EE488: System Software Design
Fall 2022

Page Tables and Virtual Address

Youjip Won

KAIST

Contents

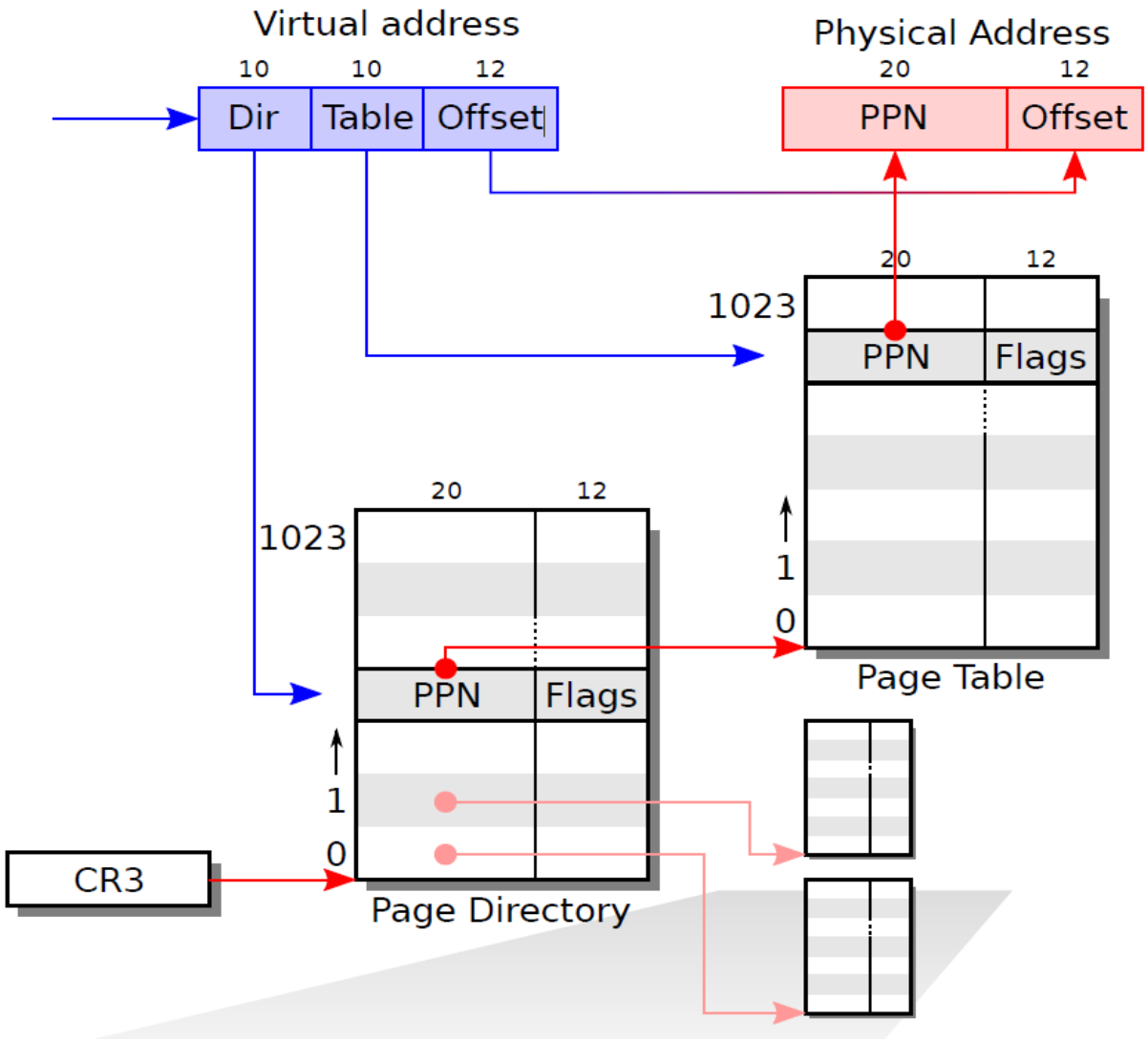
- Page table
- Page table structure
- Paging hardware
- Process address space
- Code: creating an address space
- Physical memory allocation
- Code : Physical memory allocator
- User part of an address space
- Code : sbrk (growproc)

Paging in xv6

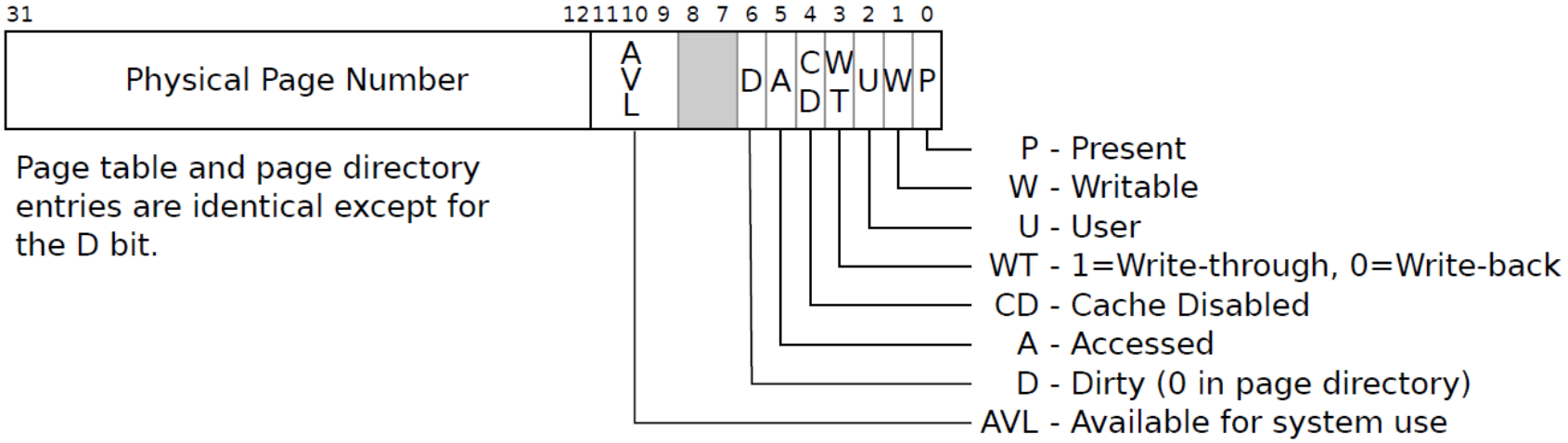
- Page Table
  - Two level tree
  - Can omit entire page table pages in which large ranges of virtual addresses have no mappings.

- used simple tricks
  - Mapping the same memory (the kernel) in several address spaces
  - Mapping the same memory more than once in one address space (each user page is also mapped into the kernel's physical view of memory)
  - Guarding a user stack with an unmapped page

Two Level Page Table Structure (Hardware)



Page table entry format



page table on the x86 architecture

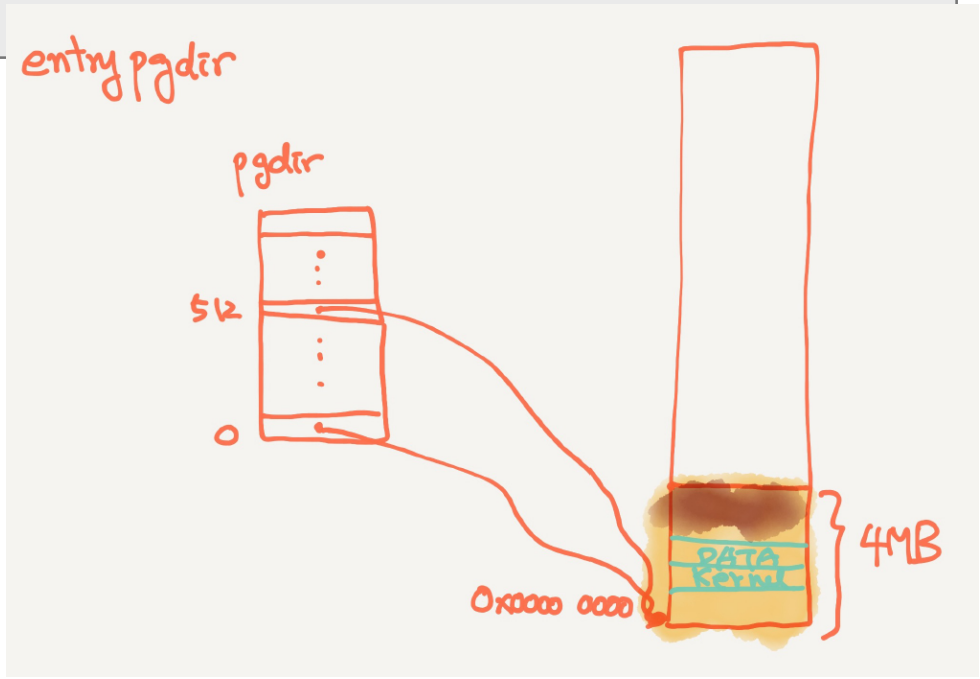
Process address space

- User Space: 0 – (KERNBASE-1), (2GB)
- Kernel Space: KERNBASE – 0xFFFFFFFF (2GB, clean PTE_U flag)
- Each process has private user address space.
- The processes share the kernel address space.
- When switching from the user space to the kernel space do not require page table switch.

The first page table before starting `main()`: `entrypgdir`

```
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE >> PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Index : 0 Index : 512



- `PTE_P` – Present Bit
- `PTE_W` – Read / Write
- `PTE_PS` – 4MB Page Size Bit

Memory position defined : `kernel.ld`

Linker script `kernel.ld` defines kernel's memory layout.

```
SECTIONS
{
    /* Link the kernel at this address:
       "." means the current address */
    . = 0x80100000; /* Must be equal to KERNLINK */

    .text : AT(0x100000) {
        *(.text .stub .text.* .gnu.linkonce.t.*)
    }

    PROVIDE(etext = .); /* Define the 'etext' symbol to this value */

    ...

    PROVIDE(data = .)

    /* The data segment */
    .data : {
        *(.data)
    }
```

Current counter (Position)

Sets current counter "." to 0x80100000 (Virtual Address)

Load kernel at 0x100000 (Physical Address_

Memory position defined : `kernel.ld` (Cont.)

Linker script `kernel.ld` defines kernel's memory layout.

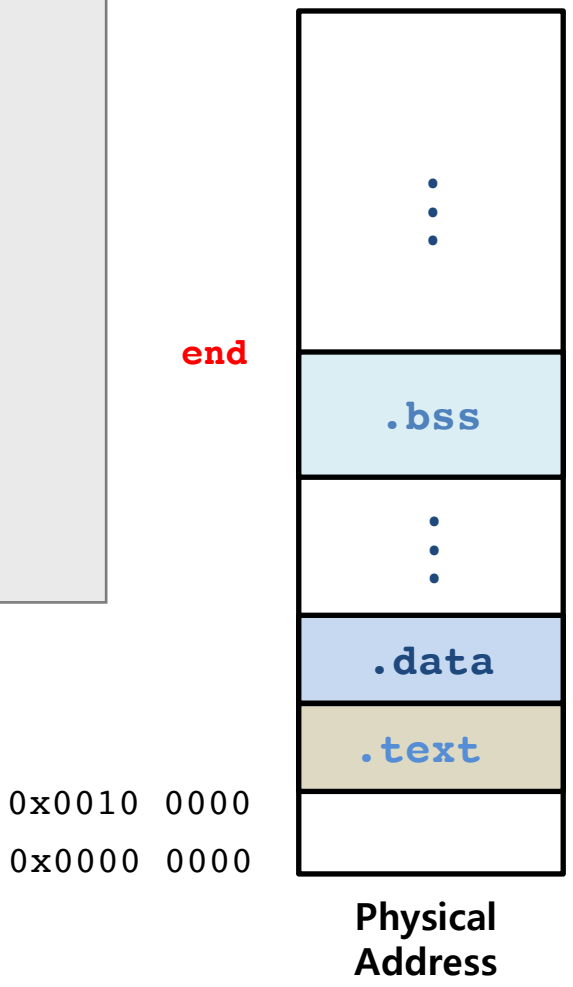
```
    ...

    .bss : {
        *(.bss)
    }

    PROVIDE(end = .);

    /DISCARD/ : {
        *(.eh_frame .note.GNU-stack)
    }
}
```

`end` symbol is defined as the next address of `.bss`



1. Create the free page list.

`kinit1()`: Create the "first" free page list of 1024 pages (4 Kbyte each).

```
extern char end[]; // first address after kernel loaded from ELF file

1216 int
1217 main(void)
1218 {
1219 kinit1(end, P2V(4*1024*1024));        // phys page allocator
1220 kvmalloc();          // kernel page table
…
1234 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startoth
ers()
1235 userinit();          // first user process
1236 mpmain();           // finish this processor's setup
1237 }
```

4 Mbyte address space

Allocating the free pages

Add the pages from `vstart` to `vend` to freepage list, `kmem`.

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

```
struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;
```

Prepare the free page list.

Add page *v to free page list, kmem.

```
void kfree(char *v) {
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```

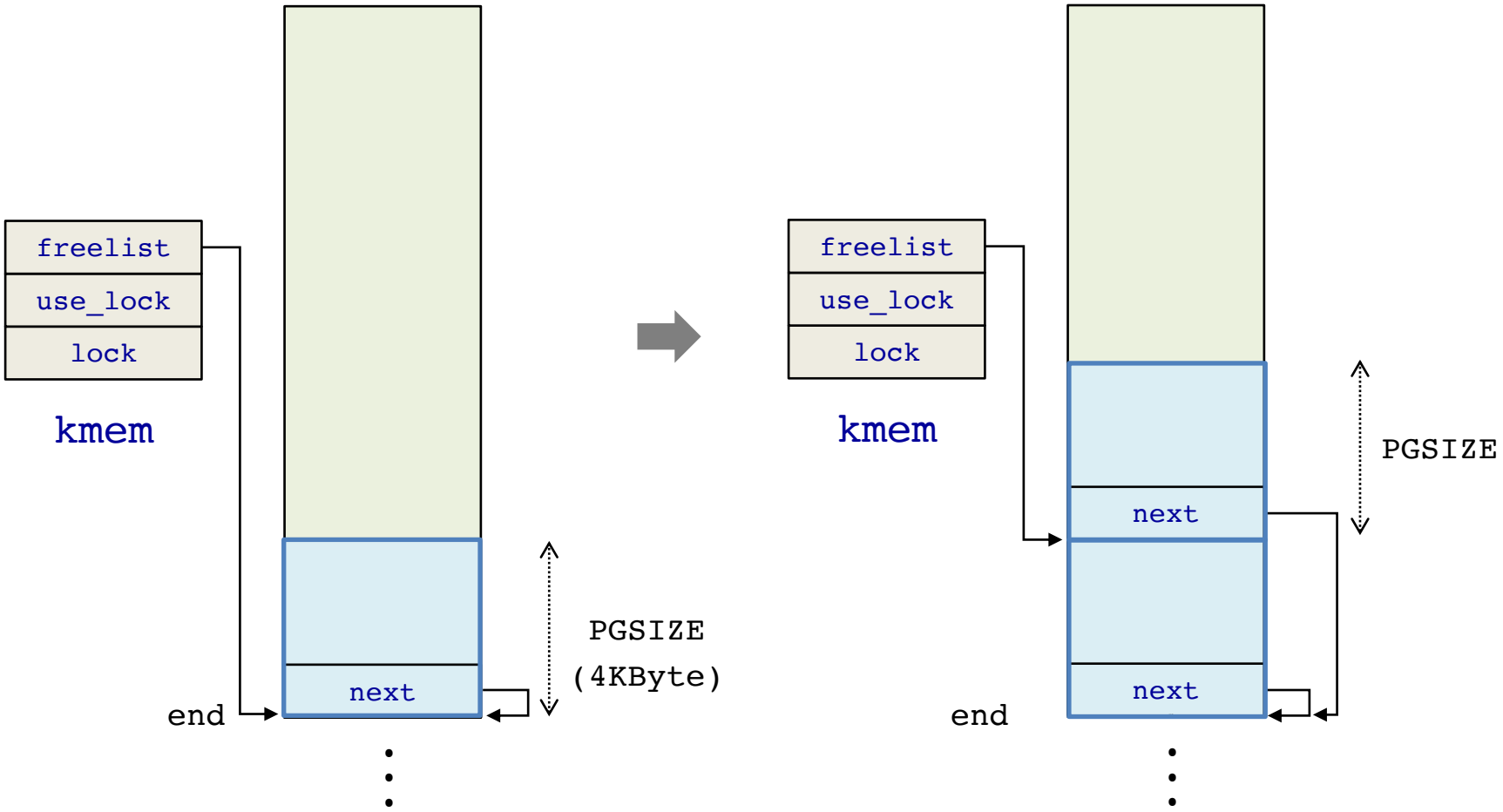
Is the address v valid?

```
struct run {
    struct run *next;
};

struct {
    struct spinlock lock;
    int use_lock;
    struct run *freelist;
} kmem;
```

Prepare the free page list. (Cont.)

Add page `*v` to free page list, `kmem`.



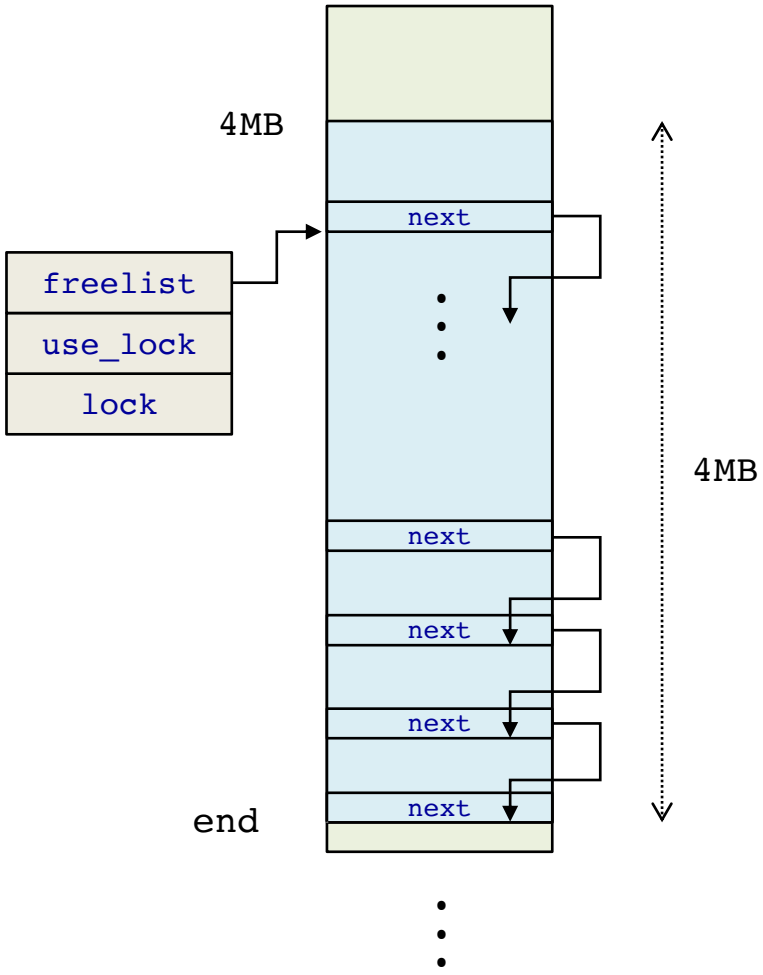
Prepare the free page list. (Cont.)

Add page *v to free page list, kmem.

```
void kfree(char *v) {

  ...

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```



2. Create virtual address space for kernel.

```
1216 int
1217 main(void)
1218 {
1219 kinit1(end, P2V(4*1024*1024));        // phys page allocator
1220 kvmalloc();              // kernel page table
…
1234 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startoth
ers()
1235 userinit();             // first user process
1236 mpmain();               // finish this processor's setup
1237 }
```

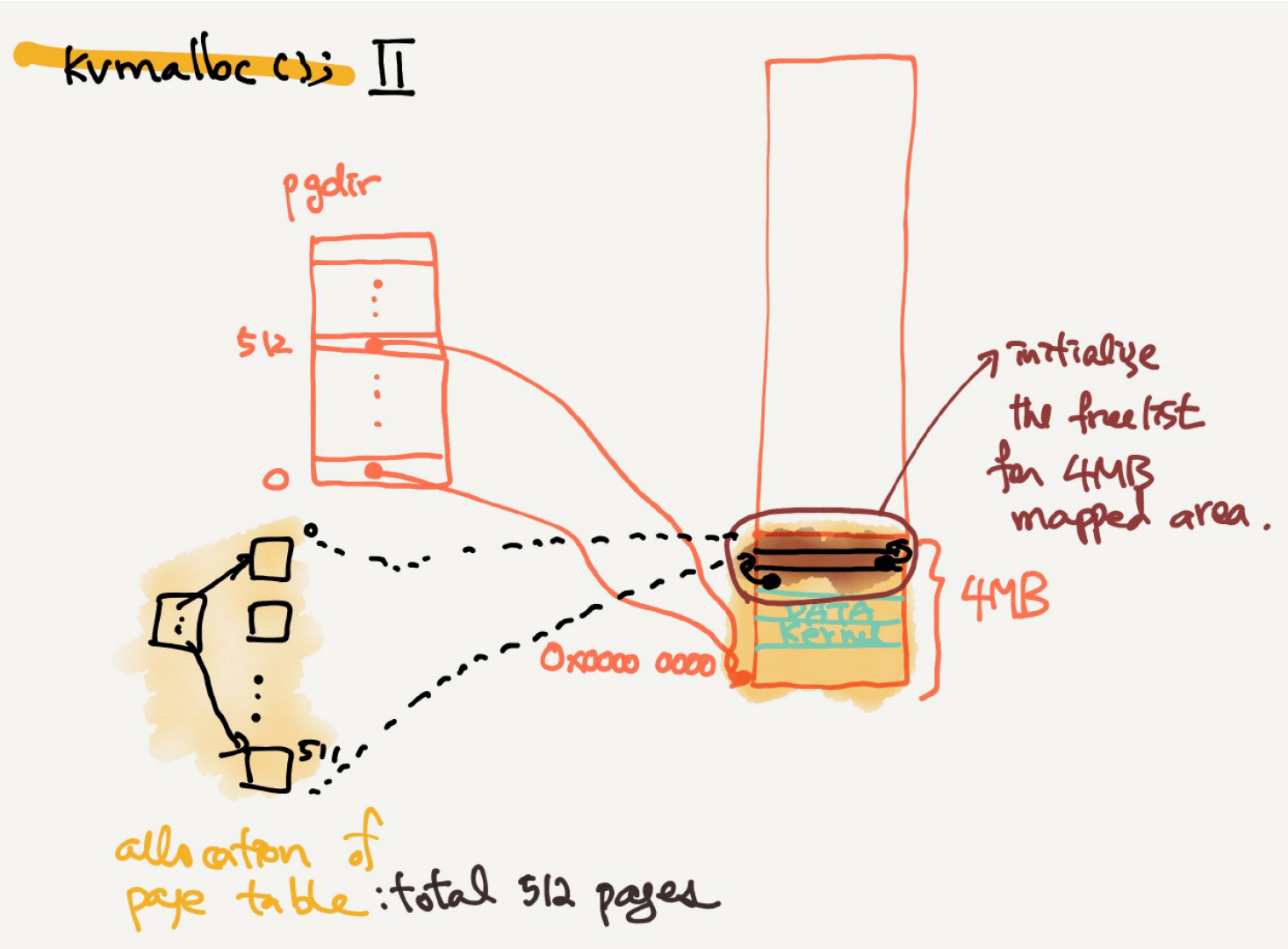
build a kernel page table and switch to it

Create a virtual address space for kernel. (Cont.)

```
1837 // Allocate one page table for the machine for the kernel address
1838 // space for scheduler processes.
1839 void
1840 kvmalloc(void)
1841 {
1842       kpgdir = setupkvm();
1843       switchkvm();
1844 }
```

Allocate page table for the kernel address.

After `setupkvm()`



Create a page table and map it.

```
1817 pde_t*
1818 setupkvm(void)
1819 {
1820      pde_t *pgdir;
1821      struct kmap *k;
1822
1823      if((pgdir = (pde_t*)kalloc()) == 0)
1824           return 0;
1825      memset(pgdir, 0, PGSIZE);
1826      if (P2V(PHYSTOP) > (void*)DEVSPACE)
1827           panic("PHYSTOP too high");
1828      for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1829           if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1830          (uint)k->phys_start, k->perm) < 0) {
1831           freevm(pgdir);
1832           return 0;
```

Allocate the page for page directory.
Fill it with 0.

Setup a page table for kernel space

```
1817 pde_t*
1818 setupkvm(void)
1819 {
1820      pde_t *pgdir;
1821      struct kmap *k;
1822
1823      if((pgdir = (pde_t*)kalloc()) == 0)
1824           return 0;
1825      memset(pgdir, 0, PGSIZE);
1826      if (P2V(PHYSTOP) > (void*)DEVSPACE)
1827           panic("PHYSTOP too high");
1828      for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1829           if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
1830      (uint)k->phys_start, k->perm) < 0) {
1831           freevm(pgdir);
1832           return 0;
```

Mapping virtual and physical addresses

V2P(a) macro

- input : virtual kernel memory address
- output : physical kernel memory address

```
#define V2P(a) (((uint) (a)) - KERNBASE)
```

```
ex) V2P(KERNLINK)

KERNELINK = KERNBASE + EXTMEM

V2P(KERNLINK) = (KERNBASE + EXTMEM) - KERNBASE = EXTMEM
```

Kernel address map

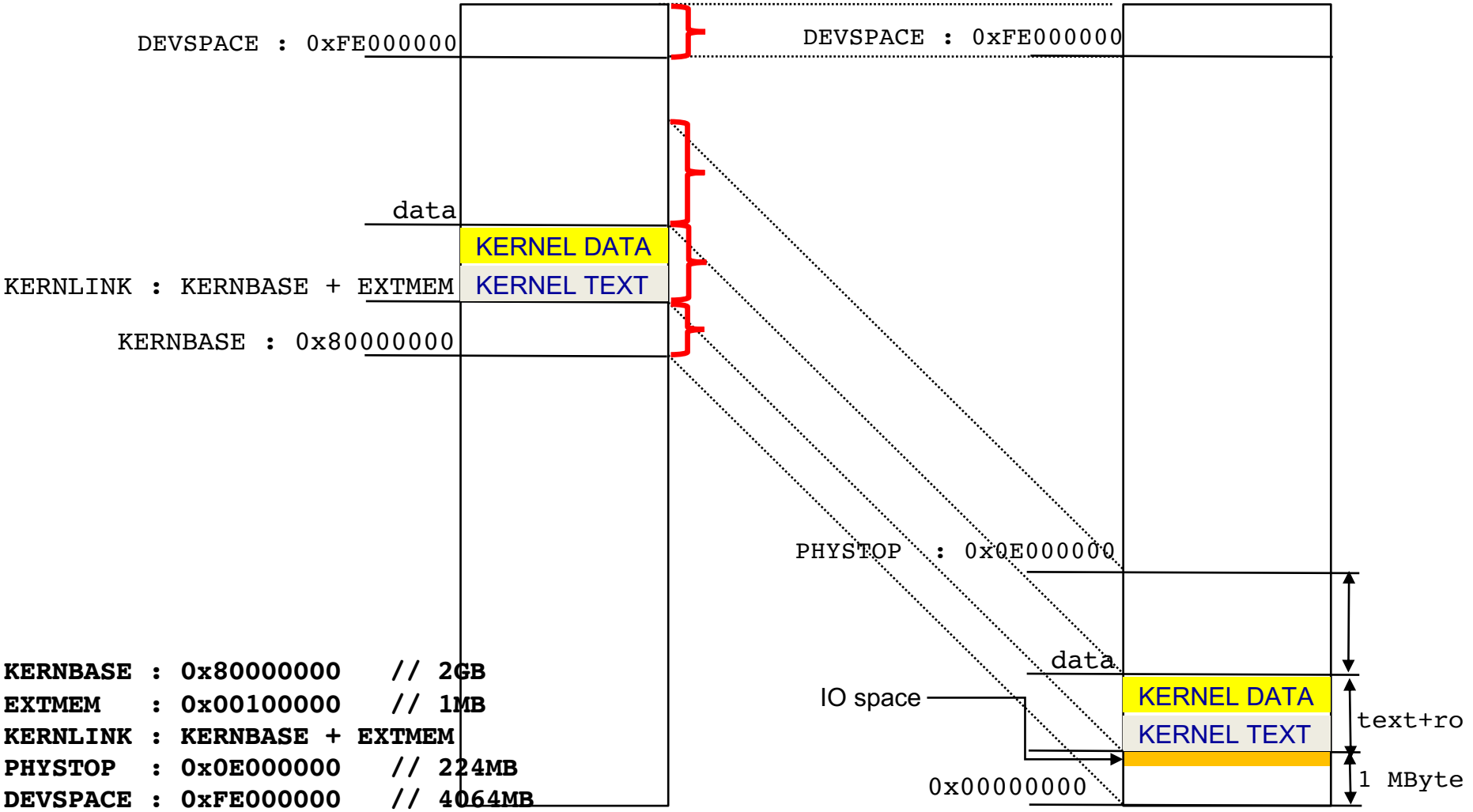
```
1804 static struct kmap {
1805       void *virt;
1806       uint phys_start;
1807       uint phys_end;
1808       int perm;
1809 } kmap[] = {
```

`phys_start` and `phys_end`

is used to calculate size of area.

```
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
};
```

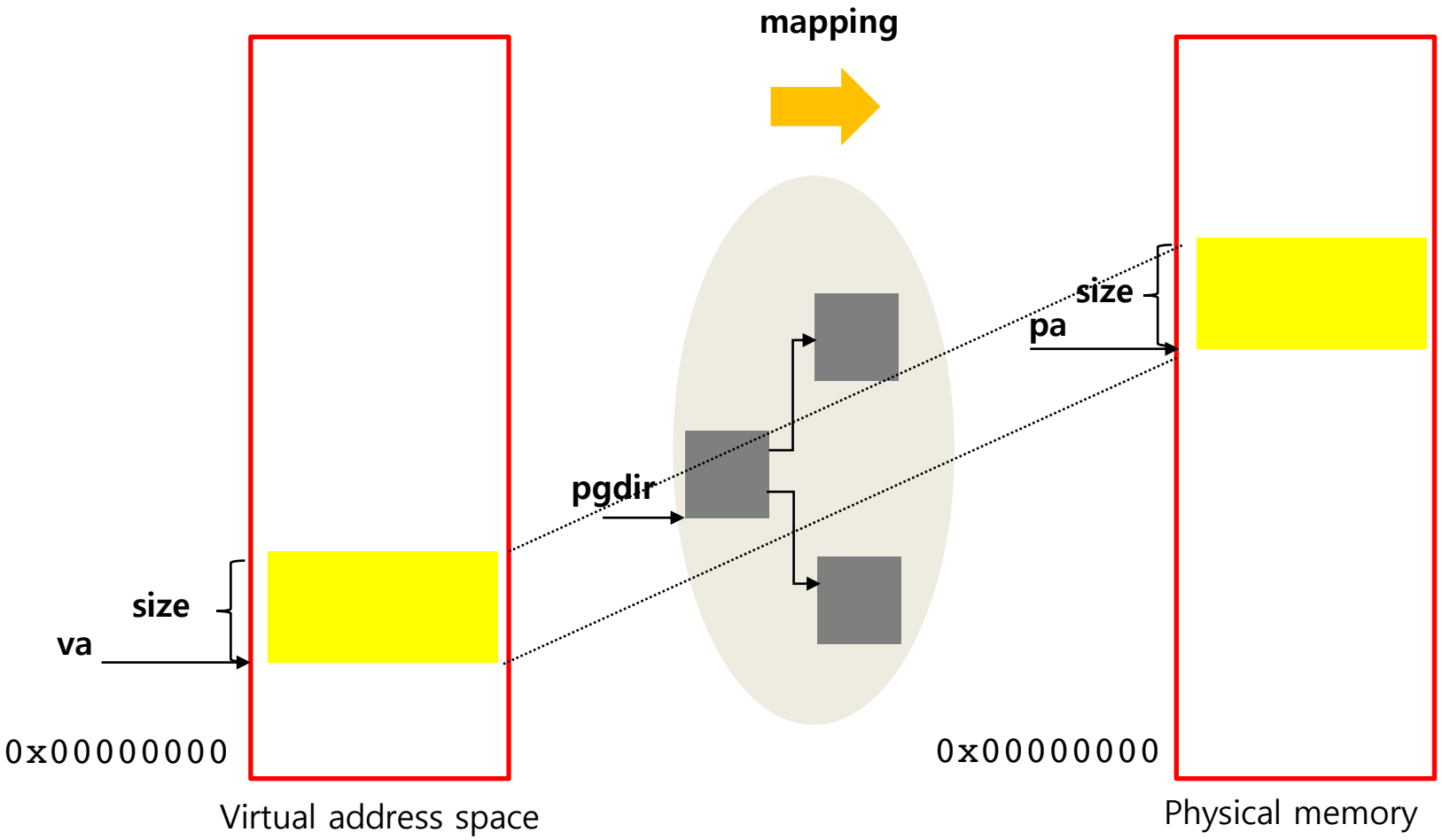
Map the address space




```
KERNBASE : 0x80000000    // 2GB
EXTMEM   : 0x00100000    // 1MB
KERNLINK : KERNBASE + EXTMEM
PHYSTOP  : 0x0E000000    // 224MB
DEVSPACE : 0xFE000000    // 4064MB
```

Map the virtual address: set up the page table entries.

```
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
```



Fill up the page table for kernel

```
1759 static int
1760 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
1761 {
1762       char *a, *last;
1763       pte_t *pte;
1764
1765       a = (char*)PGROUNDDOWN((uint)va);
1766       last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
1767       for(;;){
1768       if((pte = walkpgdir(pgdir, a, 1)) == 0)
1769             return -1;
```

Macro function to match virtual address to page size

Return `pte` with virtual address as an argument

Fill up the page table for kernel (Cont.)

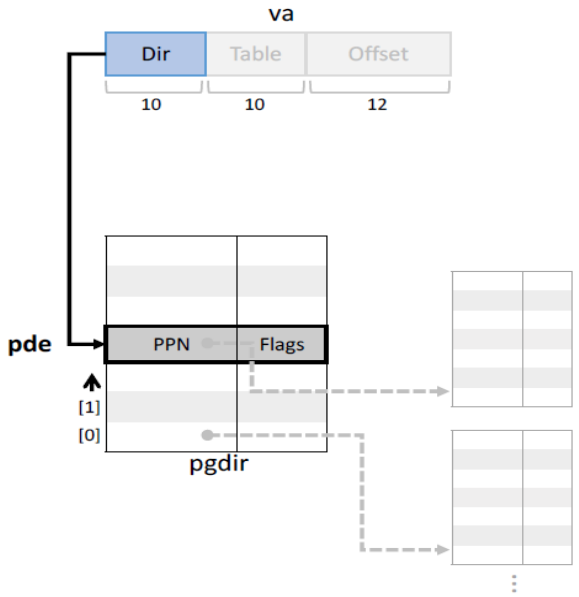
```
1770        if(*pte & PTE_P)
1771            panic("remap");
1772        *pte = pa | perm | PTE_P;
1773        if(a == last)
1774            break;
1775        a += PGSIZE;
1776        pa += PGSIZE;
1777        }
1778        return 0;
1779 }
```

Initialize the PTE to hold the PPN, permission and and **PTE_P.**

Code: creating an address space (Cont.)

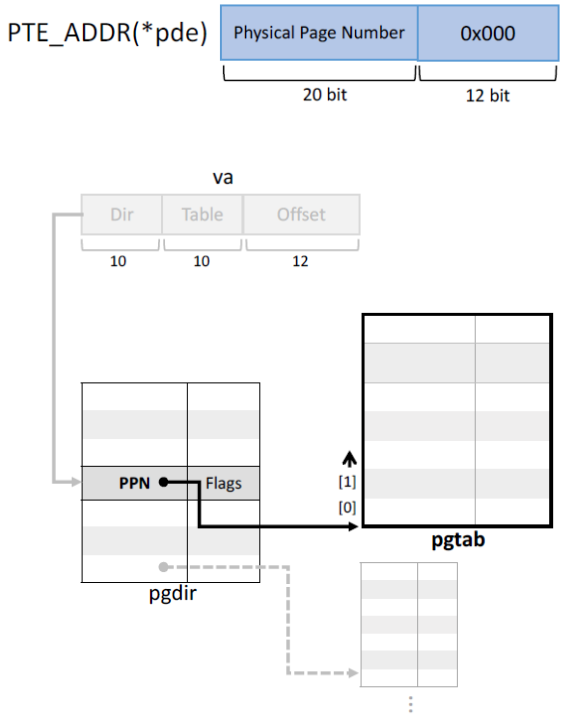
```
1734 static pte_t * walkpgdir(pde_t *pgdir, const void *va,
1736                            int alloc) {
1737      pde_t *pde;
1738      pte_t *pgtab;
1739
1740      pde = &pgdir[PDX(va)];
1741      if(*pde & PTE_P){ // PTE_P : page is present.
1742           pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743      } else {
1744           if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745                return 0;
1746           // Make sure all those PTE_P bits are zero.
1747           memset(pgtab, 0, PGSIZE);
1748           // The permissions are overly generous, but they
1749           // can be restricted by the permissions in the
1750           // page table entries, if necessary.
1751          *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752      }
1753      return &pgtab[PTX(va)];
1754 }
```

PDX(va)



Code: creating an address space (Cont.)

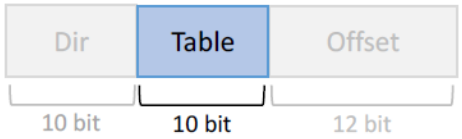
```
1734 static pte_t * walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737      pde_t *pde;
1738      pte_t *pgtab;
1739
1740      pde = &pgdir[PDX(va)];
1741      if(*pde & PTE_P){
1742           pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743      } else {
1744           if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745                return 0;
1746           // Make sure all those PTE_P bits are zero.
1747           memset(pgtab, 0, PGSIZE);
1748           // The permissions here are overly generous, but they
1749           // can be further restricted by the permissions in the
1750           // page table entries, if necessary.
1751          *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752      }
1753      return &pgtab[PTX(va)];
1754 }
```

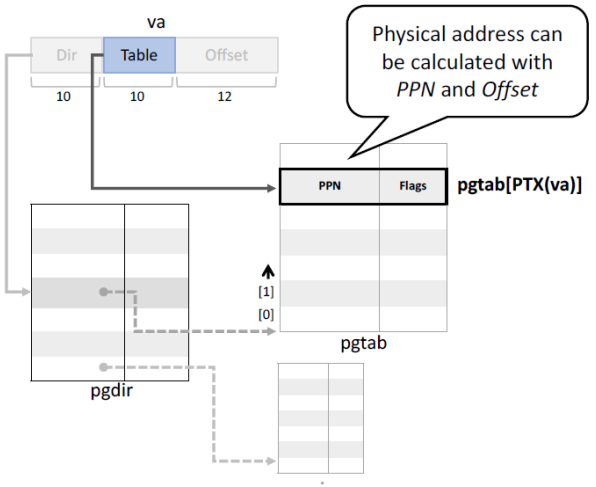


Code: creating an address space (Cont.)

```
1734 static pte_t * walkpgdir(pde_t *pgdir, const void *va, int alloc)
1736 {
1737      pde_t *pde;
1738      pte_t *pgtab;
1739
1740      pde = &pgdir[PDX(va)];
1741      if(*pde & PTE_P){
1742           pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
1743      } else {
1744           if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
1745                return 0;
1746           // Make sure all those PTE_P bits are zero.
1747           memset(pgtab, 0, PGSIZE);
1748           // The permissions here are overly generous, but they
1749           // can be further restricted by the permissions in the
1750           // page table entries, if necessary.
1751          *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
1752      }
1753      return &pgtab[PTX(va)];
1754 }
```

PTX(va)





Physical memory allocation

- The kernel must allocate and free physical memory at run-time for page tables, process user memory, kernel stacks, and pipe buffers.
- xv6 uses the physical memory between the end of the kernel and PHYSTOP for run-time allocation.
- All of physical memory must be mapped in order for the allocator to initialize the free list, but creating a page table with those mappings involves allocating page-table pages.
- xv6 solves this problem by using a separate page allocator during entry, which allocates memory just after the end of the kernel's data segment.

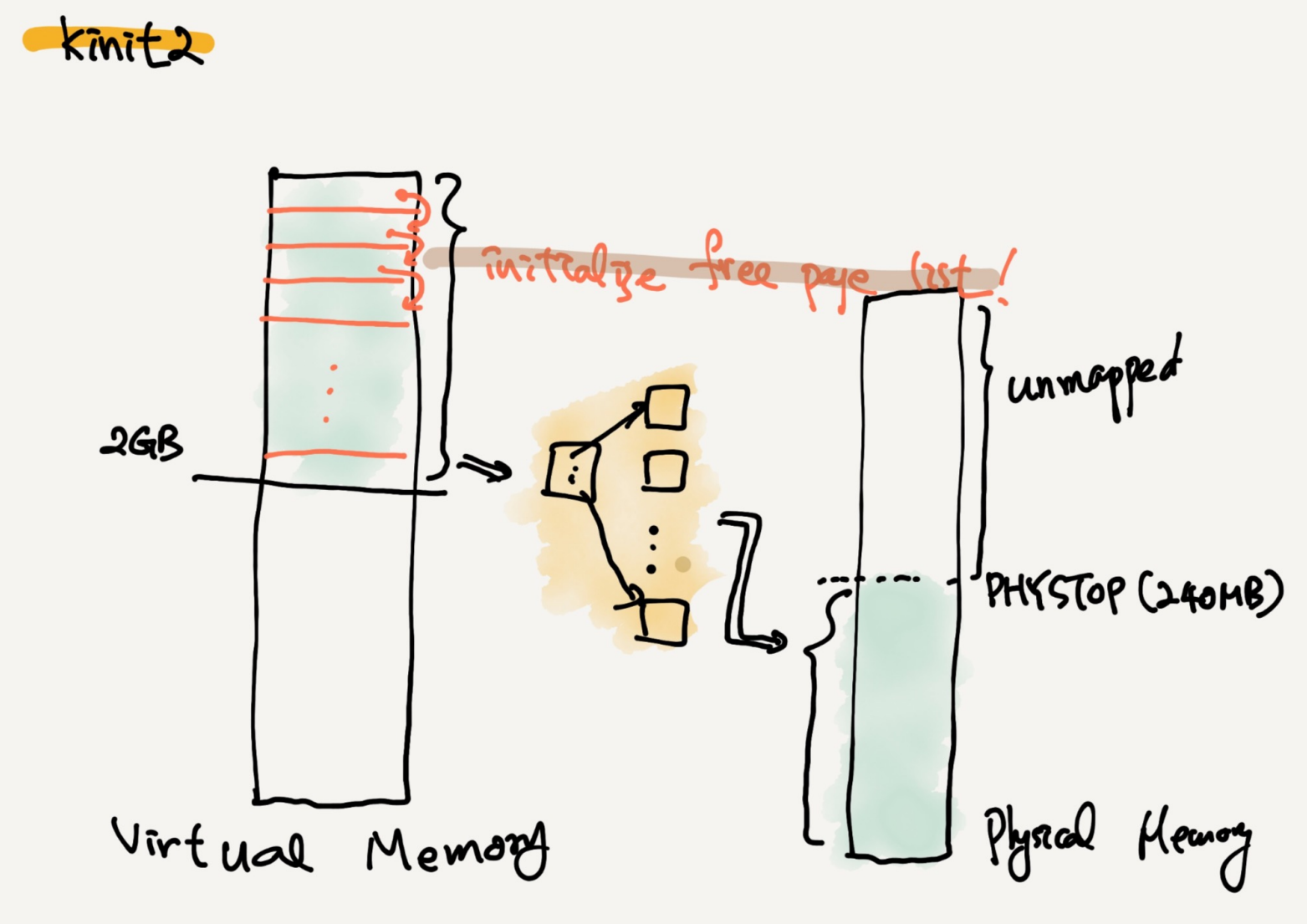
kinit1 vs. kinit2

```
1216 int
1217 main(void)
1218 {
1219 kinit1(end, P2V(4*1024*1024));        // phys page allocator
1220 kvmalloc();                          // kernel page table
…
1234 kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers
()
1235 userinit();                          // first user process
1236 mpmain();                            // finish this processor's setup
1237 }
```

Create free pages for the first 4 Mbyte region.

Create free pages for 4 Mbyte – PHYSTOP

kinit1 vs. kinit2 (Cont.)



Activating Lock

```
3130 void
3131 kinit1(void *vstart, void *vend)
3132 {
3133       initlock(&kmem.lock, "kmem");
3134       kmem.use_lock = 0;
3135       freerange(vstart, vend);
3136 }
```

```
3138 void
3139 kinit2(void *vstart, void *vend)
3140 {
3141       freerange(vstart, vend);
3142       kmem.use_lock = 1;
3143 }
```

Add to free page list

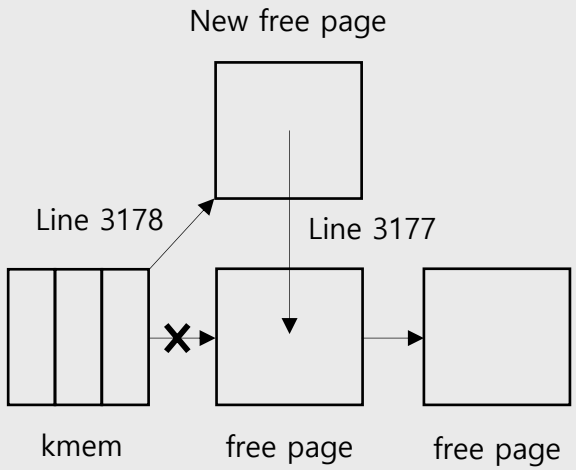
```
3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153       char *p;
3154       p = (char*)PGROUNDUP((uint)vstart);
3155       for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156             kfree(p);
3157 }
```

kfree( ): add a page to free list.(to head)

- Initialize the page to 1's.

kfree()

```
3163 void
3164 kfree(char *v)
3165 {
3166      struct run *r;
3167
3168      if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
3169           panic("kfree");
3170
3171      // Fill with junk to catch dangling refs.
3172      memset(v, 1, PGSIZE);
3173
3174      if(kmem.use_lock)
3175           acquire(&kmem.lock);
3176      r = (struct run*)v;
3177      r->next = kmem.freelist;
3178      kmem.freelist = r;
3179      if(kmem.use_lock)
3180           release(&kmem.lock);
3181 }
```



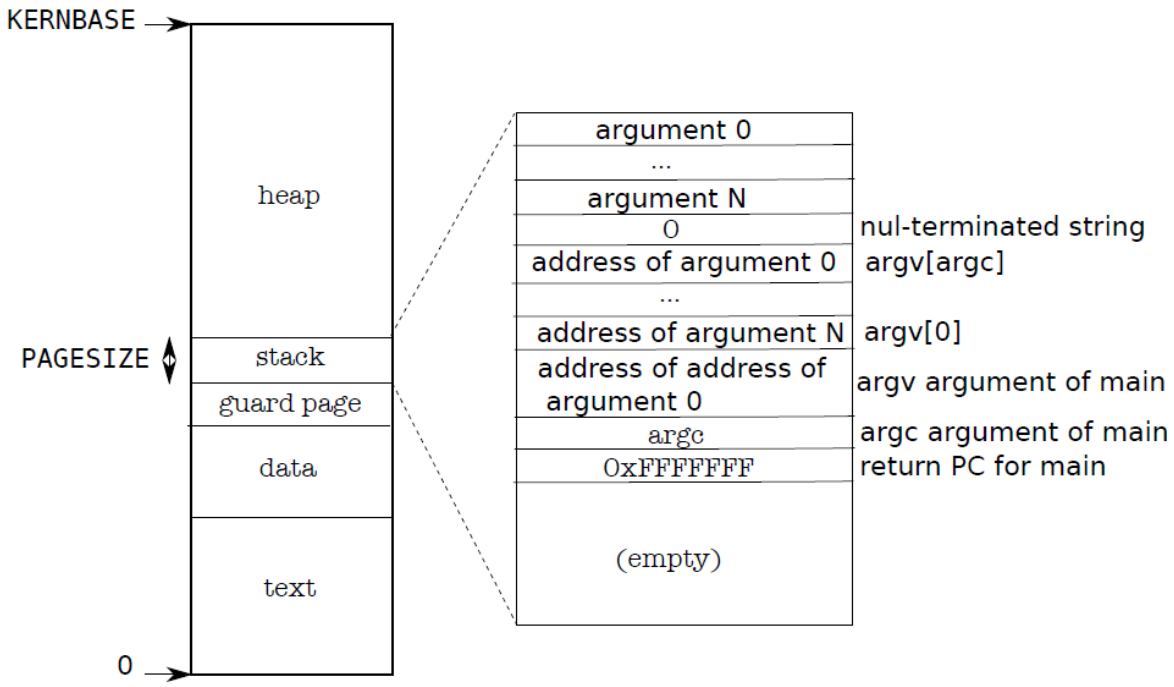
freelist and kmem

```
3115 struct run {
3116        struct run *next;
3117 };
```

```
3119 struct {
3120        struct spinlock lock;
3121        int use_lock;
3122        struct run *freelist;
3123 } kmem;
```

User address space organization in xv6

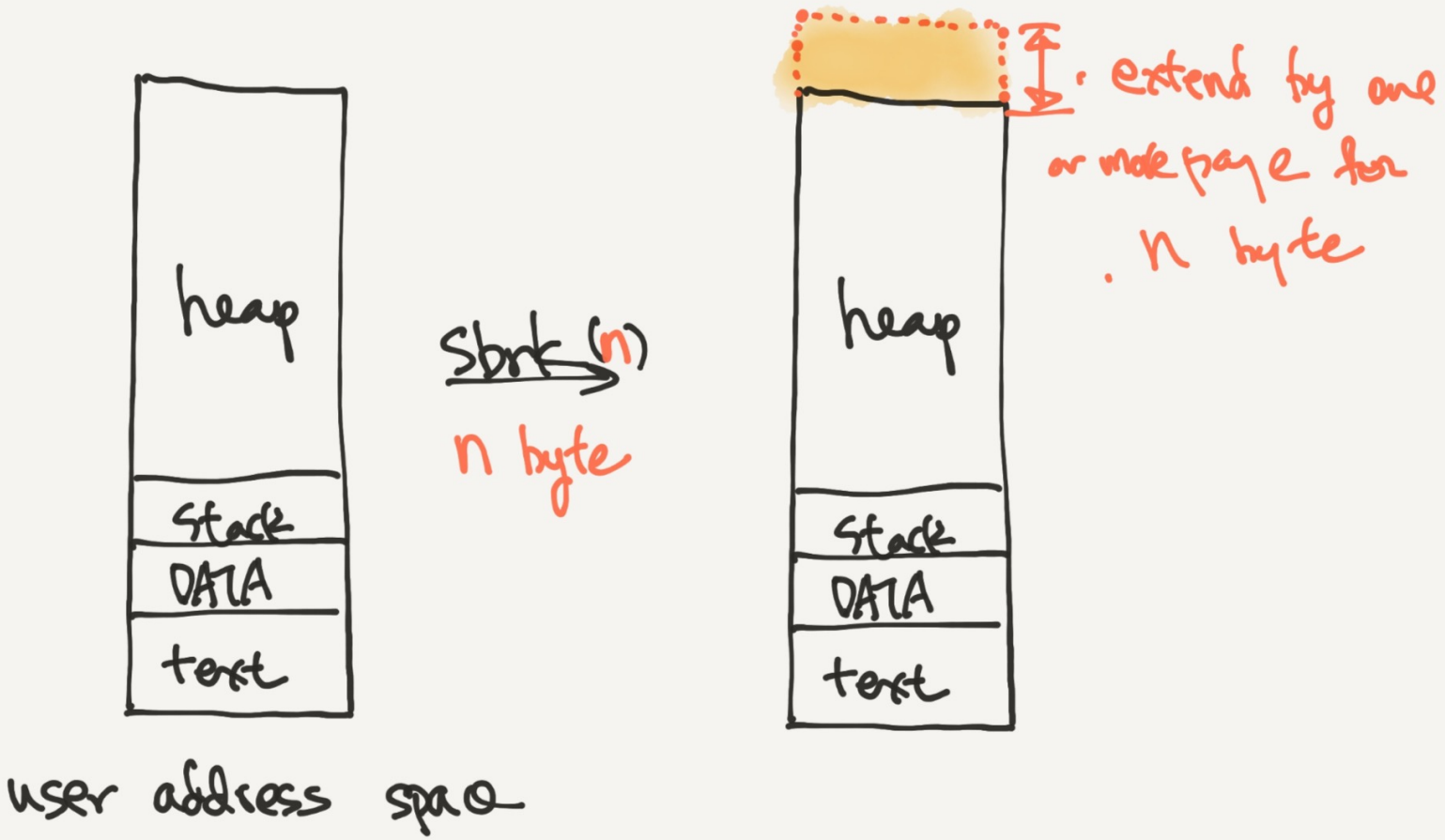
- text, data, stack and heap.
- Stack size is fixed. (single page) + guard page below the stack.
- heap can expand when the process calls `sbrk`.



Adjusting the heap size

- When the process needs more memory, it asks the kernel for more.
- `sbrk`: set breakpoint.
  - Breakpoint is the end of the heap.
  - `sbrk` can increase or decrease the end of the heap: extend or shrink the heap.
  - Extend the heap: allocate a new physical page, setup the page table for heap.
  - Shrink the heap: deallocate the physical page from the heap and add it to the free list.
- `growproc, allocuvm, deallocuvm`

Sbrk()


heap
stack
DATA
text
Sbrk (n)
n byte
heap
stack
DATA
text
: extend by one
or more page for
n byte
user address space

sbrk(growproc)

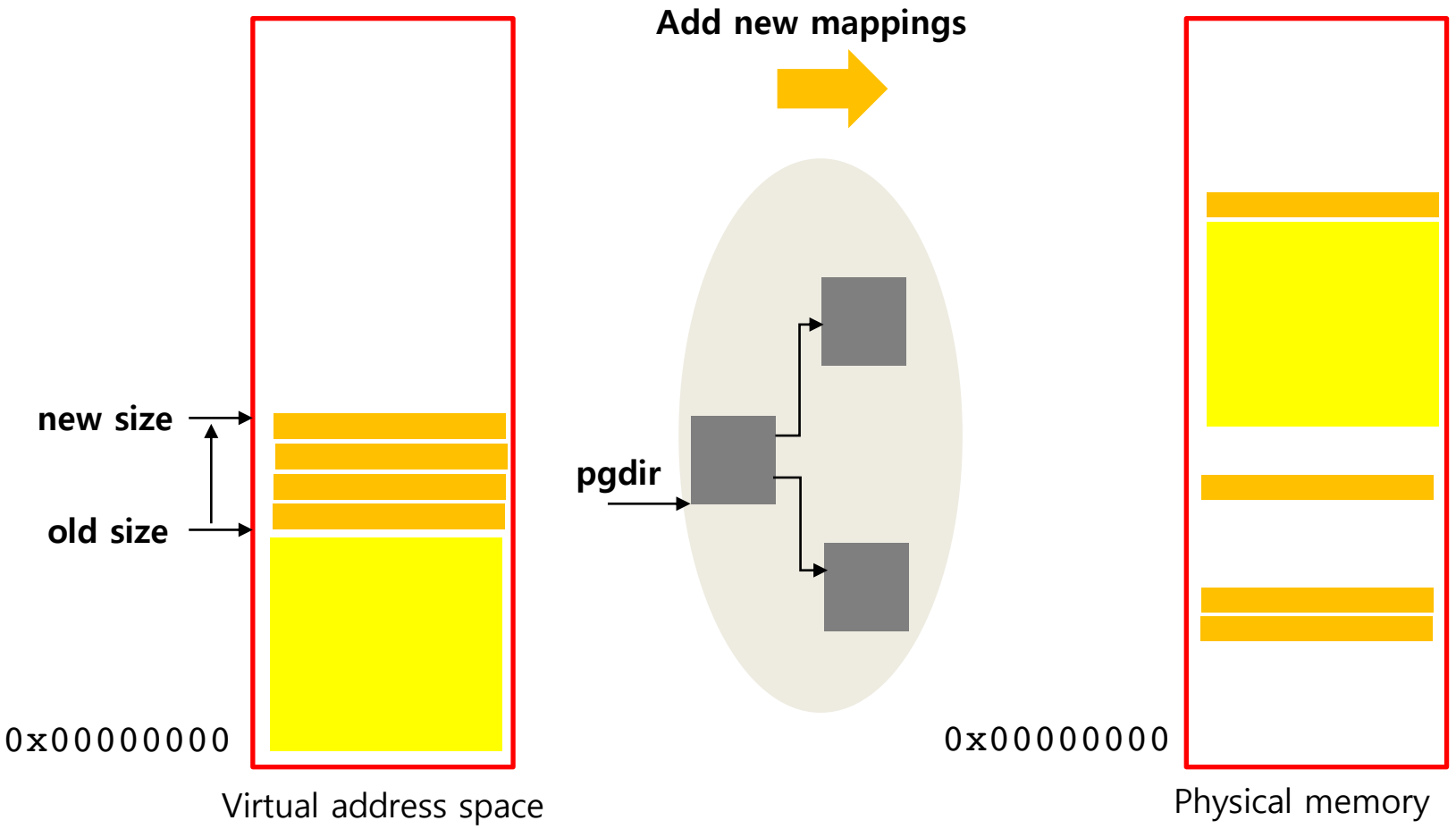
```
2557 int
2558 growproc(int n)
2559 {
2560      uint sz;
2561      struct proc *curproc = myproc();
2562
2563      sz = curproc->sz;
2564      if(n > 0){
2565           if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
2566                return -1;
2567      } else if(n < 0){
2568           if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
2569                return -1;
2570      }
2571      curproc->sz = sz;
2572     switchuvm(curproc);
2573      return 0;
2574 }
```

If n is greater than 0, add n bytes of memory

If n is less than 0, release n bytes of memory

allocuvm

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
```



sbrk(allocuvm)

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929      char *mem;
1930      uint a;
1931
1932      if(newsz >= KERNBASE)
1933           return 0;
1934      if(newsz < oldsz)
1935           return oldsz;
1936
1937      a = PGROUNDUP(oldsz);
1938      for(; a < newsz; a += PGSIZE){
1939           mem = kalloc();
1940           if(mem == 0){
1941                cprintf("allocuvm out of memory\n");
```

Allocating memory.

sbrk(allocuvm) (Cont.)

```
1942                deallocuvm(pgdir, newsz, oldsz);
1943                    return 0;
1944            }
1945        memset(mem, 0, PGSIZE);
1946        if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947            cprintf("allocuvm out of memory (2)\n");
1948            deallocuvm(pgdir, newsz, oldsz);
1949            kfree(mem);
1950            return 0;
1951            }
1952        }
1953        return newsz;
1954 }
```

Mapping virtual and physical addresses

sbrk(deallocuvm)

```
1960 int
1961 deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1962 {
1963      pte_t *pte;
1964      uint a, pa;
1965
1966      if(newsz >= oldsz)
1967           return oldsz;
1968
1969      a = PGROUNDUP(newsz);
1970      for(; a < oldsz; a += PGSIZE){
1971           pte = walkpgdir(pgdir, (char*)a, 0);
1972           if(!pte)
1973                a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
```

Return PTE of virtual address

sbrk(deallocuvm) (Cont.)

```
1974            else if((*pte & PTE_P) != 0){
1975                    pa = PTE_ADDR(*pte);
1976                    if(pa == 0)
1977                            panic("kfree");
1978                    char *v = P2V(pa);
1979                    kfree(v);
1980                    *pte = 0;
1981            }
1982    }
1983    return newsz;
1984 }
```

Add to free page list.