff]  PTE[107-3ff] --------WP 00107-003ff
[80400-8dfff]  PDE[201-237] -------UWP
  [80400-8dfff]  PTE[000-3ff] --------WP 00400-0dfff
[fe000-fffff]  PDE[3f8-3ff] -------UWP
  [fe000-fffff]  PTE[000-3ff] --------WP fe000-fffff
```

after lcr3

xv6: Starting the first process

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

vm.c

xv6: Starting the first process

```c
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

vm.c