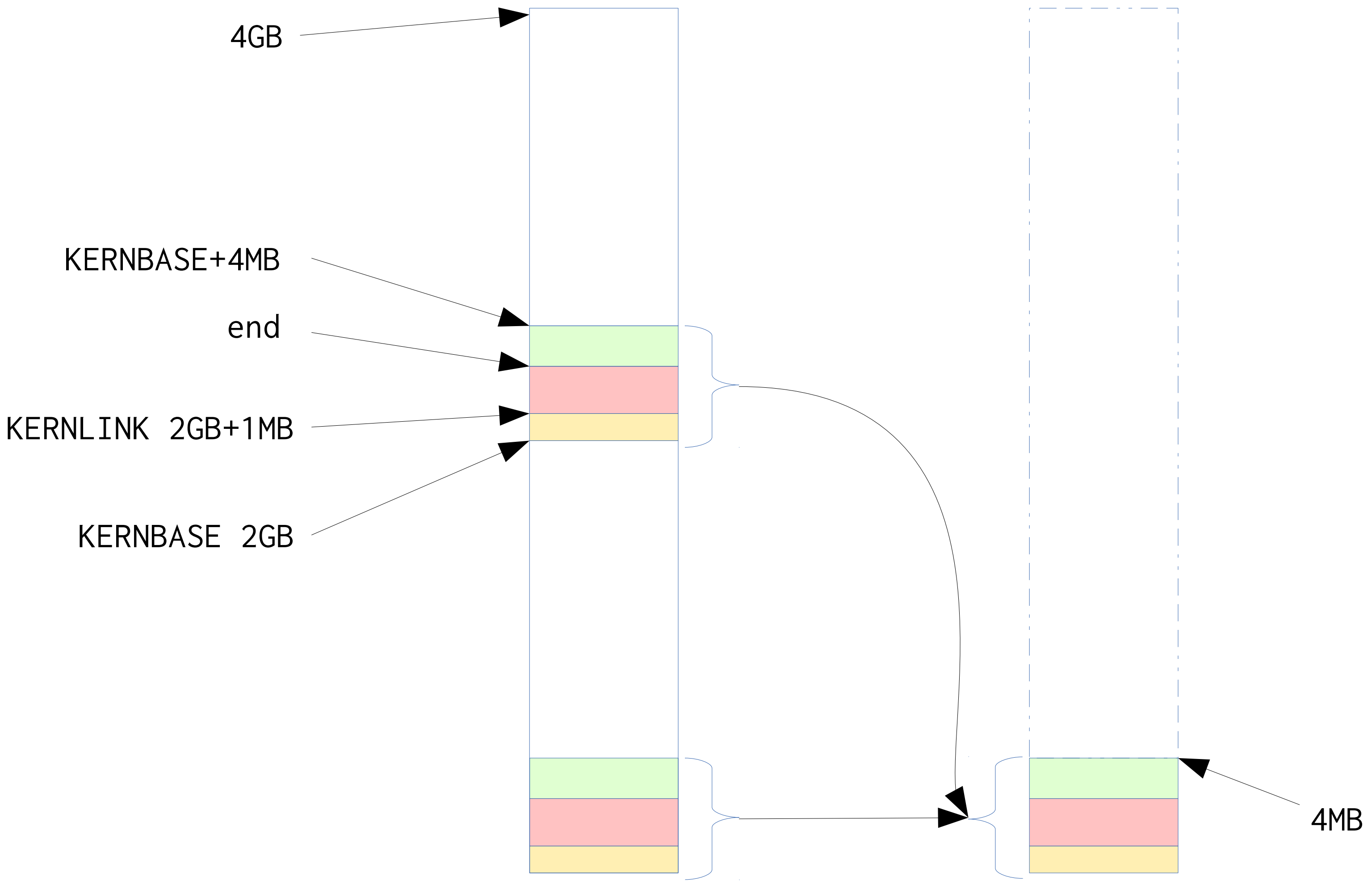
Operativni sistemi

dr.sc. Amer Hasanović

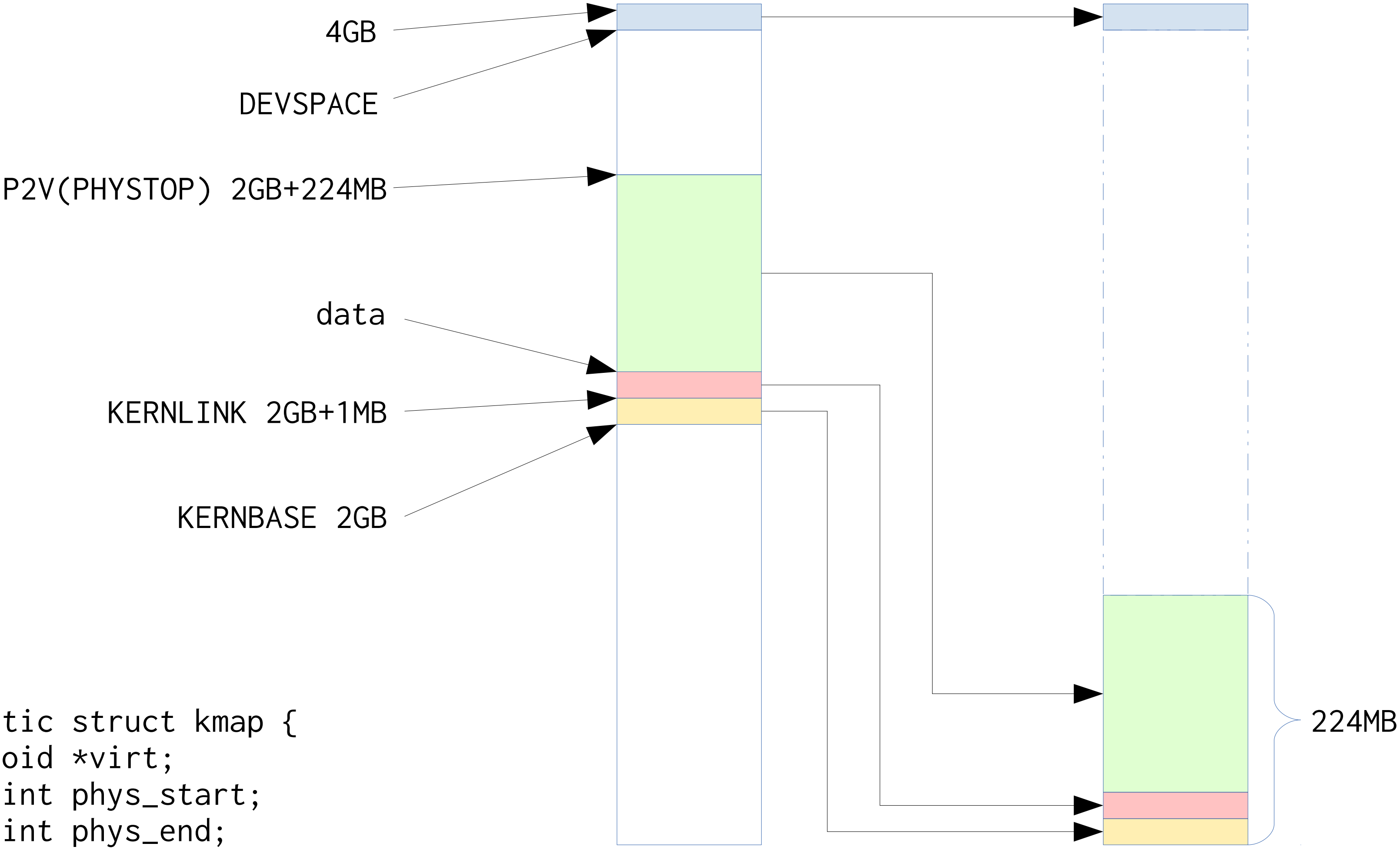
Inicijalni kernel virtuelni adresni (VA) prostor

- Nakon preuzimanja kontrole od bootloader-a, OS formira virtuelni adresni prostor na način da aktivira jednostavno straničenje:
  - dvije 4MB stranice iz virtuelne memorije mapiraju se na istu fizičku lokaciju:
    - `[0, 4MB) VA        → [0,4MB) PA`
    - `[2GB, 2GB+4MB) VA → [0,4MB) PA`
  - niz `entrypgdir` sadrži detalje ovog mapiranja.
- Formirano mapiranje koristi se samo tokom boot procesa.
- Da bi mogao da efikasno podrži više paralelnih procesa kao i da alocira prostor za potrebne data strukture, kernel mora kreirati novi virtuelni adresni prostor baziran na stranicama veličine 4KB.

Inicijalno VA → PA mapiranje



Kernel adresni prostor



```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,              EXTMEM,    PTE_W},
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
};
```

```
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)
```

- Da bi se kreirao željeni virtuelni adresni prostor potrebno je:
  - podijeliti 4 segmenta VA prostora na stranice veličine po 4KB
  - formirati MMU mapiranje VA → PA straničenjem:
    - kreirati i adevatno konfigurati tabelu direktorija tj PD:
      - PD staje u jednu stranicu od 4KB
    - kreirati i adekvatno konfigurirati tabele za translaciju stranica (PT) u aktivnim direktorijima:
      - i to, po jedan PT za svaki aktivni direktorij;
      - jedan PT staje u jednu stranicu od 4KB.
    - učitati PD adresu u `%cr3` nakon čega CPU počinje da koristi novokreirani VA prostor

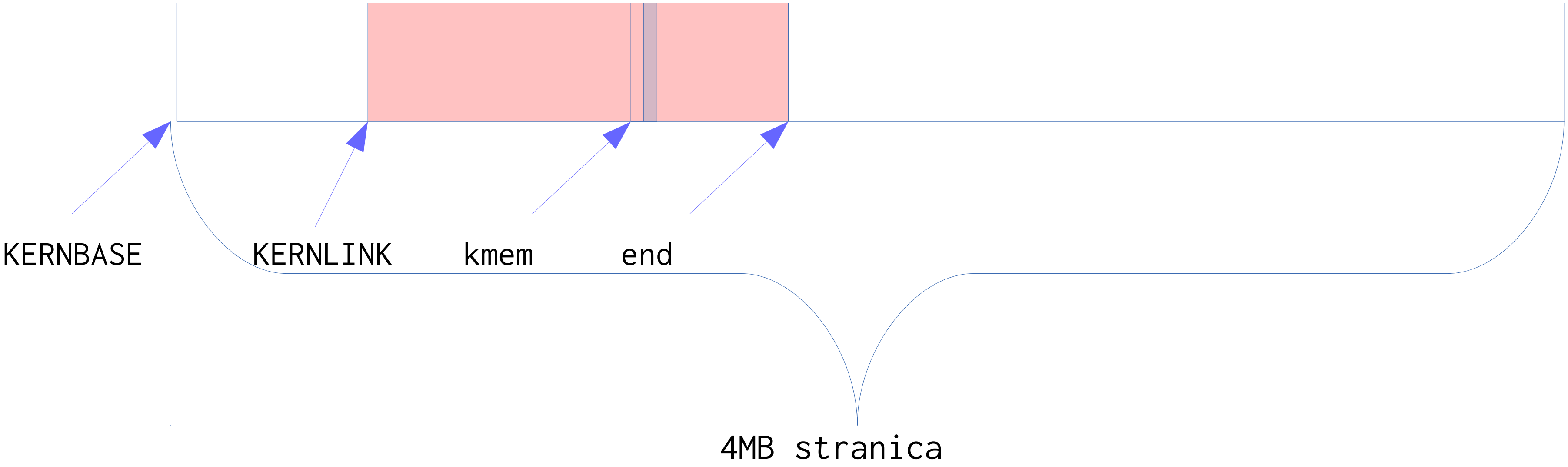
Alokator stranica

- Data struktura koja upravlja prostorom koji je neophodan za pohranu PD i PT-ova, kao i ostalih objekata koje dinamički koristi kernel:
  - inicijalno koristi slobodni prostor u intervalu `[end,KERNBASE+4MB)`, preostao nakon učitavanja kernela u memoriju (funkcija `kinit1` aktivira ovaj slobodni prostor)
  - pred kraj boot procesa upravljani prostor proširuje se do lokacije `KERNBASE+PHYSTOP;` (funkcija `kinit2` aktivira ovaj slobodni prostor)
  - organizira slobodan prostor kojim upravlja u manje komade memorije od po `4KB` (stranice).

- održava informaciju o slobodnim stranicama pomoću posebne linkane liste slobodnih stranica:
  - pointer na početak aktuelne liste slobodnih stranica drži se u globalnoj strukturi `kmem`;
  - na početku svake slobodne stranice nalazi se pointer na početak slijedeće slobodne stranice;
  - funkcije `kalloc` i `kfree` manipuliraju ovom strukturom.

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

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

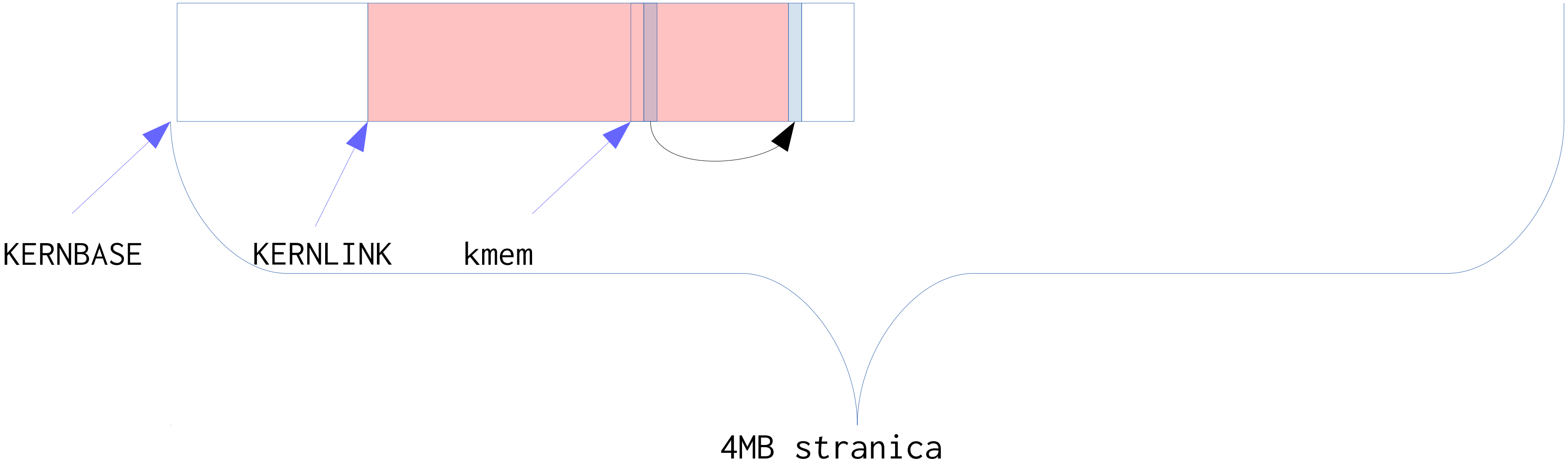


```
void kfree(char *v)
{
  struct run *r;
  memset(v, 1, PGSIZE);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
}
```

```
char* kalloc(void)
{
  struct run *r;
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  return (char*)r;
}
```

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

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

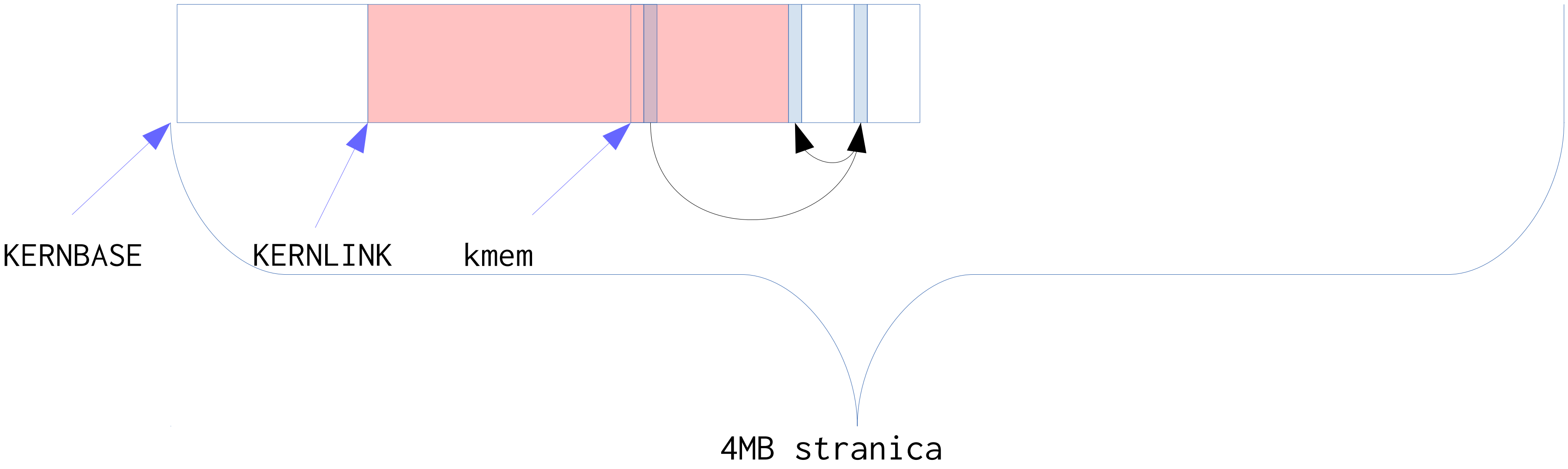


```
void kfree(char *v)
{
  struct run *r;
  memset(v, 1, PGSIZE);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
}
```

Prvi poziv: `kfree(end);`

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

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

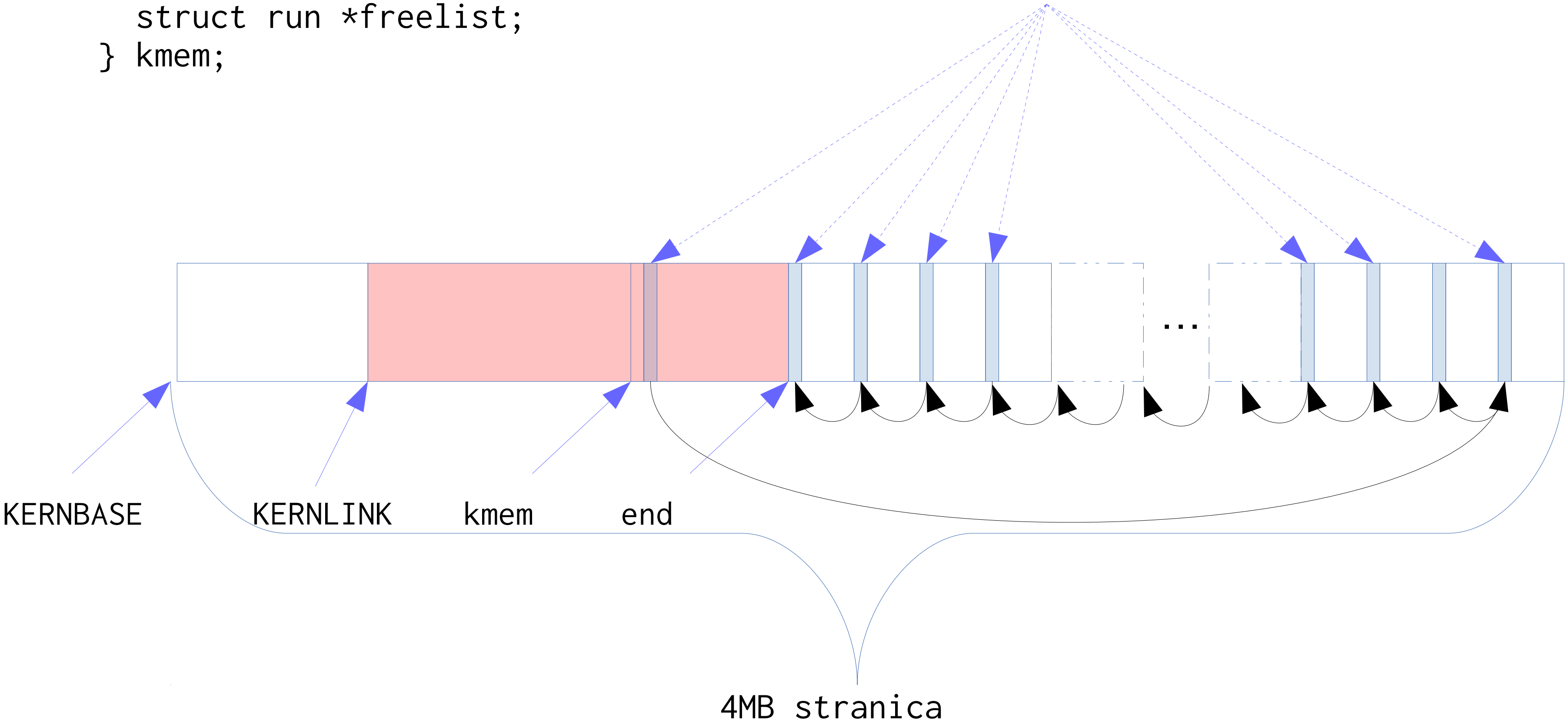


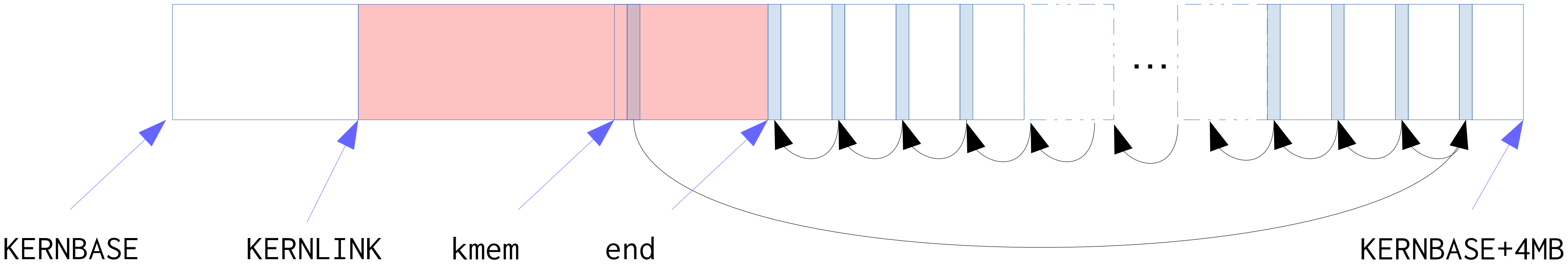
```
void kfree(char *v)
{
  struct run *r;
  memset(v, 1, PGSIZE);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
}
```

Drugi poziv: `kfree(end+PGSIZE);`

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

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





```
kinit1(end, P2V(4*1024*1024));
```

```
void kinit1(void *vstart, void *vend)
{
  //...
  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);
}
```

Formiranje kernel AP

```
int main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}
```

```
int main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
  picinit();       // interrupt controller
  ioapicinit();    // another interrupt controller
  consoleinit();   // I/O devices & their interrupts
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  iinit();         // inode cache
  ideinit();       // disk
  if(!ismp)
    timerinit();   // uniprocessor timer
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  // Finish setting up this processor in mpmain.
  mpmain();
}
```

```
void kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

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

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;

  memset(pgdir, 0, PGSIZE);
  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 0;
  return pgdir;
}
```

```
void kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

```
static struct kmap {
  void *virt;
  uint phys_start;
  uint phys_end;
  int perm;
} kmap[] = {
 { (void*)KERNBASE, 0,             EXTMEM,    PTE_W},
 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0},
 { (void*)data,     V2P(data),     PHYSTOP,   PTE_W},
 { (void*)DEVSPACE, DEVSPACE,      0,         PTE_W},
};
```

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

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;

  memset(pgdir, 0, PGSIZE);
  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 0;
  return pgdir;
}
```

```
void kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

```
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;
}
```

```
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)];
}
```

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

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;

  memset(pgdir, 0, PGSIZE);
  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 0;
  return pgdir;
}
```

```
void kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```

```
void switchkvm(void)
{
  lcr3(v2p(kpgdir));
}
```

```
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;
}
```