ICS143A: Principles of Operating Systems

Final recap, sample questions

Anton Burtsev
March, 2017

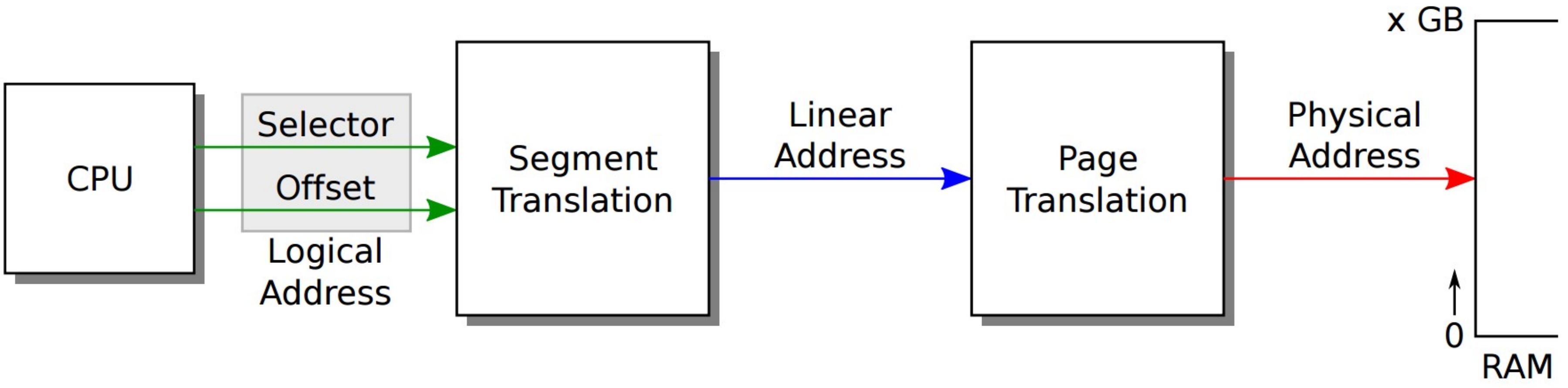
What is operating system?

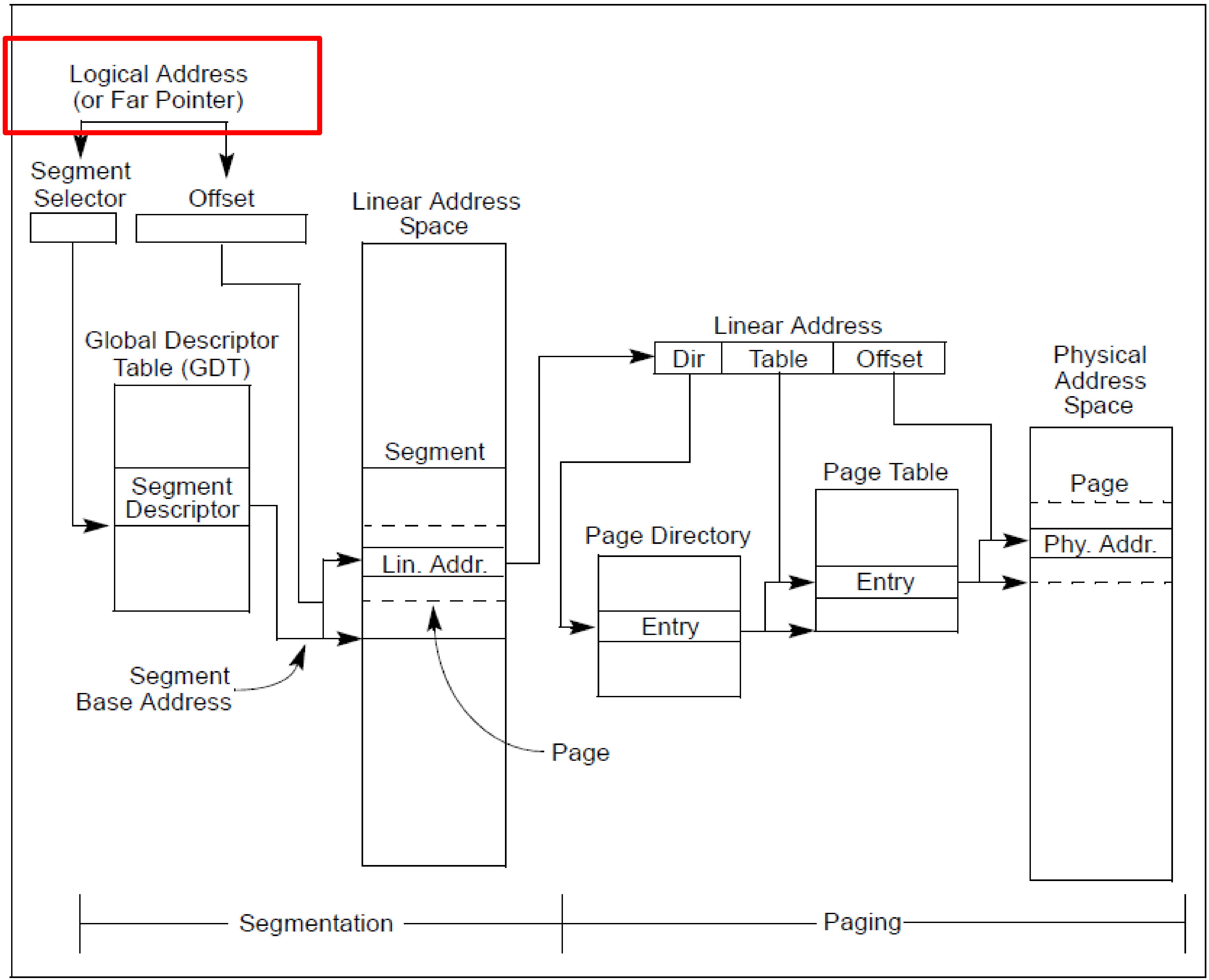
What is a process?

What mechanisms are involved into implementation of a process?

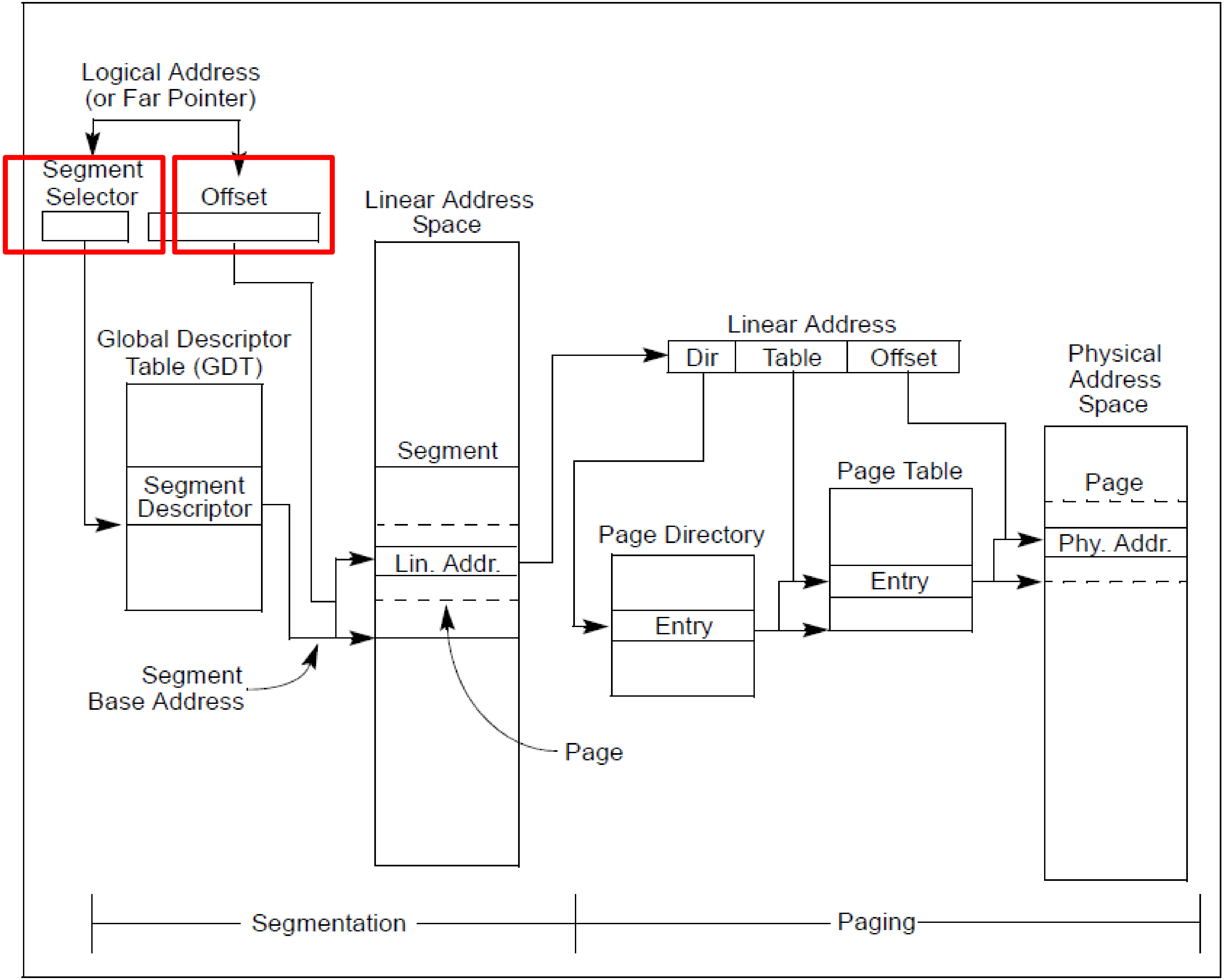
Describe the x86 address translation pipeline (draw figure), explain stages.

Address translation

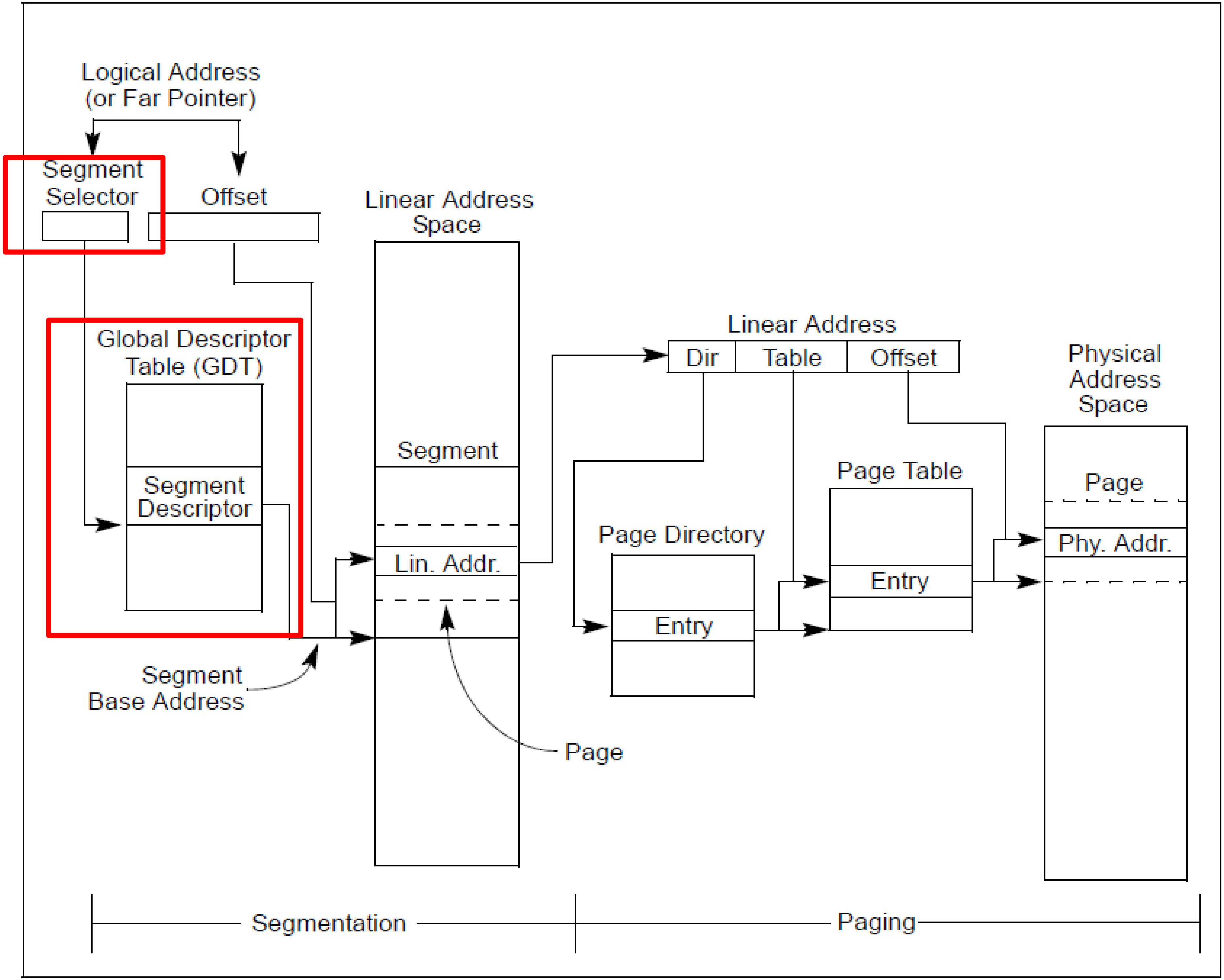


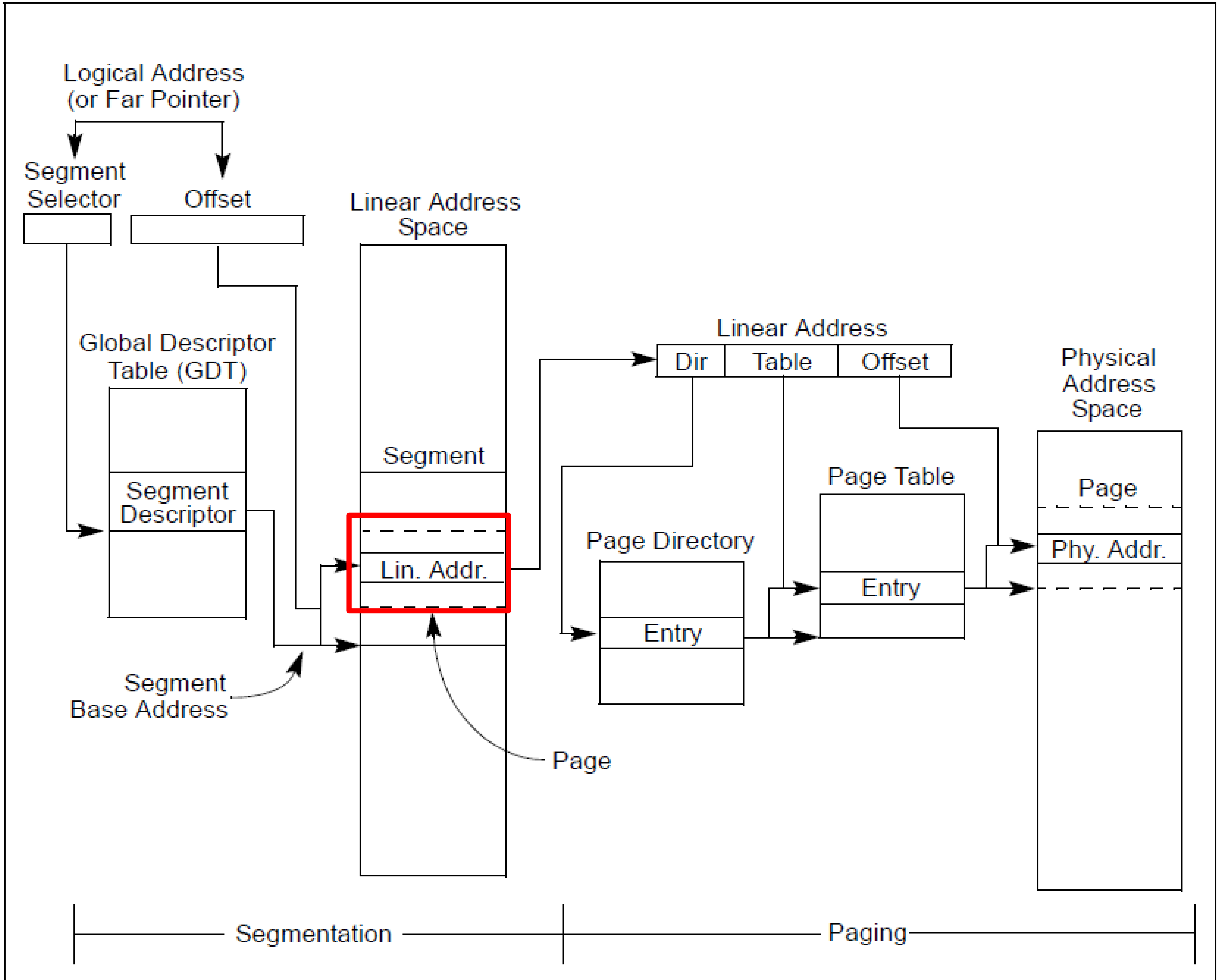


Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging

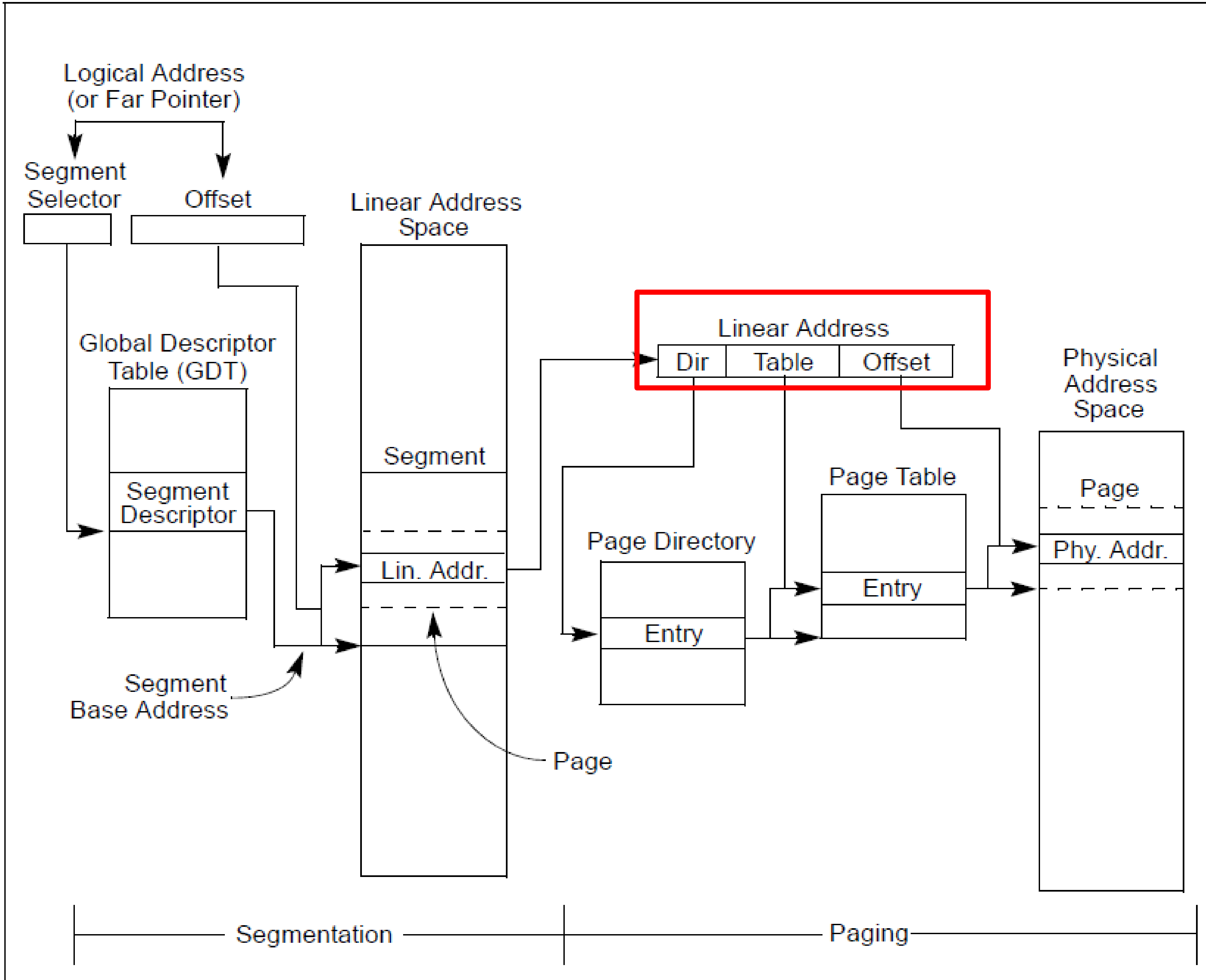


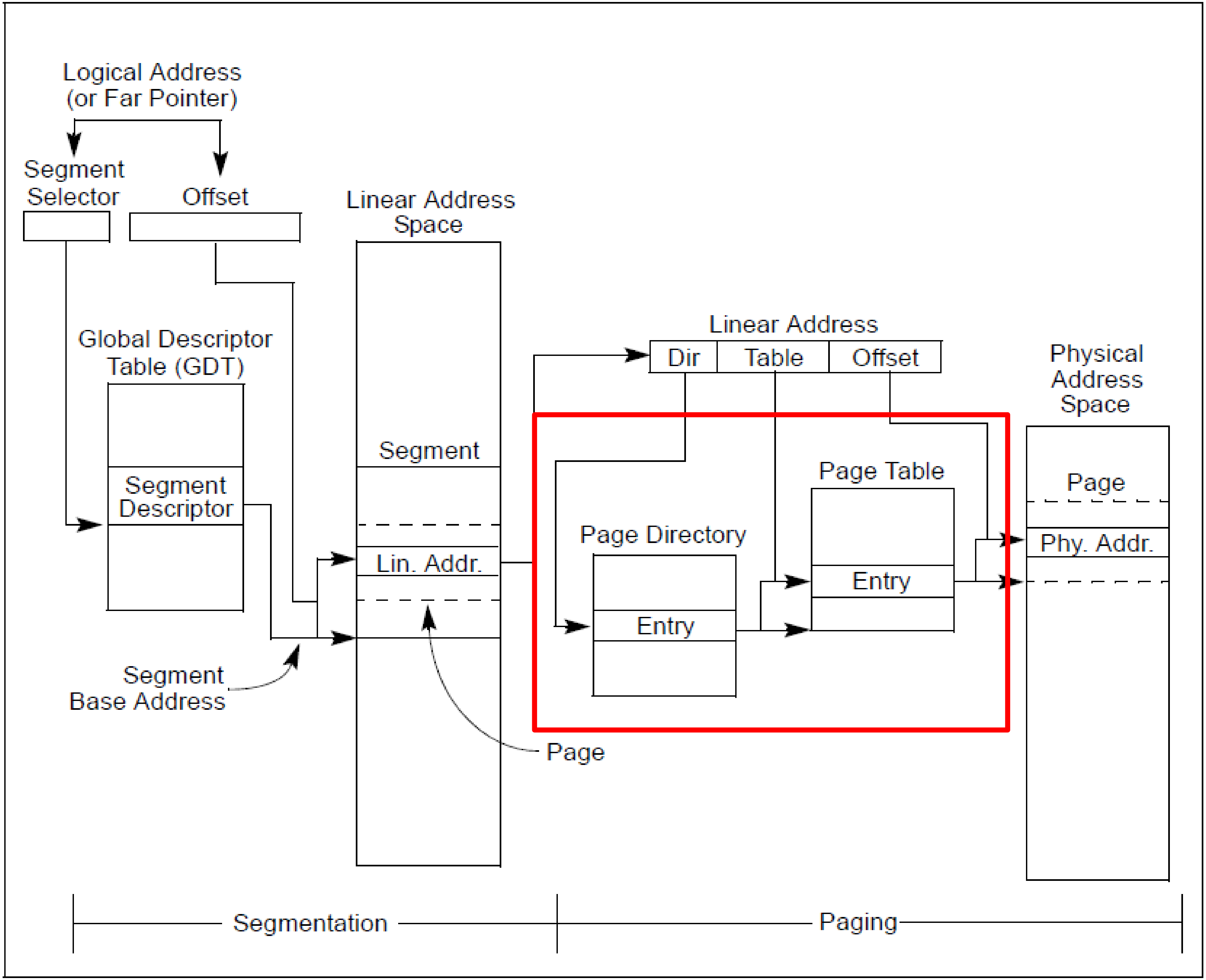
Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

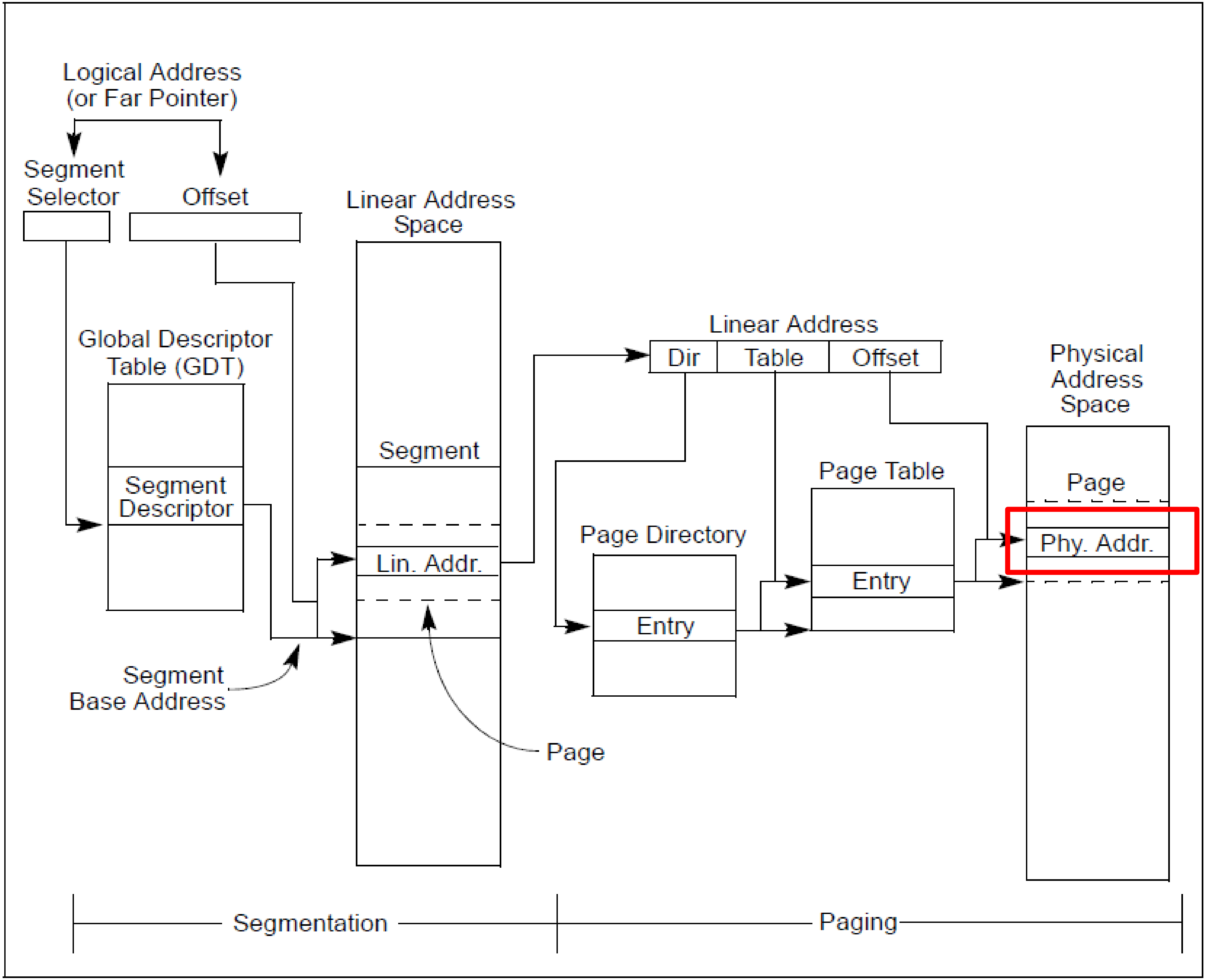


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

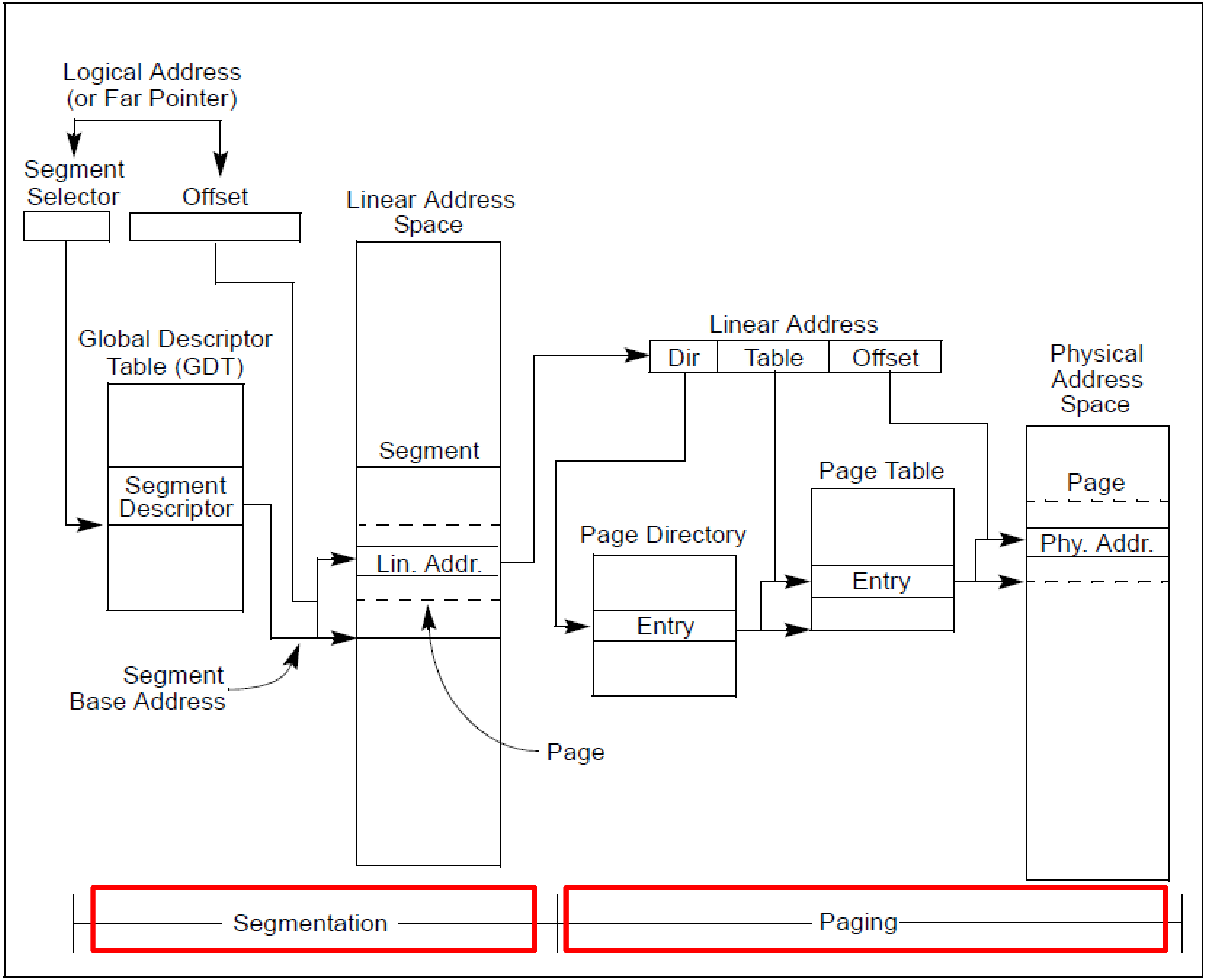

Logical Address
(or Far Pointer)
Segment Selector
Offset
Linear Address Space
Global Descriptor Table (GDT)
Segment Descriptor
Segment
Lin. Addr.
Segment Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical Address Space
Page
Phy. Addr.
Segmentation
Paging



Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging



Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

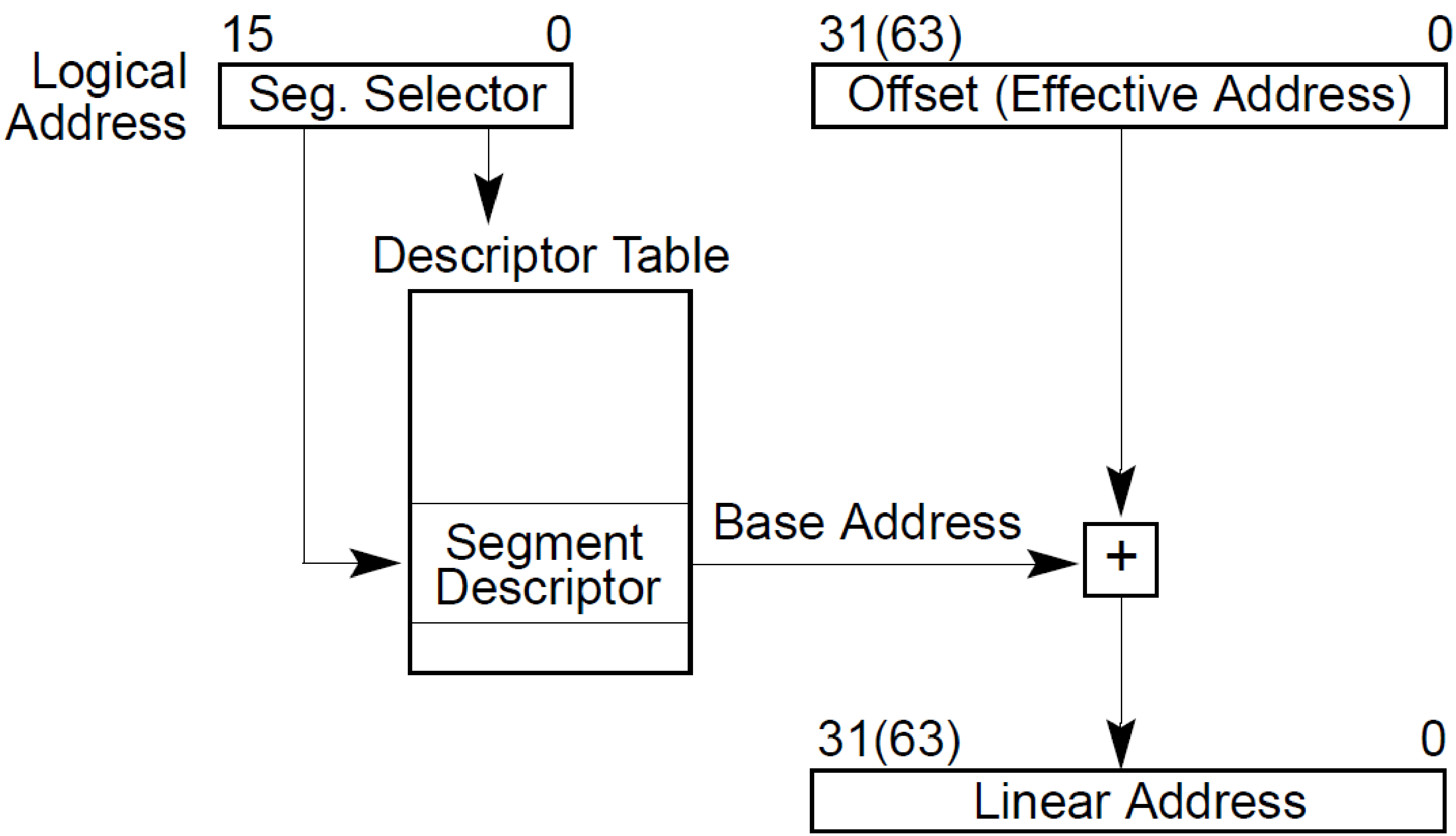


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
Global Descriptor
Table (GDT)
Segment
Descriptor
Segment
Lin. Addr.
Segment
Base Address
Page
Linear Address
Dir
Table
Offset
Page Directory
Entry
Page Table
Entry
Physical
Address
Space
Page
Phy. Addr.
Segmentation
Paging

What is the linear address? What address is in the registers, e.g., in %eax?

Logical and linear addresses

- Segment selector (16 bit) + offset (32 bit)



What segments do the following instructions use? push, jump, mov

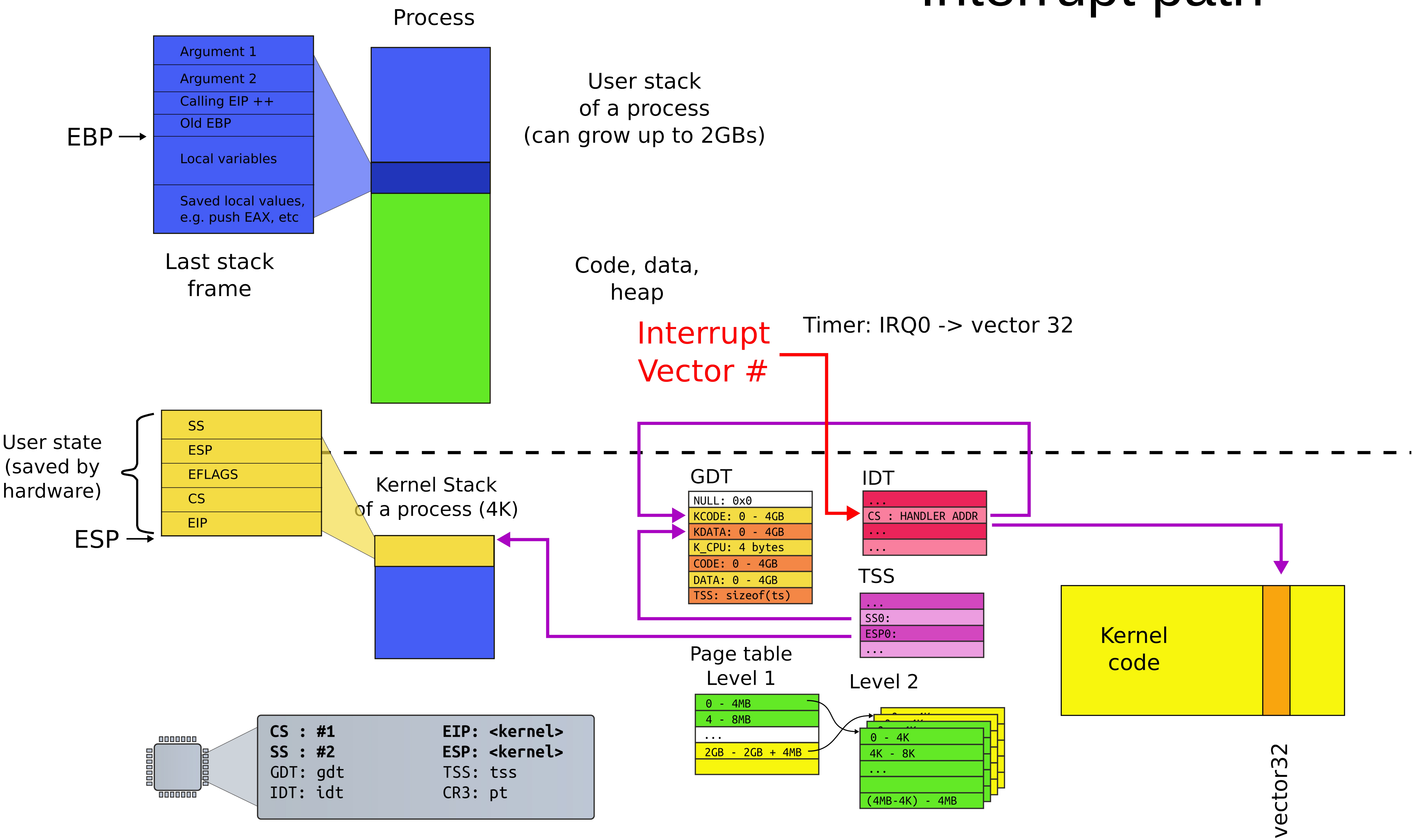
What's on the stack? Describe layout of a stack and how it changes during function invocation?

Example stack

```
:    :
| 10 | [ebp + 16] (3rd function argument)
|  5 | [ebp + 12] (2nd argument)
|  2 | [ebp + 8]  (1st argument)
| RA | [ebp + 4]  (return address)
| FP | [ebp]      (old ebp value)
|    | [ebp - 4]  (1st local variable)
:    :
:    :
|    | [ebp - X]  (esp - the current stack pointer)
```

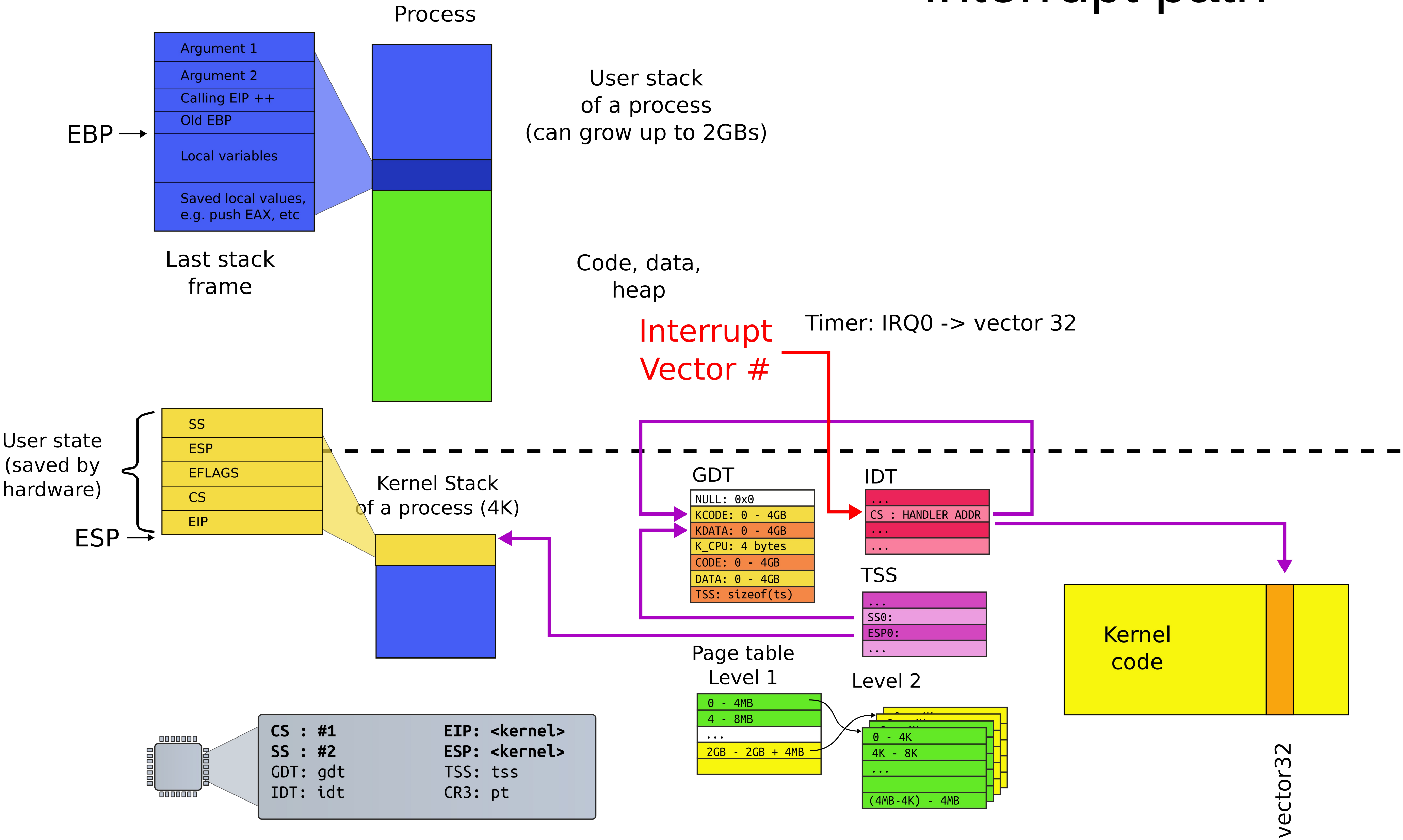
Describe the steps and data structures involved into a user to kernel transition (draw diagrams)

Interrupt path



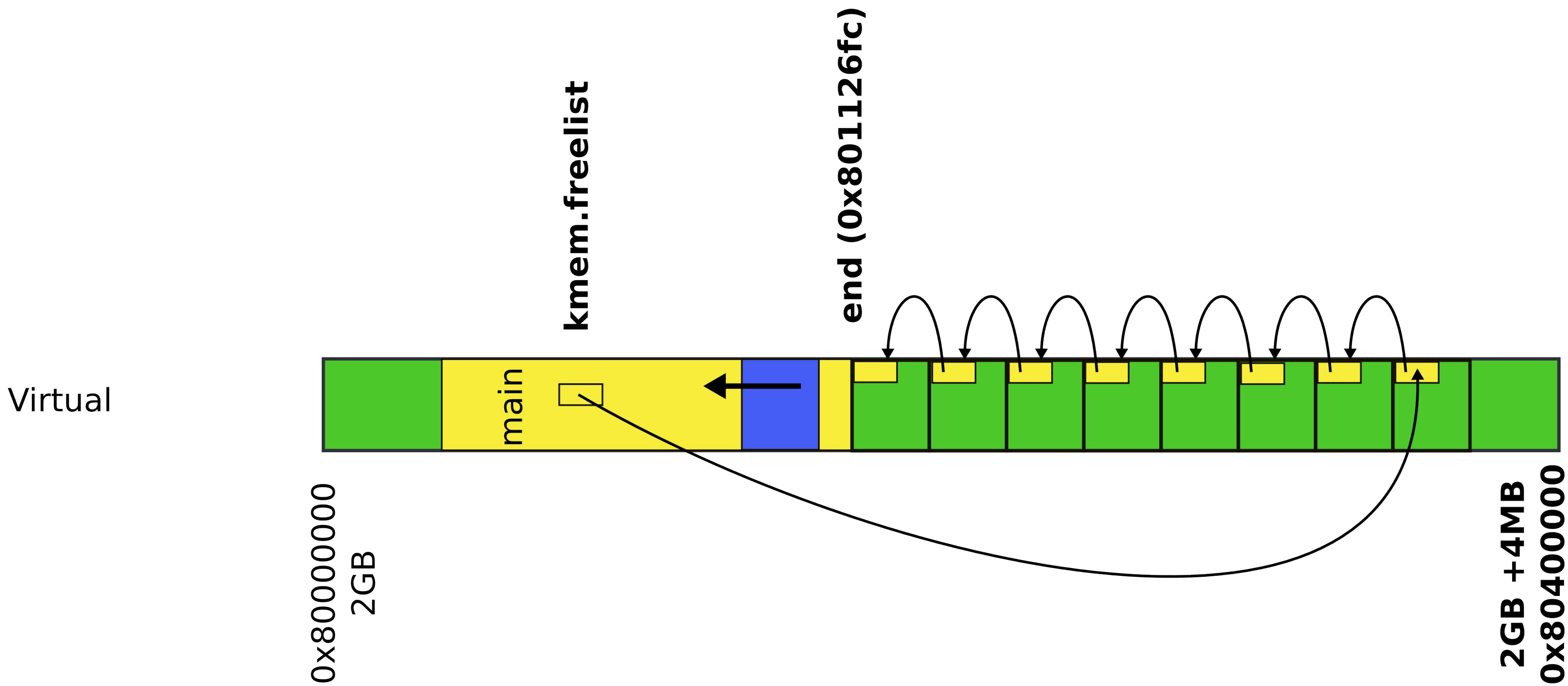
Which stack is used for execution of an interrupt handler? How does hardware find it?

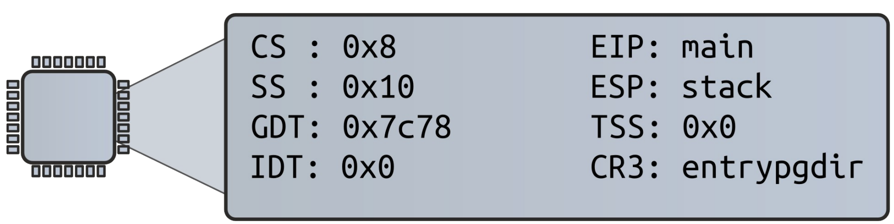
Interrupt path



Describe organization of the memory allocator in xv6?

Physical page allocator





Protected Mode

Where did free memory came from?

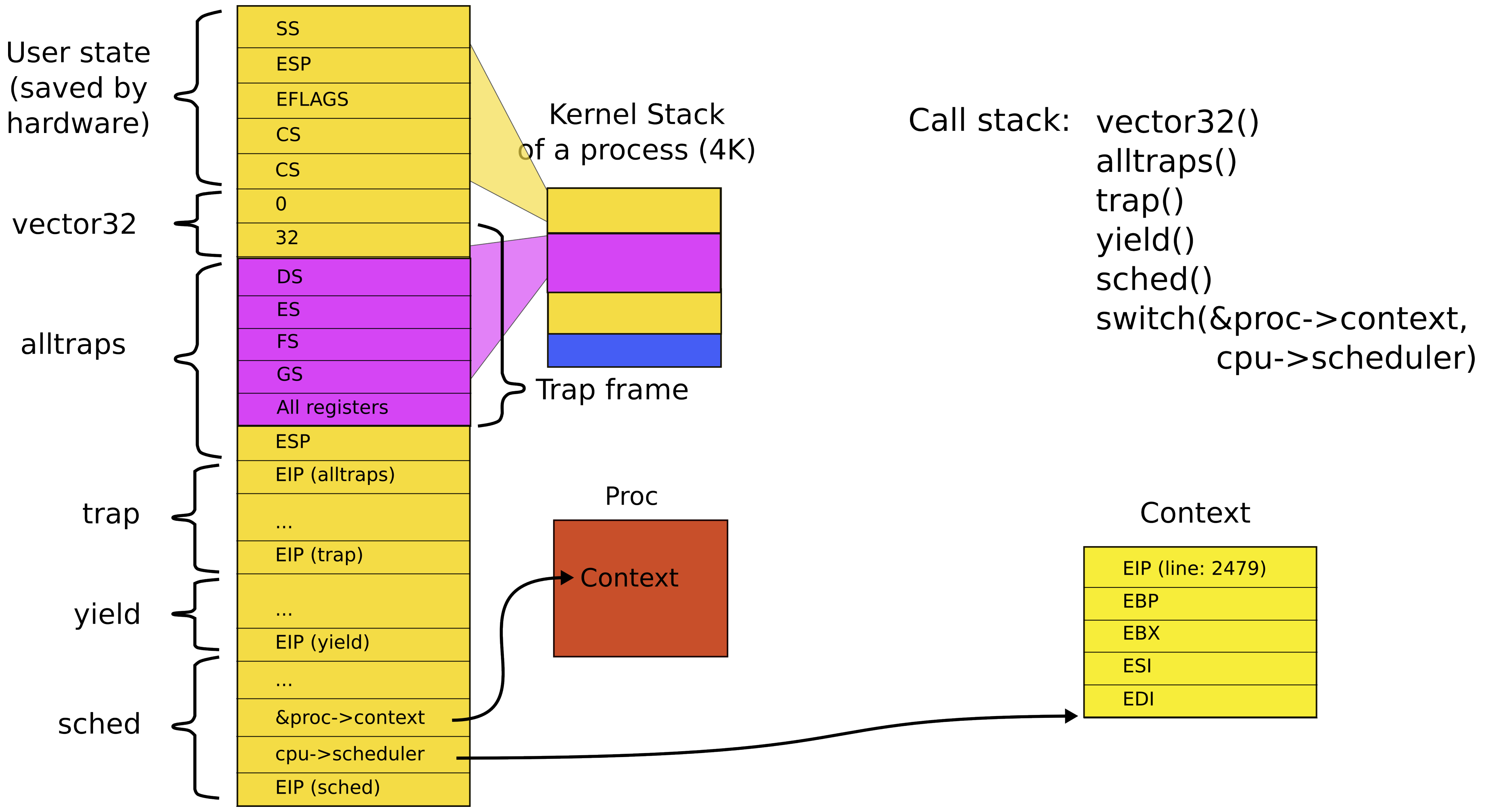
How do we switch between processes?

swtch()

```
swtch:
movl 4(%esp), %eax
movl 8(%esp), %edx

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

# Switch stacksh
movl %esp, (%eax)
movl %edx, %esp

# Load new callee-save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

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

Stack inside swtch()



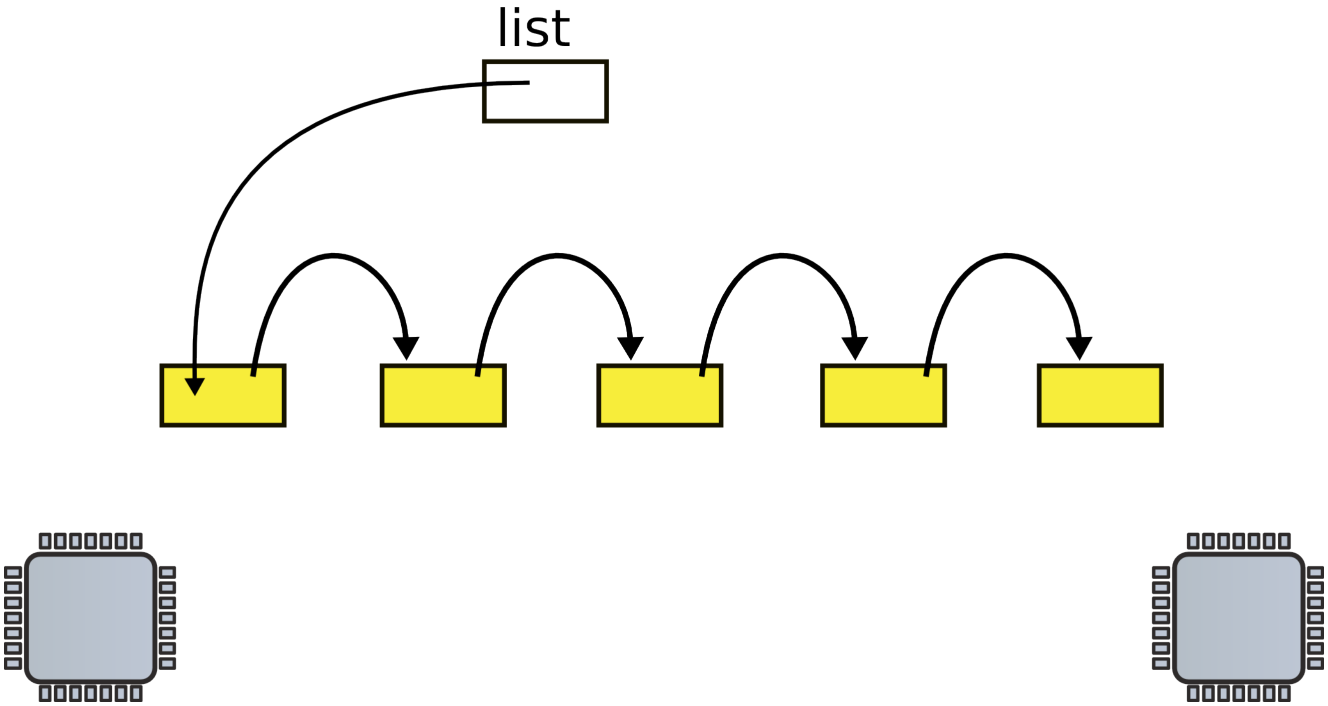
What is the interface between the kernel and user-level processes?

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    proc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
    proc->pid, proc->name, num);
    proc->tf->eax = -1;
  }
}
```

```
void
syscall(void)
{
  int num;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // proc->tf->eax = syscalls[num]();
    proc->tf->esp -= 4;
    *(int*)ptoc->tf->esp = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    // proc->tf->eax = -1;
    proc->tf->esp -= 4;
    *(int*)ptoc->tf->esp = -1;
  }
}
```

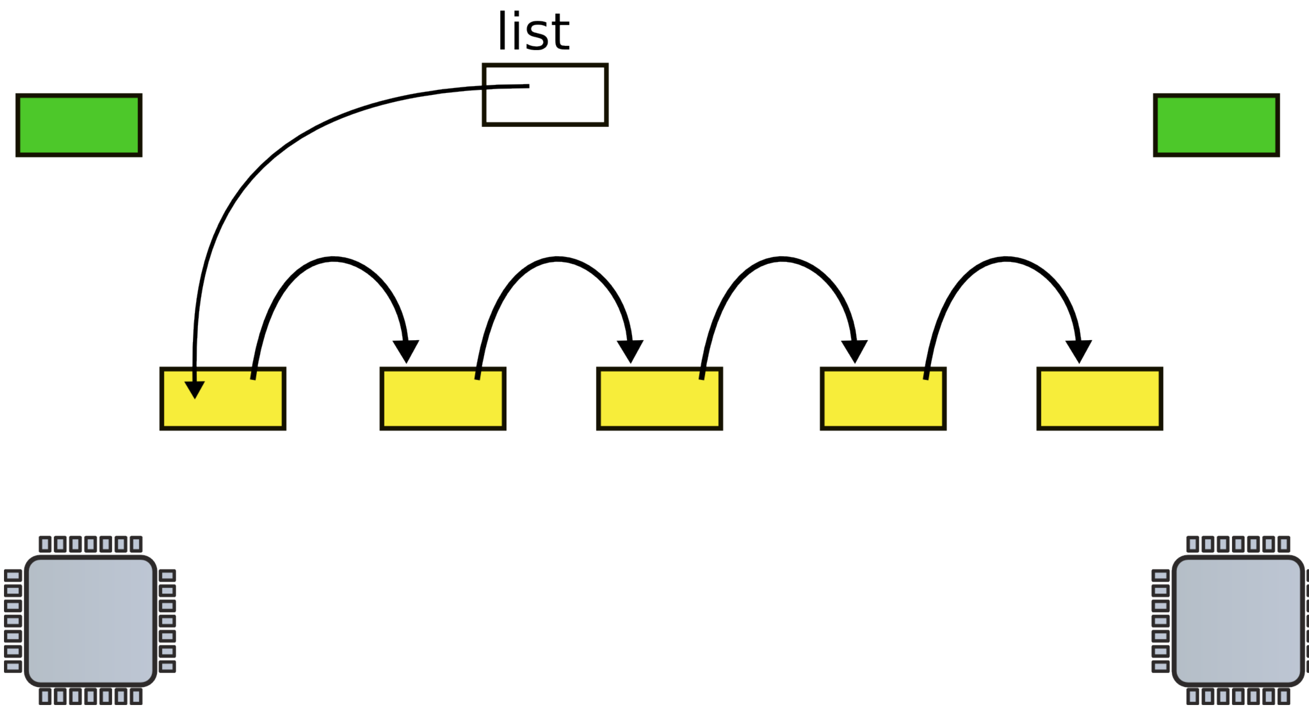
Why do we need locks?

Request queue (e.g. incoming network packets)

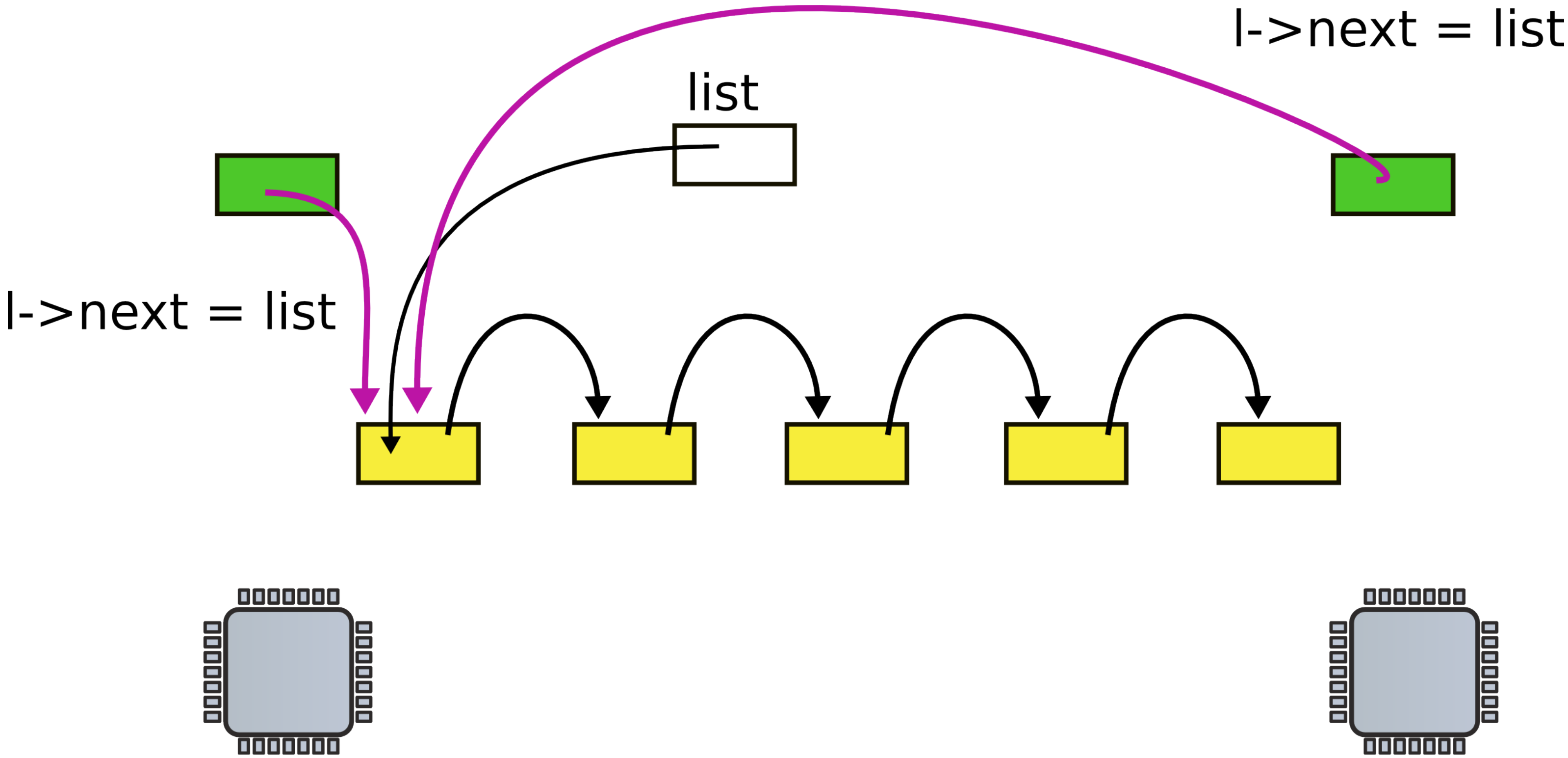


- Linked list, list is pointer to the first element

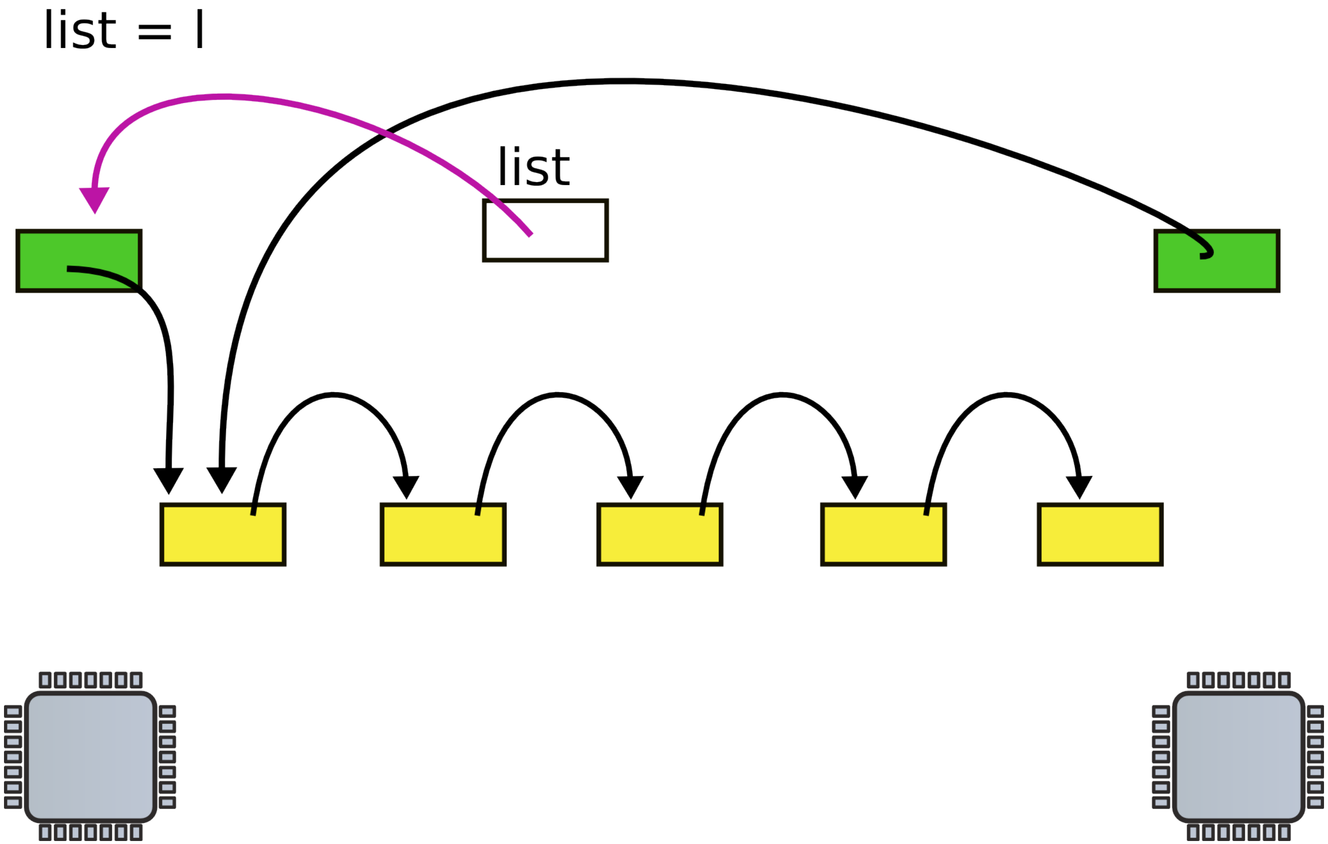
CPU 1 and 2 allocate new request



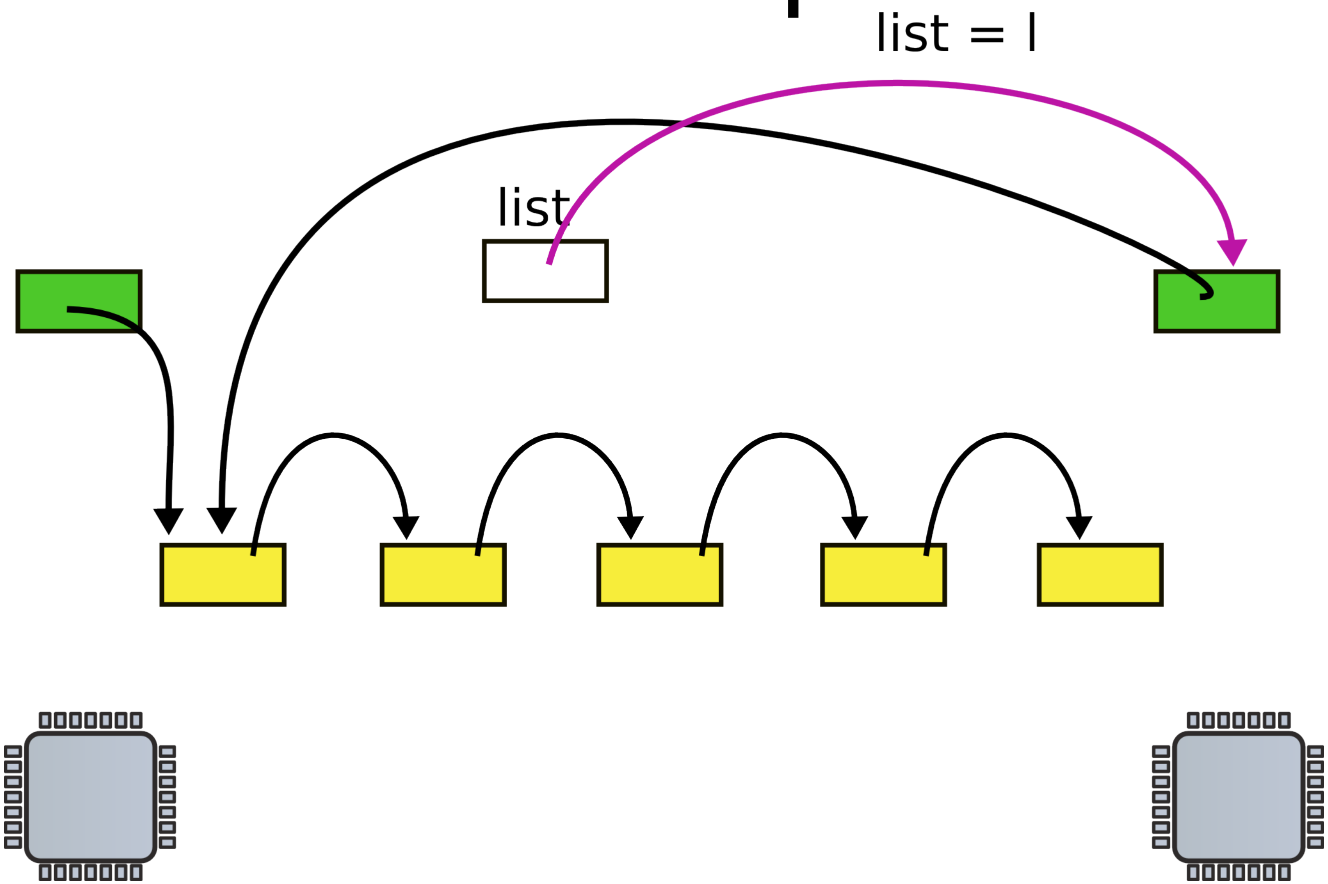
CPU 1 and 2 update next pointer



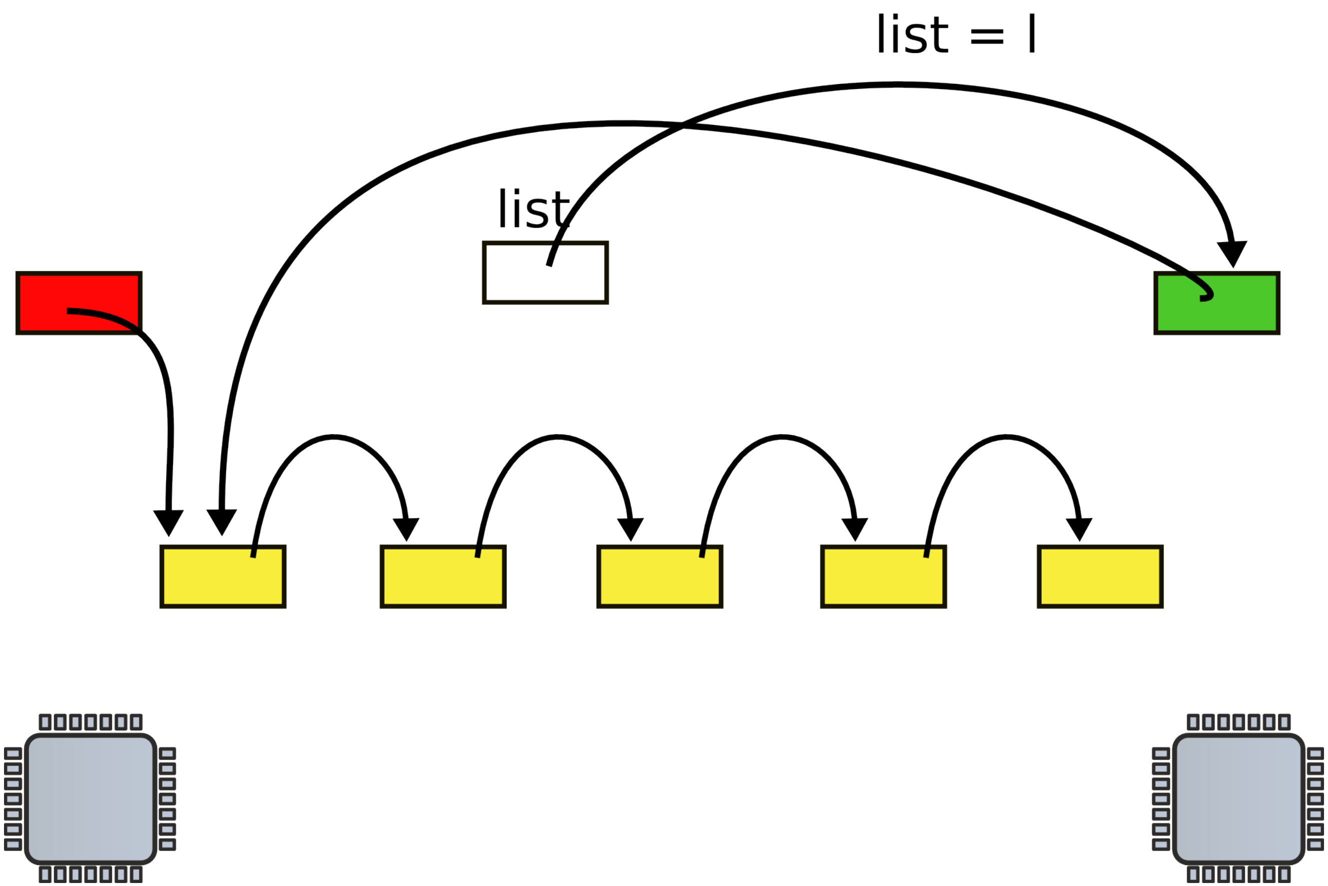
CPUs 1 updates head pointer



CPU2 updates head pointer



State after the race



List implementation with locks

```
insert(int data)
{
    struct list *l;
    l = malloc(sizeof *l);
    acquire(&listlock);
    l->data = data;
    l->next = list;
    list = l;
    release(&listlock);
}
```

- Critical section

Correct implementation

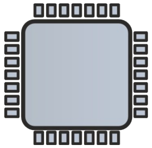
```
void
acquire(struct spinlock *lk)
{
...
  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;
...
}
```

Xchg instruction

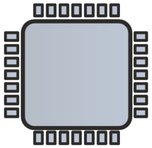
- Swap a word in memory with a new value
  - Atomic!
  - Return old value

Deadlocks

```
acquire(A)

acquire(B) {
    while(xchg(&B−>locked, 1) != 0)
}
```



```
acquire(B)

acquire(A) {
    while(xchg(&A−>locked, 1) != 0)
}
```



Lock ordering

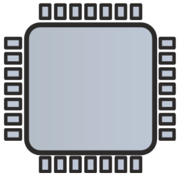
- Locks need to be acquired in the same order

Locks and interrupts

Network
interrupt

```
network_packet(){
    ....
    insert() {
        acquire(A)
        ...
    }

}
```

```
network_packet(){
    ....
    insert() {
        acquire(A)
        ...
    }

}
```



Locks and interrupts

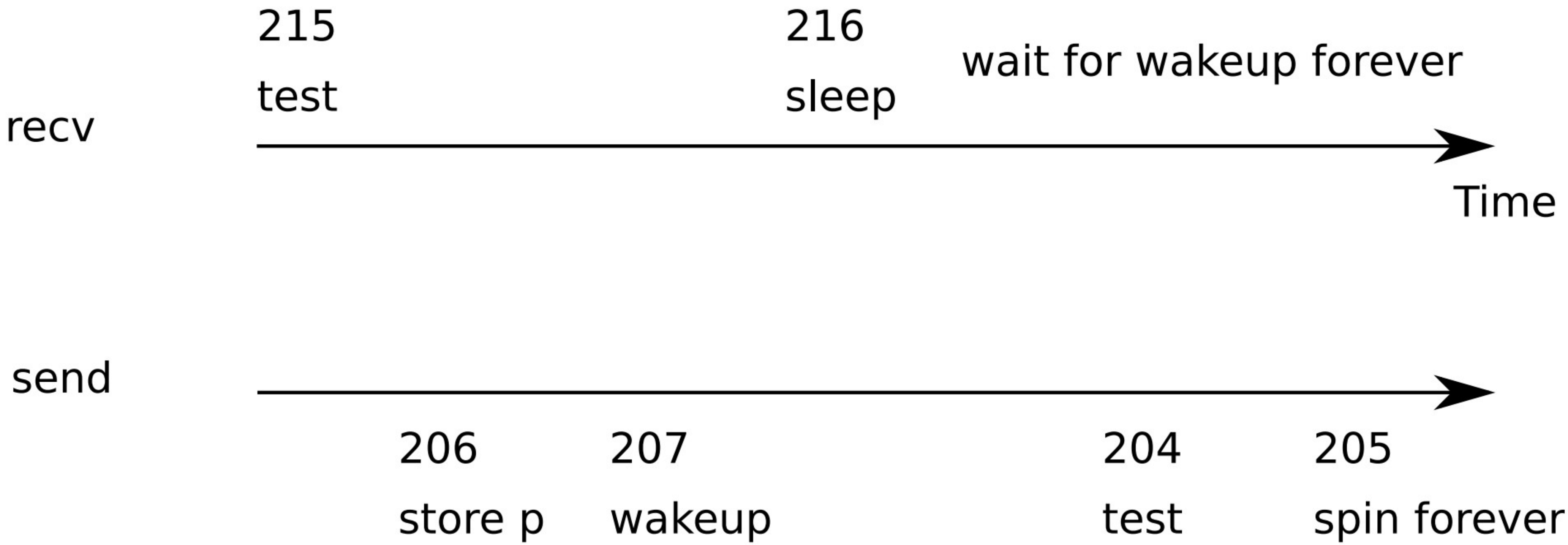
- Never hold a lock with interrupts enabled

Send/receive queue

```
void*
send(struct q *q, void *p)
{
  while(q->ptr != 0)
    ;
  q->ptr = p;
  wakeup(q); /*wake recv*/
}
```

```
void*
recv(struct q *q)
{
  void *p;

  while((p = q->ptr) == 0)
    sleep(q);
  q->ptr = 0;
  return p;
}
```

Lost wakeup problem



The role of file systems

- Sharing
  - Sharing of data across users and applications
- Persistence
  - Data is available after reboot

Architecture

- On-disk and in-memory data structures represent
  - The tree of named files and directories
  - Record identities of disk blocks which hold data for each file
  - Record which areas of the disk are free

Crash recovery

- File systems must support crash recovery
  - A power loss may interrupt a sequence of updates
  - Leave file system in inconsistent state
    - E.g. a block both marked free and used

Multiple users

- Multiple users operate on a file system concurrently
  - File system must maintain invariants

Speed

- Access to a block device is several orders of magnitude slower
  - Memory: 200 cycles
  - Disk: 20 000 000 cycles
- A file system must maintain a cache of disk blocks in memory

Block layer

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Read and write data
  - From a block device
  - Into a buffer cache
- Synchronize across multiple readers and writers

Transactions

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Group multiple writes into an atomic transaction

Files

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Unnamed files
  - Represented as inodes
  - Sequence of blocks holding file's data

Directories

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

- Special kind of inode
  - Sequence of directory entries
  - Each contains name and a pointer to an unnamed inode

Pathnames

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

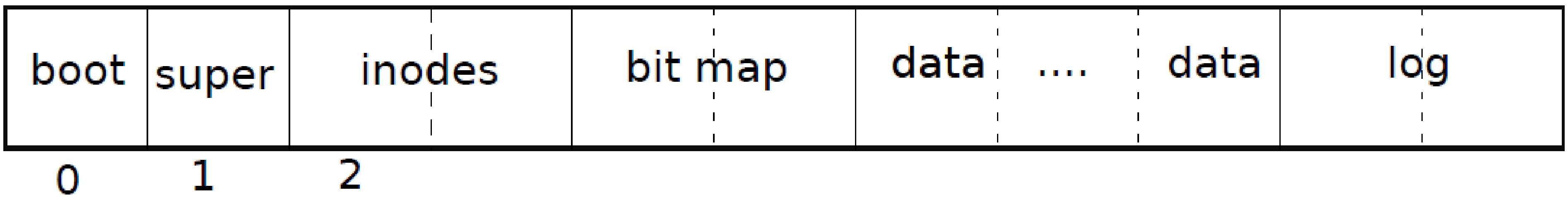
- Hierarchical path names
  - /usr/bin/sh
  - Recursive lookup

System call

System calls	File descriptors
Pathnames	Recursive lookup
Directories	Directory inodes
Files	Inodes and block allocator
Transactions	Logging
Blocks	Buffer cache

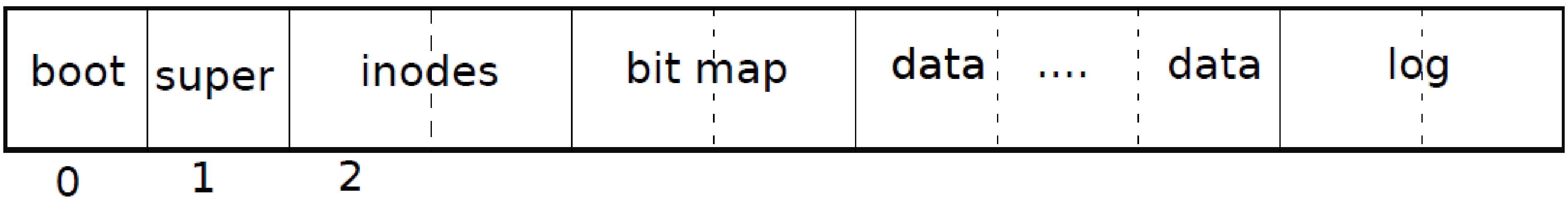
- Abstract UNIX resources as files
  - Files, sockets, devices, pipes, etc.
- Unified programming interface

File system layout on disk



- Block #0: Boot code
- Block #1: Metadata about the file system
  - Size (number of blocks)
  - Number of data blocks
  - Number of inodes
  - Number of blocks in log

File system layout on disk



- Block #2 (inode area)
- Bit map area: track which blocks are in use
- Data area: actual file data
- Log area: maintaining consistency in case of a power outage or system crash

Typical use of transactions

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

Strawman scheduler (xv6)

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?

Xv6 scheduler

```
scheduler(void)
{
  for(;;){
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
  }
}
```

Strawman scheduler (xv6)

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?

Strawman scheduler

- Organize all processes as a simple list
- In schedule():
  - Pick first one on list to run next
  - Put suspended task at the end of the list
- Problem?
  - Only allows round-robin scheduling
  - Can’t prioritize tasks

Priority based scheduling

- Higher-priority processes run first
- Processes within the same priority are round-robin

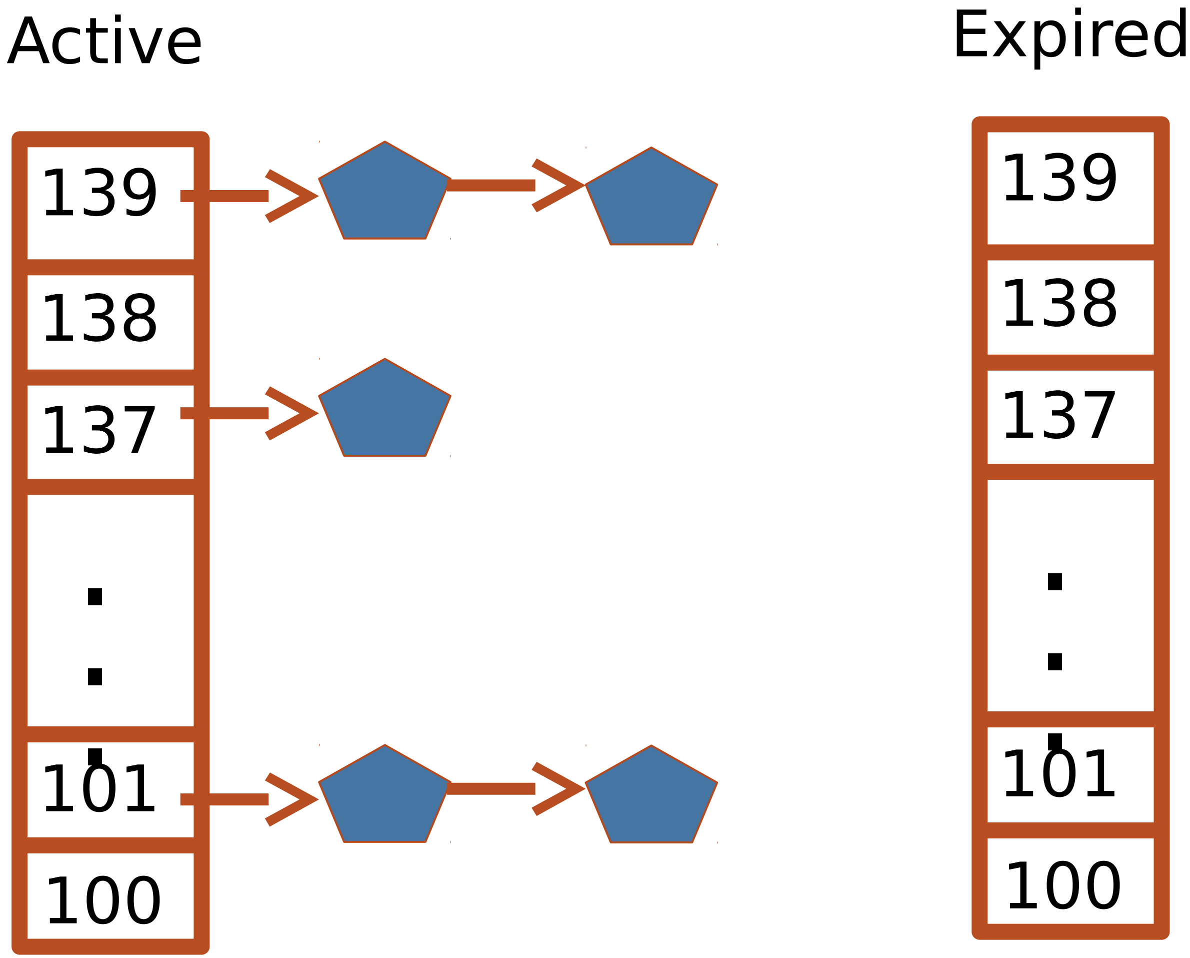
O(1) scheduler (Linux 2.6 – 2.6.22)

- Priority based scheduling
- Goal: decide who to run next, independent of number of processes in system
  - Still maintain ability to prioritize tasks, handle partially unused quanta, etc

O(1) data structures

- runqueue: a list of runnable processes
  - Blocked processes are not on any runqueue
  - A runqueue belongs to a specific CPU
  - Each task is on exactly one runqueue
  - Task only scheduled on runqueue's CPU unless migrated
- 2 *40 * #CPUs runqueues
  - 40 dynamic priority levels (more later)
  - 2 sets of runqueues – one active and one expired

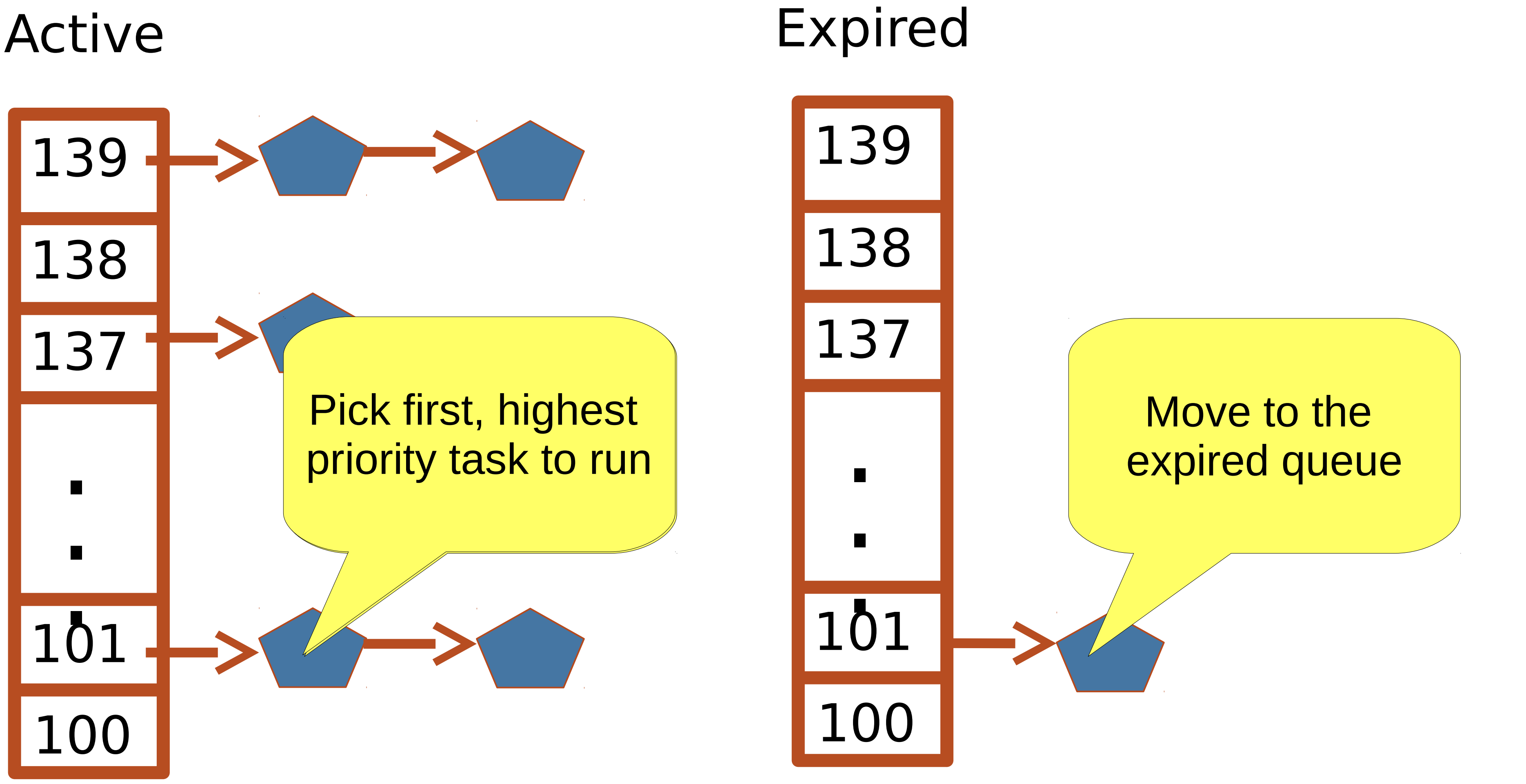
O(1) data structures (contd)



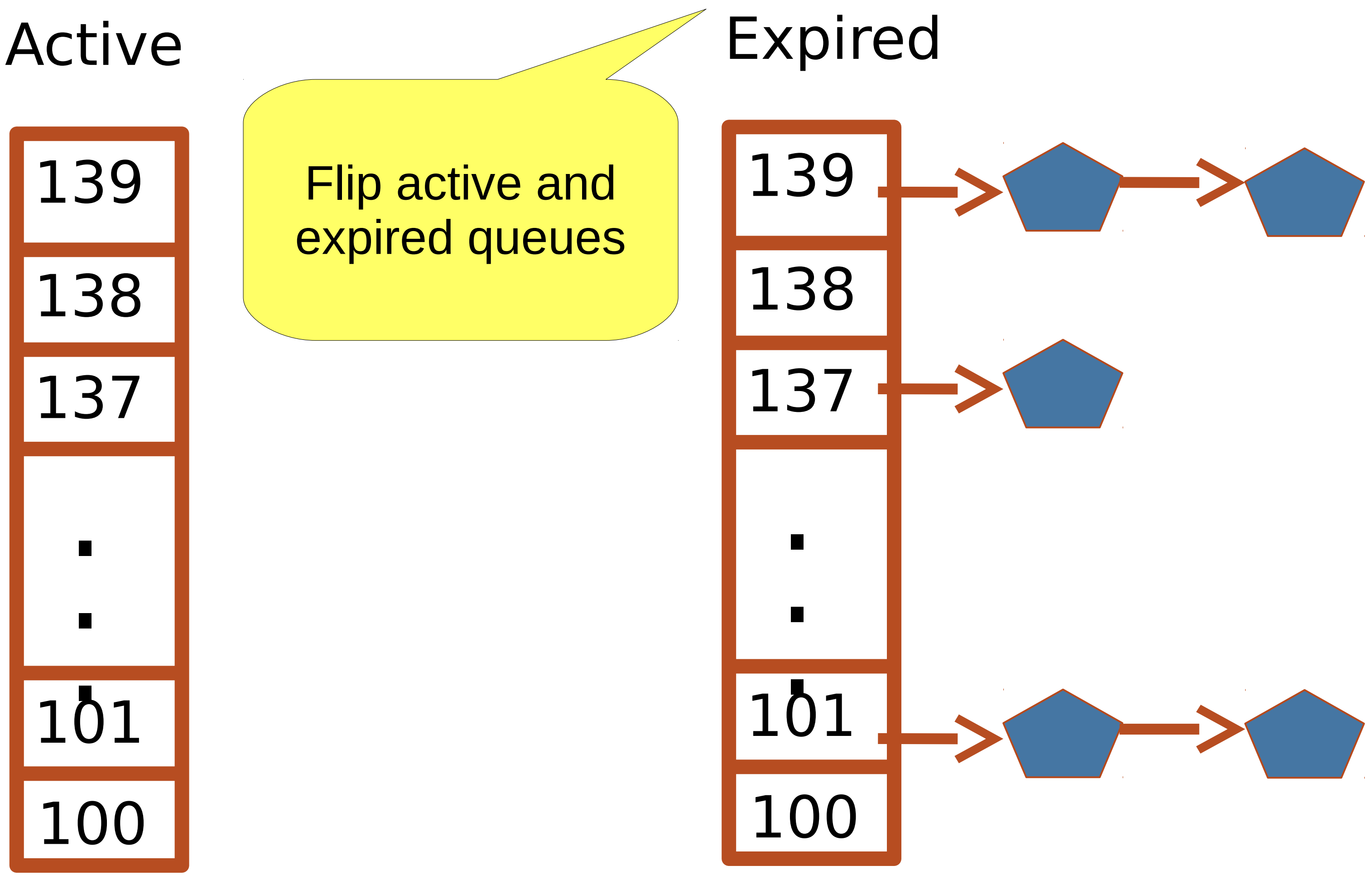
O(1) intuition

- Take the first task off the lowest-numbered runqueue on active set
  - Confusingly: a lower priority value means higher priority
- When done, put it on appropriate runqueue on expired set
- Once active is completely empty, swap which set of runqueues is active and expired
- Constant time, since fixed number of queues to check; only take first item from non-empty queue

O(1) example


Active
Expired
139
138
137
101
100
139
138
137
101
100
Pick first, highest priority task to run
Move to the expired queue

What now?

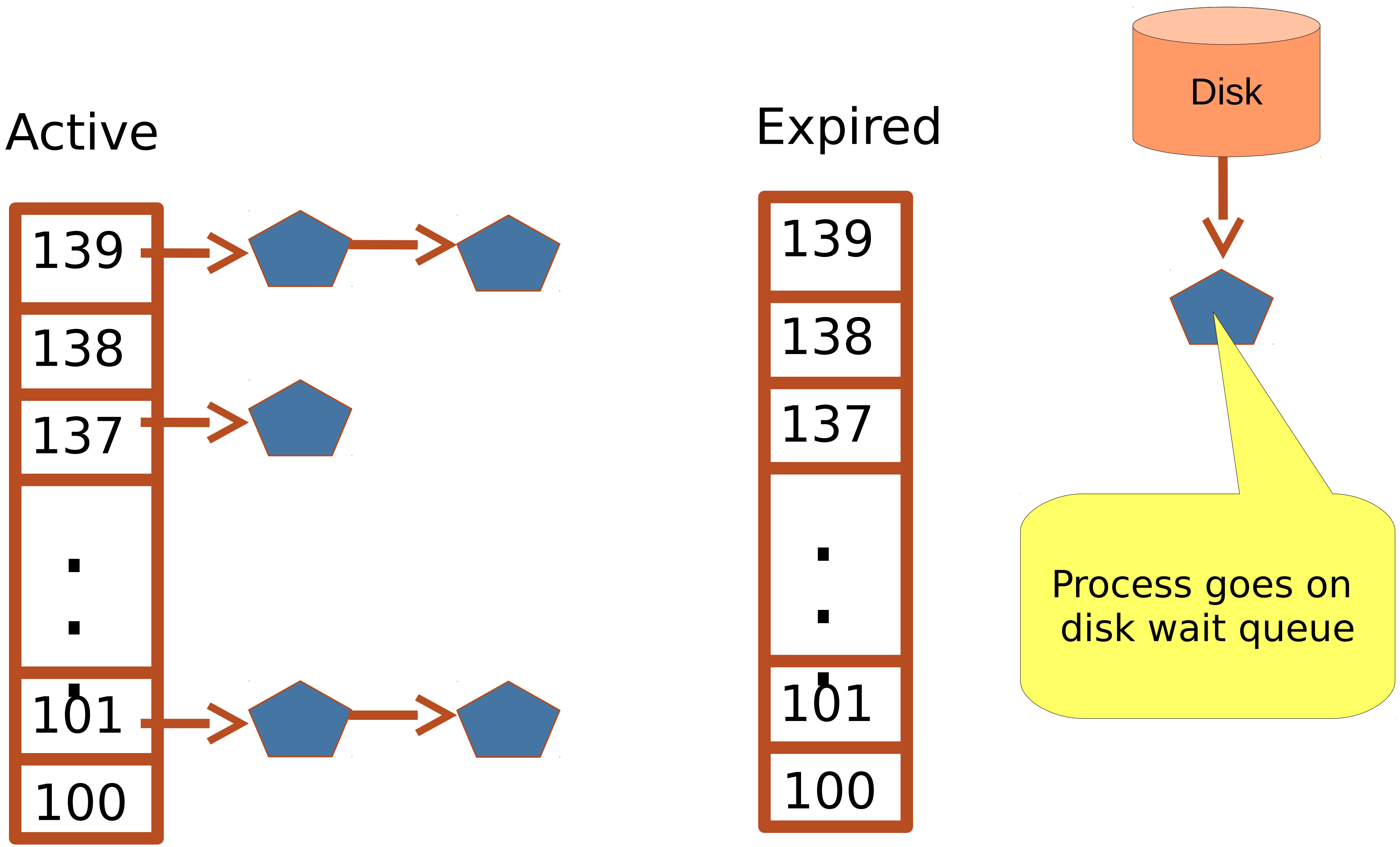


Active
Flip active and expired queues
Expired
139
138
137
101
100
139
138
137
101
100

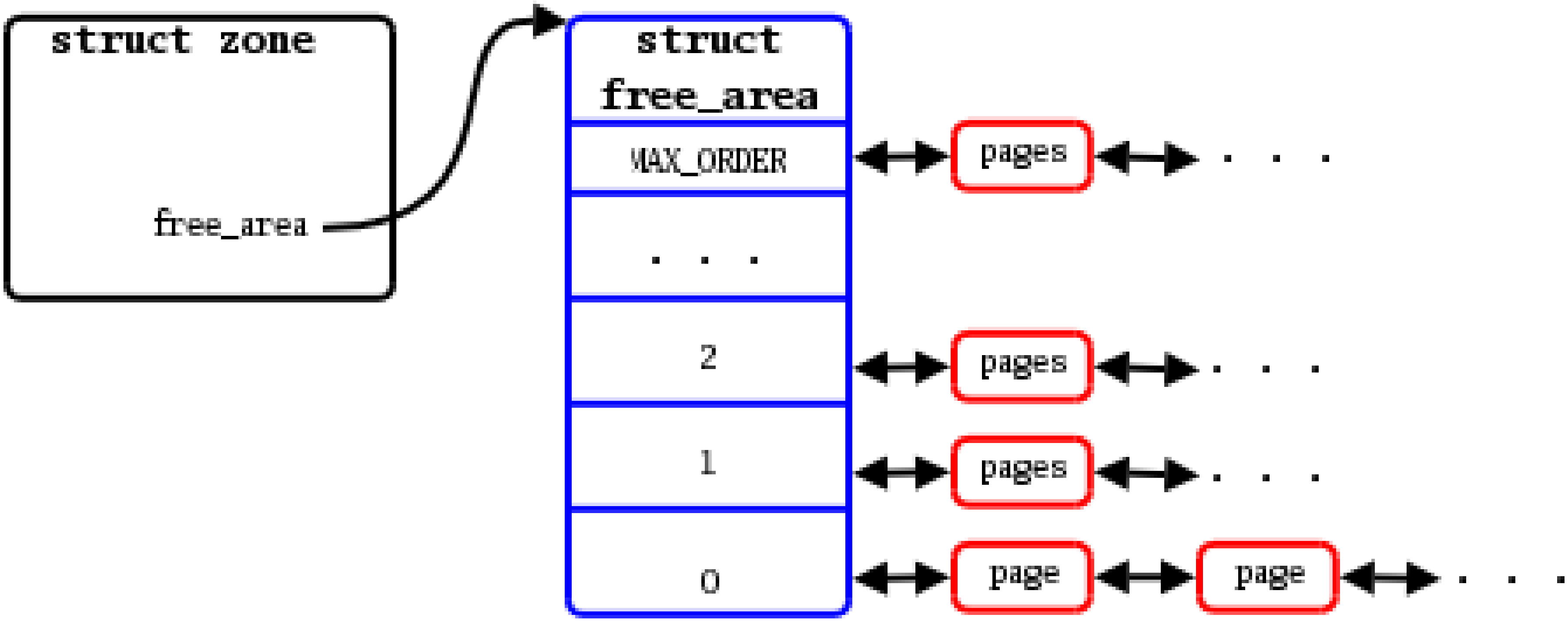
Blocked tasks

- What if a program blocks on I/O, say for the disk?
  - It still has part of its quantum left
  - Not runnable, so don’t waste time putting it on the active or expired runqueues
- We need a “wait queue” associated with each blockable event
  - Disk, lock, pipe, network socket, etc.

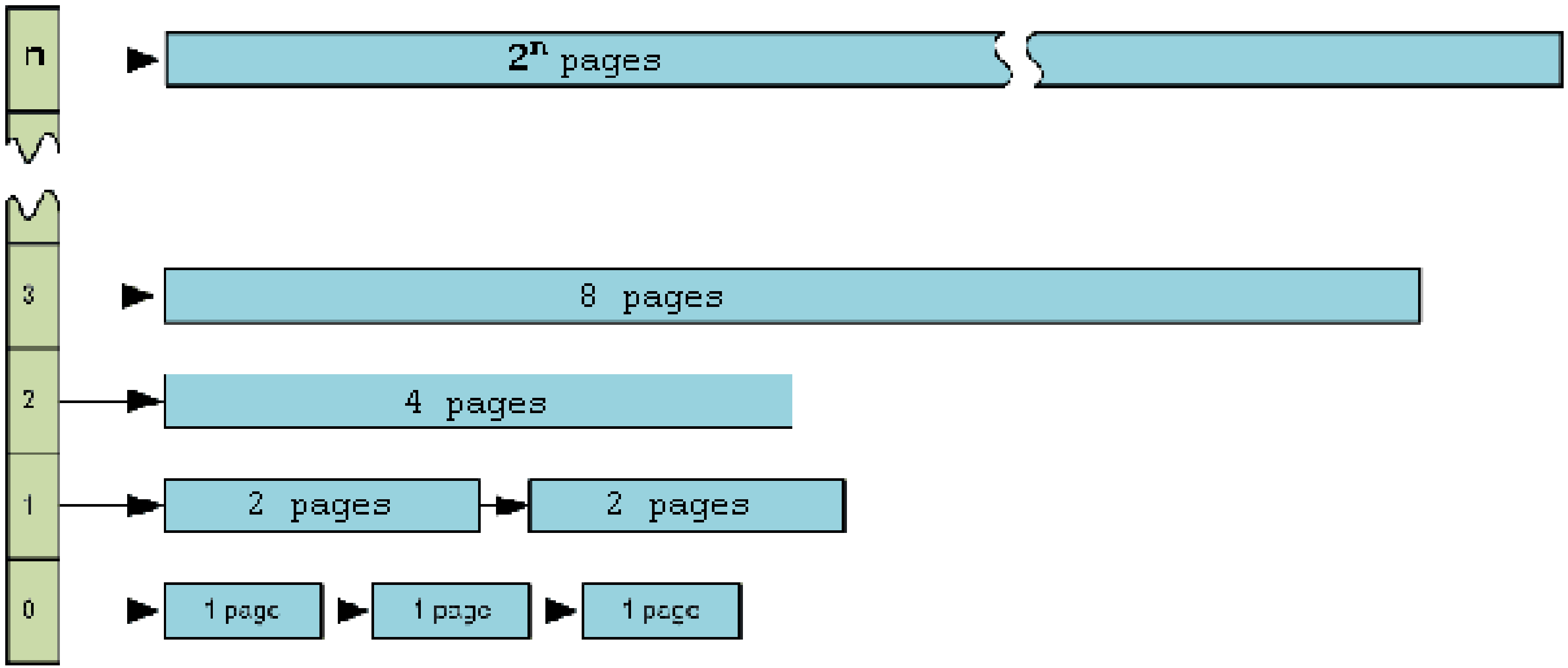
Blocking example



Buddy memory allocator



Buddy allocator



What's wrong with buddy?

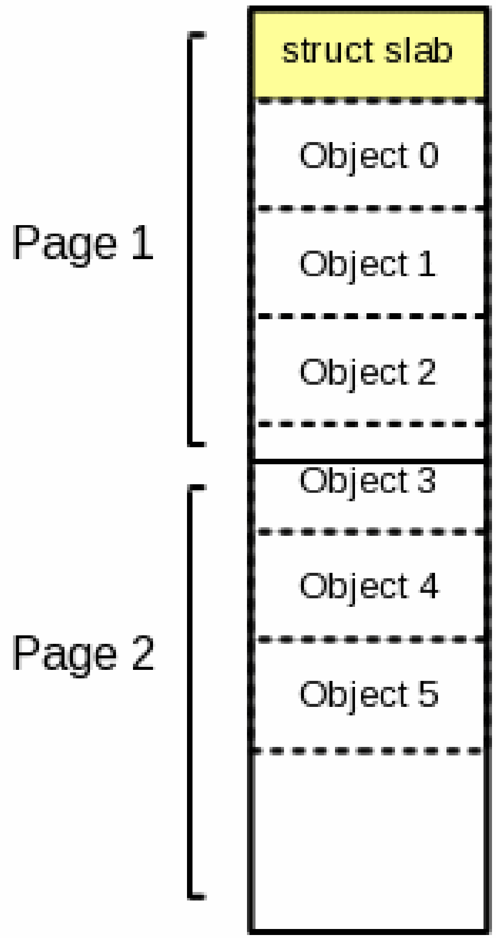
What's wrong with buddy?

- Buddy allocator is ok for large allocations
  - E.g. 1 page or more
- But what about small allocations?
  - Buddy uses the whole page for a 4 bytes allocation
    - Wasteful
  - Buddy is still slow for short-lived objects

Slab:
Allocator for object of a fixed size

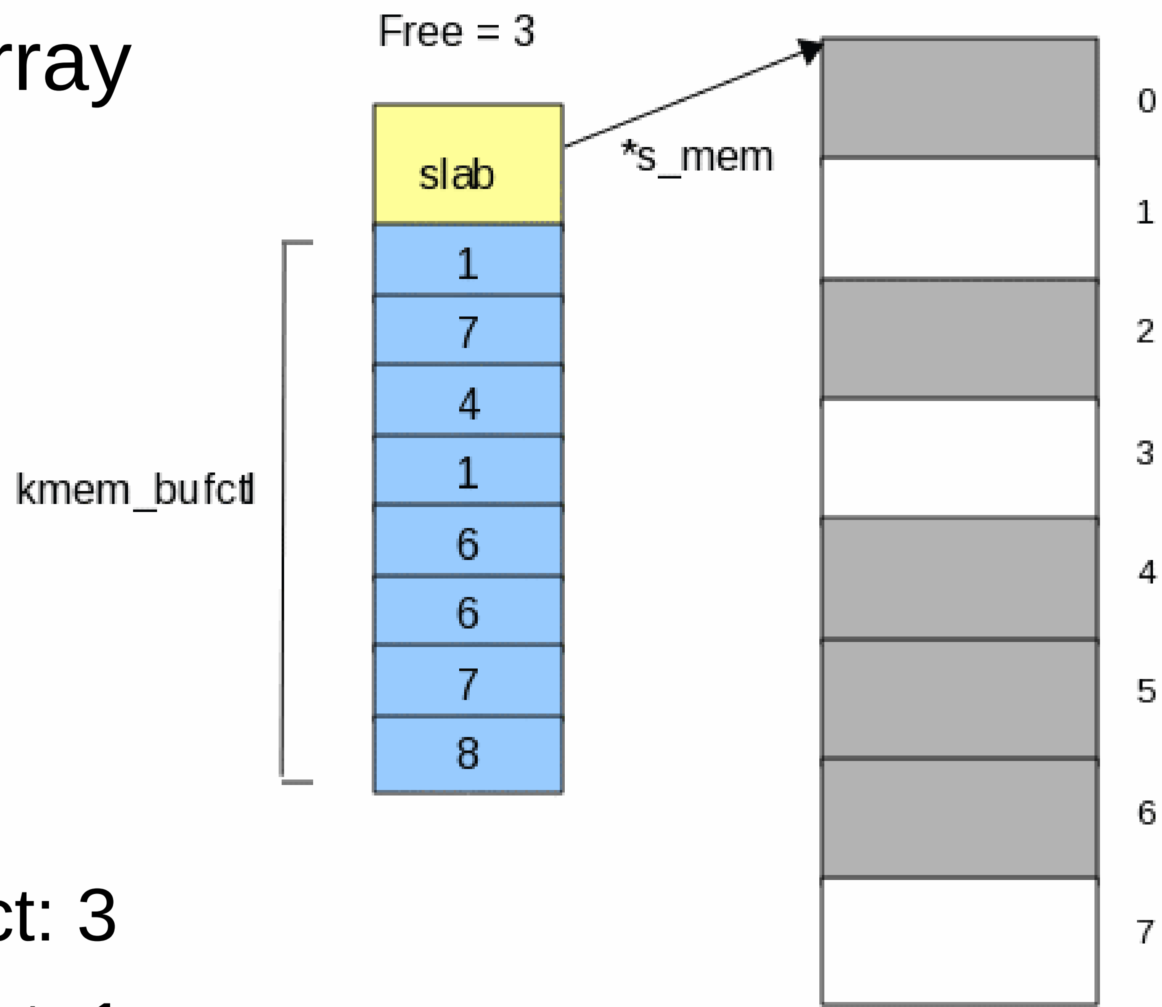
Slab

- A 2 page slab with 6 objects



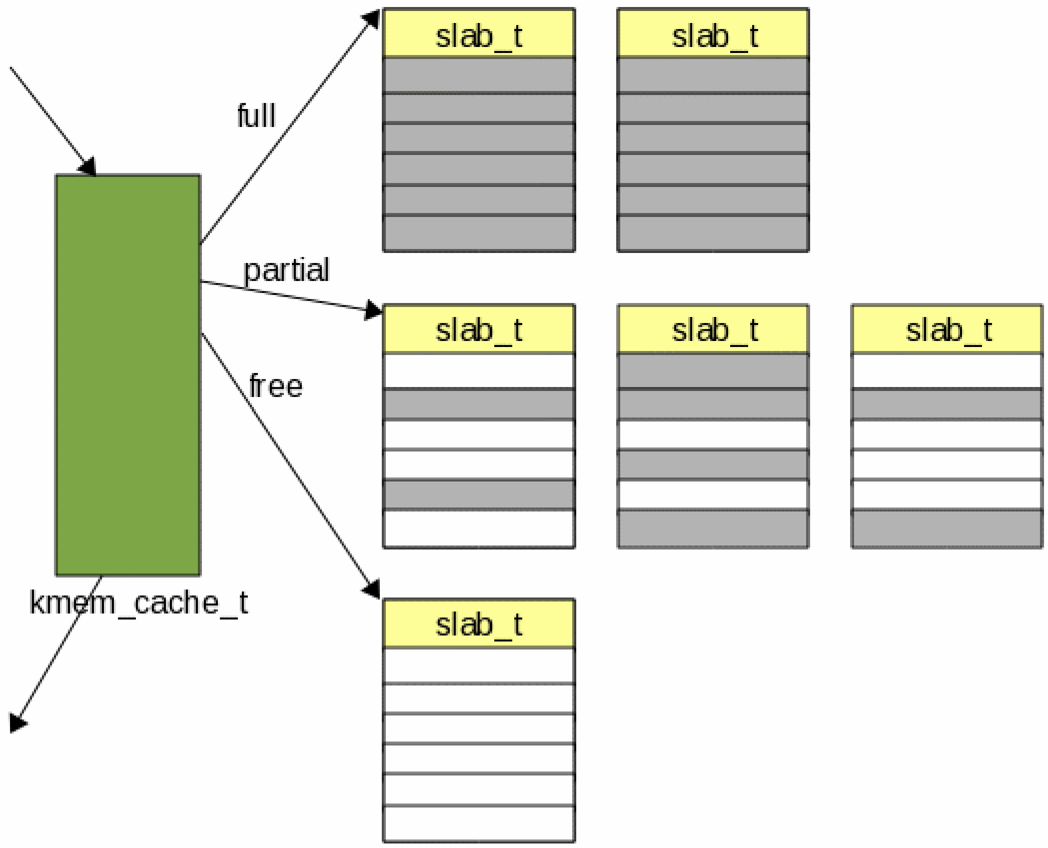
Keeping track of free objects

- kmem_bufctl array is effectively a linked list



  - First free object: 3
  - Next free object: 1

A cache is formed out of slabs



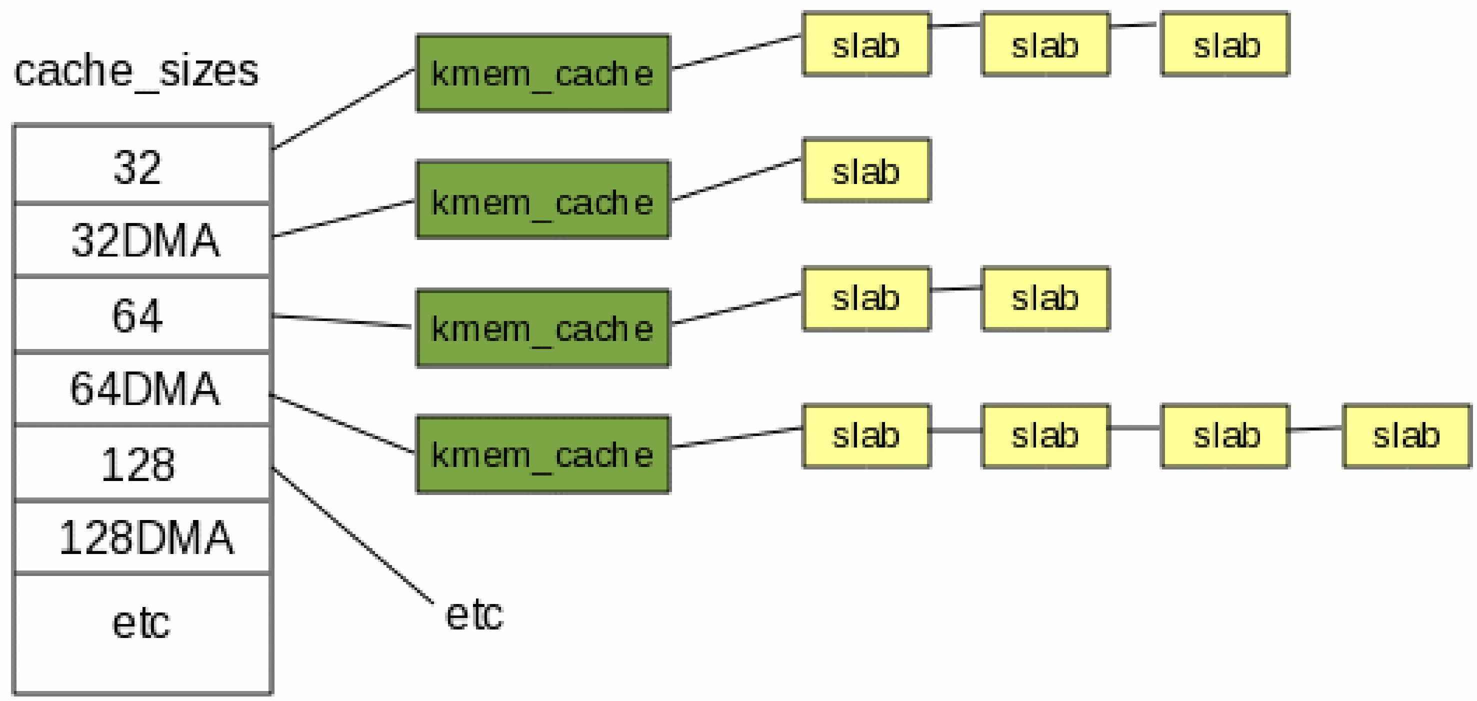
Slab is fine, but what's wrong?

Slab is fine, but what's wrong?

- We can only allocate objects of one size

Kmalloc(): variable size objects

- A table of caches
  - Size: 32, 64, 128, etc.



Thank you!