Código 1

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

Código 2

```
void * kalloc(void){
  struct run *r;

  acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  release(&kmem.lock);

  if(r)
    memset((char*)r, 5, PGSIZE);
  return (void*)r;
}
```

Código 3

```
void kfree(void *pa) {
  struct run *r;

  ...

  r = (struct run*)pa;

  acquire(&kmem.lock);
  r->next = kmem.freelist;
  kmem.freelist = r;
  release(&kmem.lock);
}
```

Código 4

```
#define PAGE_SIZE 4096  // Tamaño de la página: 4KB
#define PTE_COUNT 1024  // Número de entradas por tabla (2^10)

typedef uint32_t pte_t;  // Definimos pte_t como un entero de 32
bits
typedef uint32_t uint32;

// Función para mapear una página virtual a una frame de memoria
// física
int mappage(pte_t *pagetable, uint32 va) {
  // Máscaras para extraer los índices de las tablas de página
  uint32 idx_l1 = (va >> 22) & 0x3FF;  // Los primeros 10 bits
  uint32 idx_l2 = (va >> 12) & 0x3FF;  // Los siguientes 10 bits

  // Obtenemos la entrada de la tabla de primer nivel (nivel 1)
  pte_t *l2_pagetable = (pte_t *)pte_pa(&pagetable[idx_l1]);

  // Si la tabla de nivel 2 no existe, la creamos
  if (l2_pagetable == 0) {
    // Asignamos un nuevo frame de memoria física para la tabla
de
    // segundo nivel
    l2_pagetable = (pte_t *)kalloc();
    if (l2_pagetable == 0) {
      // Error: No se pudo asignar memoria
      return -1;
    }
    // Inicializamos la entrada de la tabla de nivel 1 para
apuntar a
    // la nueva tabla de nivel 2
    pte_init(&pagetable[idx_l1], (uint32)l2_pagetable);
  }

  // Verificamos si la página ya está mapeada
  if (pte_pa(&l2_pagetable[idx_l2]) != 0) {
    // Error: La página ya está mapeada
    return -1;
  }

  // Asignamos un nuevo frame de memoria física para la página
  void *frame = kalloc();
  if (frame == 0) {
    // Error: No se pudo asignar memoria
    return -1;
  }

  // Inicializamos la entrada de la tabla de nivel 2 para apuntar
al
  // frame de memoria física
  pte_init(&l2_pagetable[idx_l2], (uint32)frame);

  // Éxito
  return 0;
```

```
}
```

Código 5

```
pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)
{
  if(va >= MAXVA)
    panic("walk");

  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];

    if(*pte & PTE_V) { // Si la pagina esta presente
      pagetable = (pagetable_t)PTE2PA(*pte);

    } else {
      if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
        return 0;
      memset(pagetable, 0, PGSIZE);
      *pte = PA2PTE(pagetable) | PTE_V;
    }
  }
  return &pagetable[PX(0, va)];
}
```

Código 6

```
while(true) {
     int transferencia = nextTransferencia();
     obtener(lock)
     int saldo = obtenerSaldo();
     saldo += transferencia
     guardarSaldo(saldo)
     dejar(lock)
}
```

Código 7

```
struct spinlock {

  uint locked;

};
```

Código 8

```
void
acquire(struct spinlock *lk)
{
[...]

  while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
    ;

[...]
}
```

Código 9

```
void
release(struct spinlock *lk)
{
[...]

  __sync_lock_release(&lk->locked);

[...]
}
```

Código 10

```
struct sleeplock {
 uint locked;       // Is the lock held?
 struct spinlock lk; // spinlock protecting this sleep lock
};
```

Código 11

```
void
acquiresleep(struct sleeplock *lk)
{
 acquire(&lk->lk);

 while (lk->locked) {
   // Atomically release lock and sleep on chan.
   // Reacquires lock when awakened.
   sleep(lk, &lk->lk);
 }
 lk->locked = 1;

 release(&lk->lk);
}
```

Código 12

```
void
releasesleep(struct sleeplock *lk)
{
 acquire(&lk->lk);

 lk->locked = 0;
 // Wake up all processes sleeping on chan.
 wakeup(lk);

 release(&lk->lk);
}
```