Memory Management

Memory Management – Address Space

Instructions and data to memory addresses can be done in following ways:

Compile time -- When it is known at compile time where the process will reside, compile time binding is used to generate the absolute code.

Load time -- When it is not known at compile time where the process will reside in memory, then the compiler generates re-locatable code.

Execution time -- If the process can be moved during its execution from one memory segment to another, then binding must be delayed to be done at run time

Ease of Programming

```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov EAX, [0]
        mov EBX, [4]
        cmp EAX,EBX
        jle 0x7 //Label_1
        mov [8], EAX
        jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….

Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```

Code and Data segments are separate and both assumed to start from 0

Every Memory Data Access should add the value stored in Data Segment Register By default.

Segment Register (Data): 2100

Address of j: 2100
Address of k: 2104
Address of max: 2108

Main Memory

Address	Contents
0000	Operating System (Kernel)
0700	Other User Process
0900	Our Code Segment
1900	Vacant Space
2100	Our Data Segment
2300	Vacant Space
2500	

Ease of Process Mobility

```
if (j>k)
  max = j
else
  max = k
```

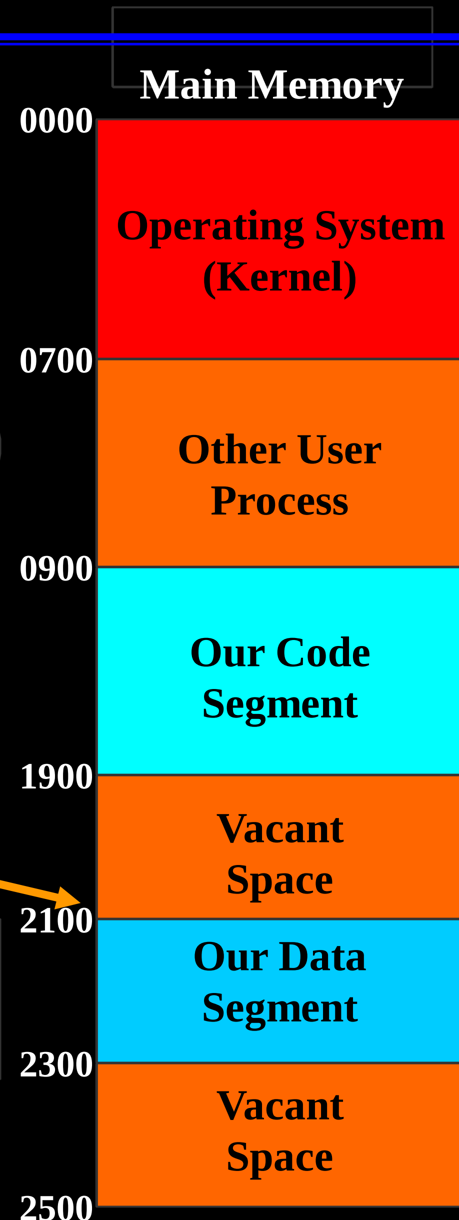
```
Code_Segment:
          mov EAX, [0]
          mov EBX, [4]
          cmp EAX,EBX
          jle 0x7 //Label_1
          mov [8], EAX
          jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….
Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```

A new process needs a segment of size 260
The space is available but not contiguous

Segment Register (Data)

2100

Address of j: 2100
Address of k: 2104
Address of max: 2108

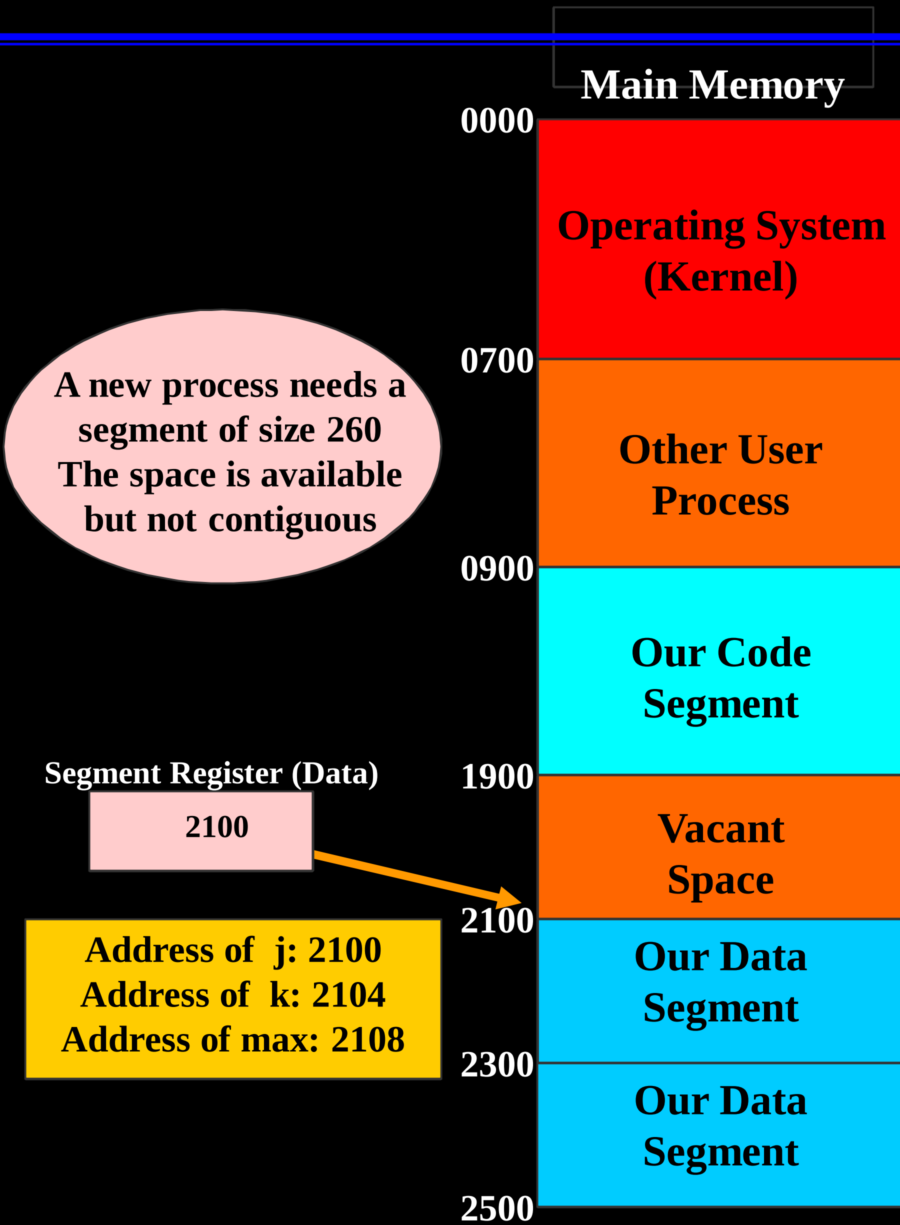


Ease of Process Mobility

```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov EAX, [0]
        mov EBX, [4]
        cmp EAX,EBX
        jle 0x7 //Label_1
        mov [8], EAX
        jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….
```

```
Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```



Ease of Process Mobility

```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov EAX, [0]
        mov EBX, [4]
        cmp EAX,EBX
        jle 0x7 //Label_1
        mov [8], EAX
        jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….
```

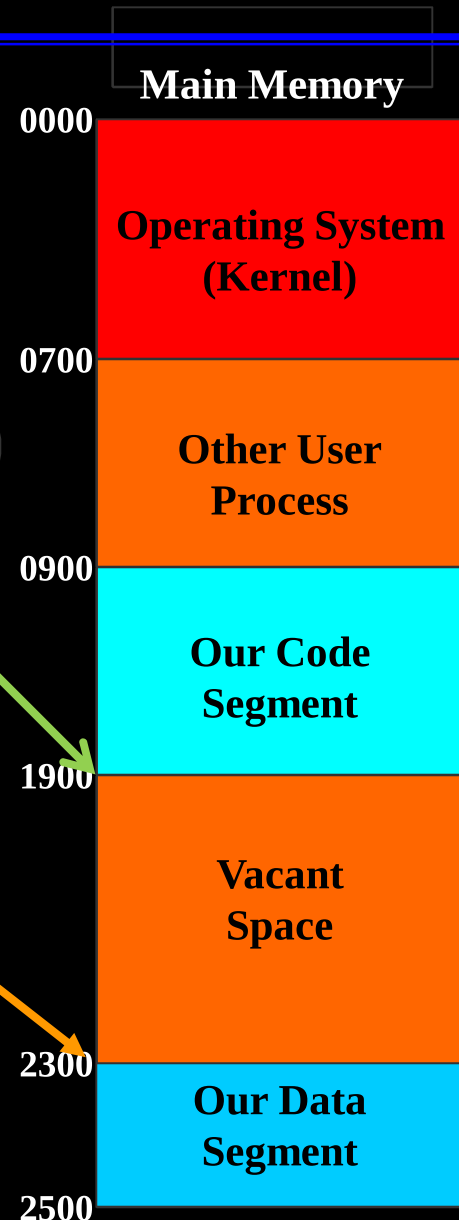
```
Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```

A new process needs a segment of size 260
The space is available but not contiguous

Segment Register (Data)

2300

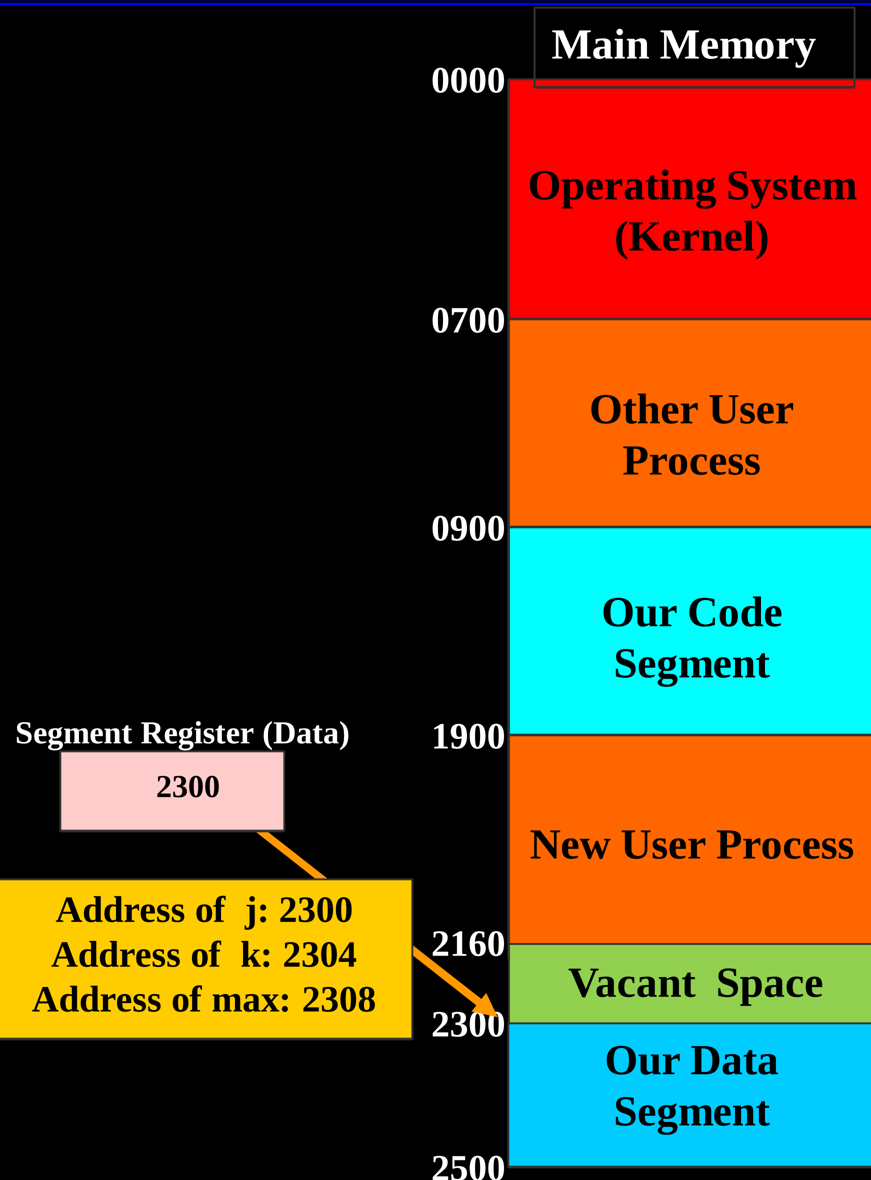
Address of j: 2300
Address of k: 2304
Address of max: 2308



Ease of Process Mobility

```
if (j>k)
  max = j
else
  max = k
```

```
Code_Segment:
        mov EAX, [0]
        mov EBX, [4]
        cmp EAX,EBX
        jle 0x7 //Label_1
        mov [8], EAX
        jmp 0x5 //Label_2
Label_1: mov [8], EBX
Label_2: ….
Data Segment:
0: // Allocated for j
4: // Allocated for k
8: // Allocated for max
```

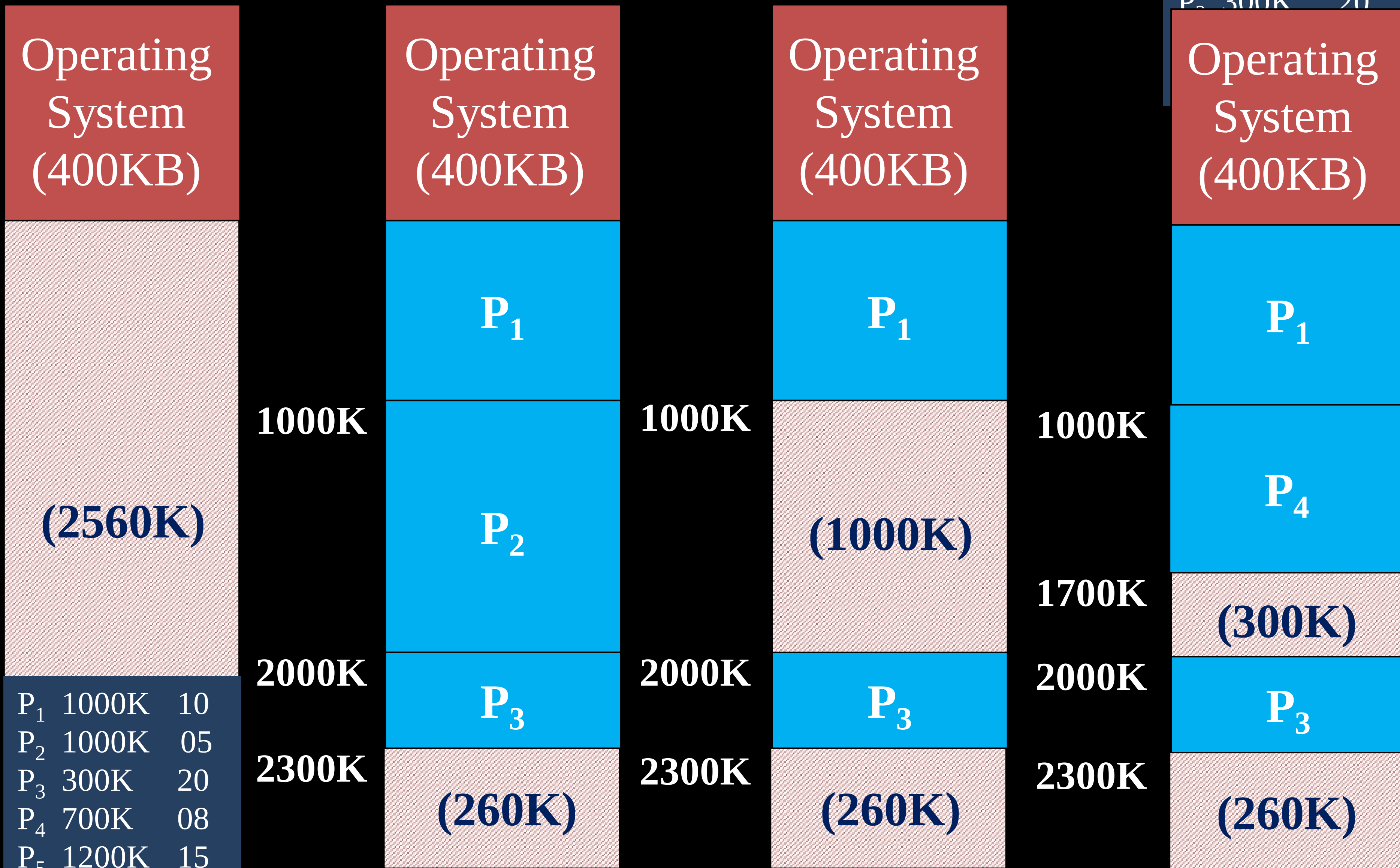


Memory Management – Dynamic Partition

Process	Memory	Time
P_1	1000K	10
P_2	1000K	05
P_3	300K	20
P_4	700K	08
P_5	1200K	15

Memory Management – Dynamic Partition

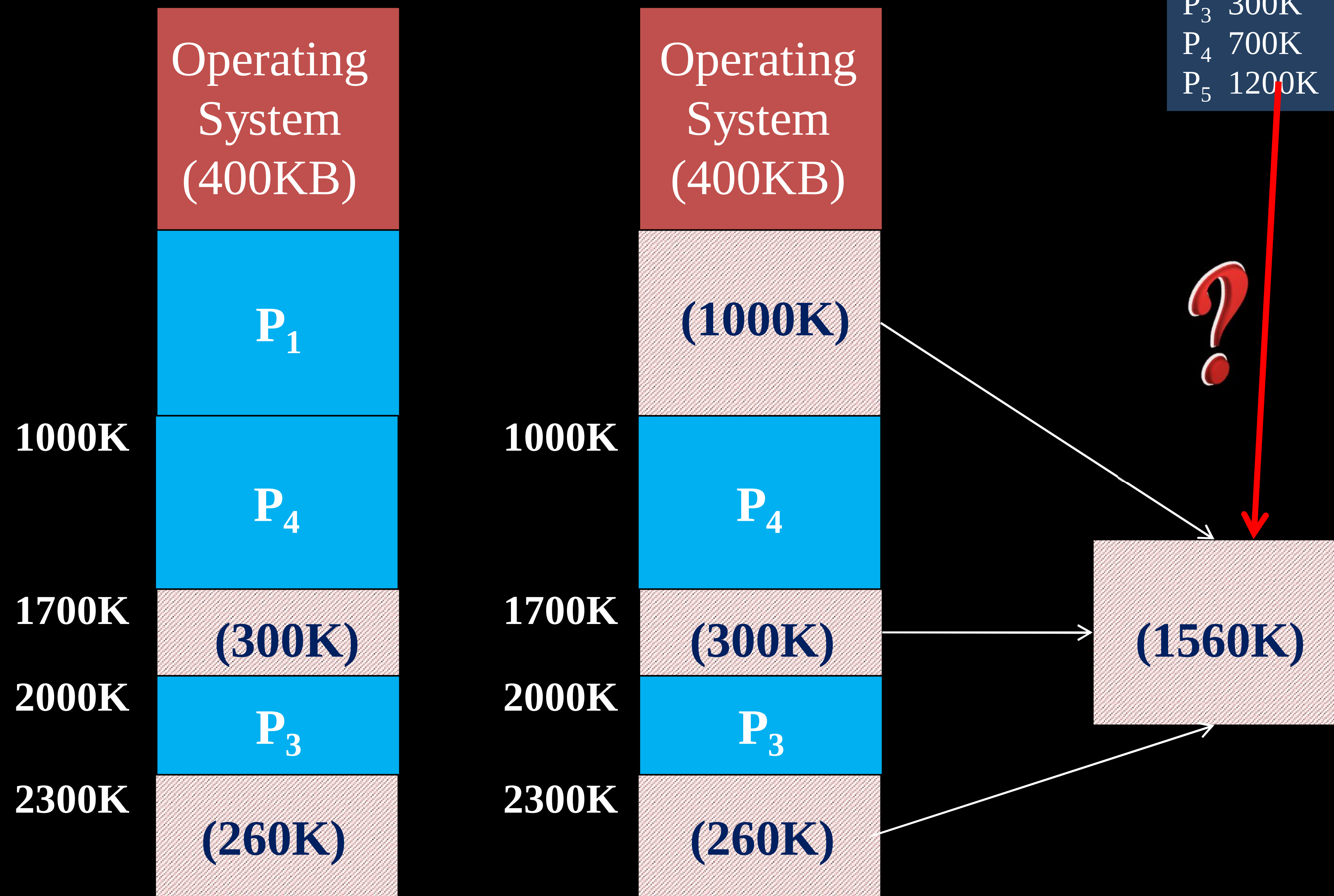
P_1	1000K	10
P_2	1000K	05
P_3	300K	20
P_4	700K	08
P_5	1200K	15



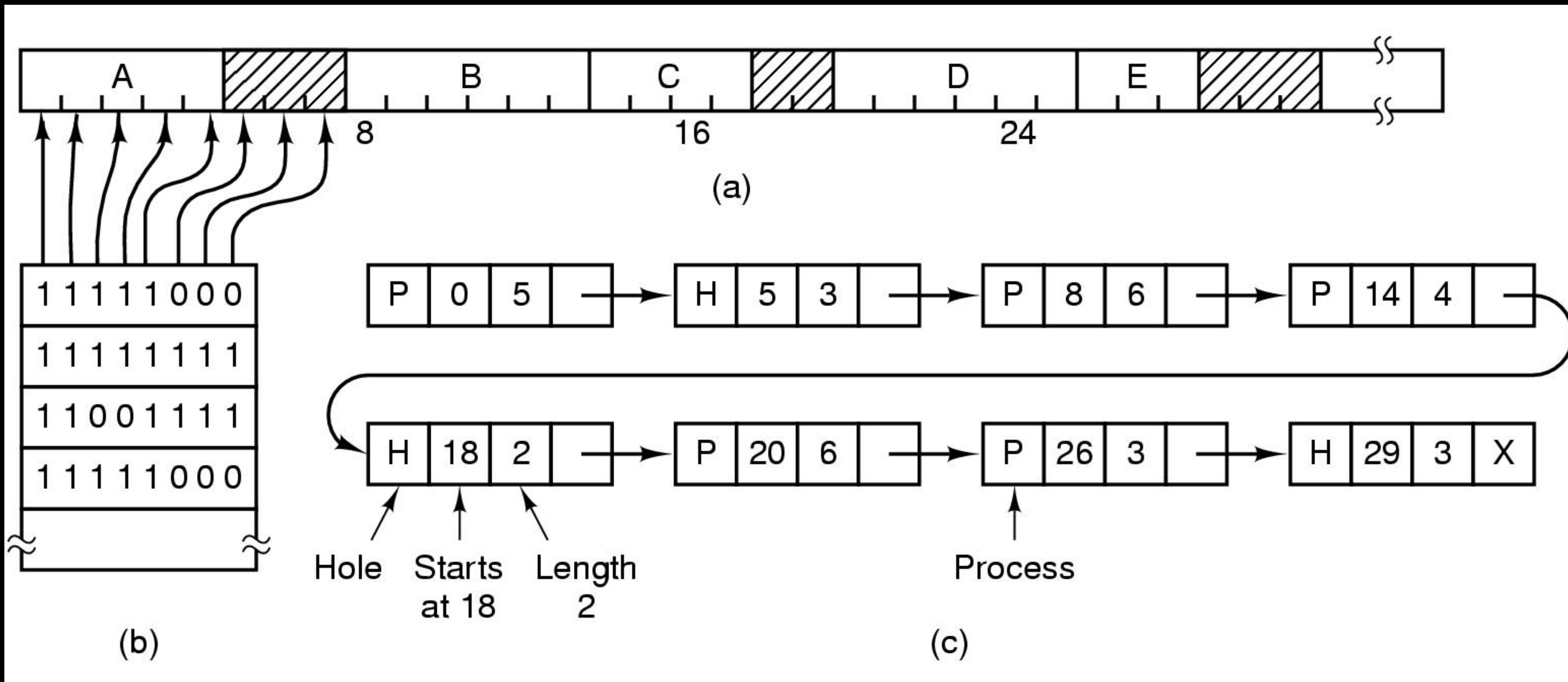
P_1	1000K	10
P_2	1000K	05
P_3	300K	20

Memory Management – Dynamic Partition

P_1	1000K	10
P_2	1000K	05
P_3	300K	20
P_4	700K	08
P_5	1200K	15

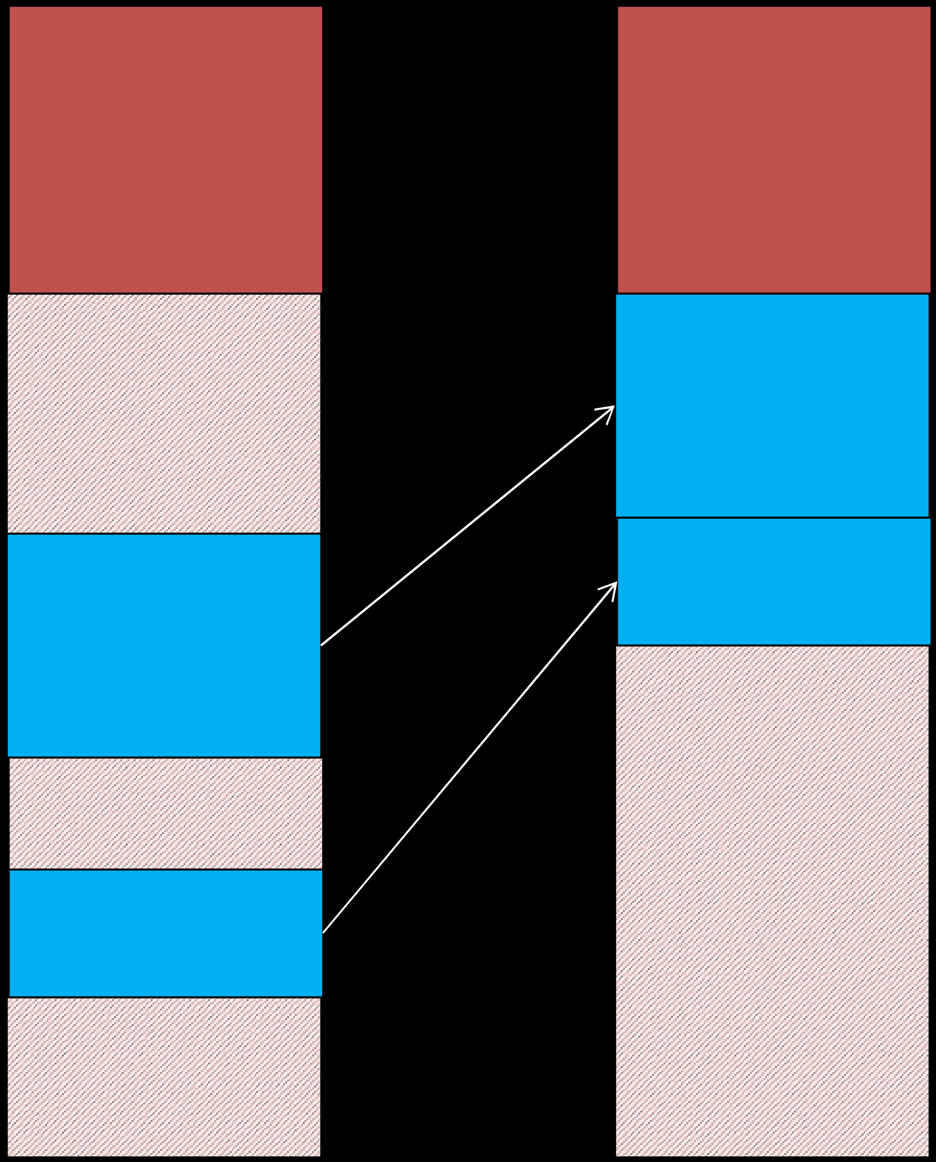


Memory Management - with bit map



Part of memory with 5 processes, 3 holes

Memory Management – Compaction



Shuffle memory contents to place all free memory together in one large block. Compaction is possible only if relocation is dynamic, and is done at execution time

I/O problem

Latch job in memory while it is involved in I/O
Do I/O only into OS buffers

Memory Management – Allocation Algorithm

First-fit:

Scans memory form the beginning and chooses the first available block that is large enough

Fastest

May have many process loaded in the front end of memory that must be searched over when trying to find a free block

Best-fit:

Chooses the block that is closest in size to the request

Worst performer overall

Since smallest block is found for process, the smallest amount of fragmentation is left

Memory compaction must be done more often

Memory Management – Allocation Algorithm

Worst-fit:

Allocate the *largest* hole

Must search entire list to produce the largest leftover hole

Next-fit:

Scans memory from the location of the last placement

More often allocate a block of memory at the end of memory where the largest block is found

The largest block of memory is broken up into smaller blocks

Compaction is required to obtain a large block at the end of memory

First-fit and best-fit better than worst-fit in terms of speed and storage utilization

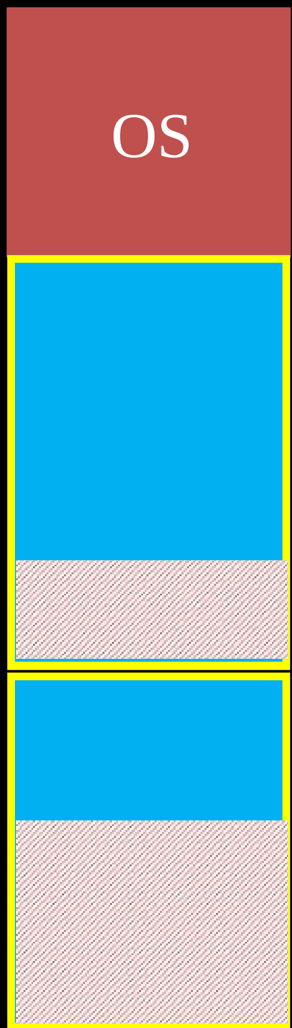
Memory Management – Allocation Algorithm

Memory Management – Address Space

Compare Best-Fit with Worst-Fit. What is the basic advantage of Worst-Fit ?

A criticism of the best fit algorithm is that the space remaining after allocating a block of the required size is so small that in general it is of no real use. The worst fit algorithm maximizes the chance that the free space left after a placement will be large enough to satisfy another request, thus minimizing the frequency of compaction. The disadvantage of this approach is that the largest blocks are allocated first; therefore a request for a large area is more likely to fail.

Memory Management – Fragmentation

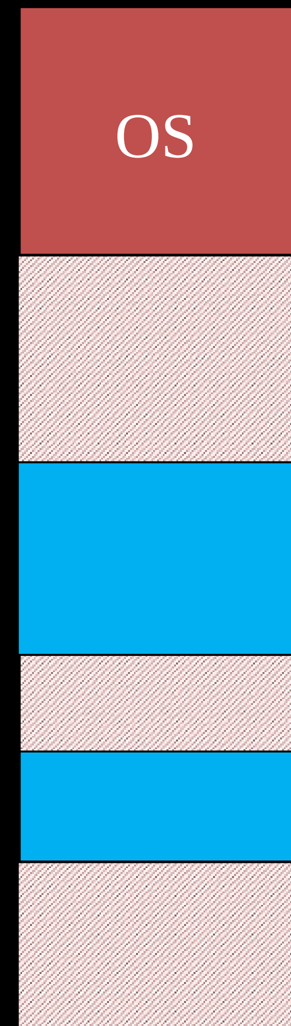


Internal
(Fixed Partition)

Fragmentation problem - There are two kinds of fragmentations:

External: It exists when total memory space available satisfies the requirement, but it is not contiguous. Total storage is fragmented into a large number of smaller blocks.

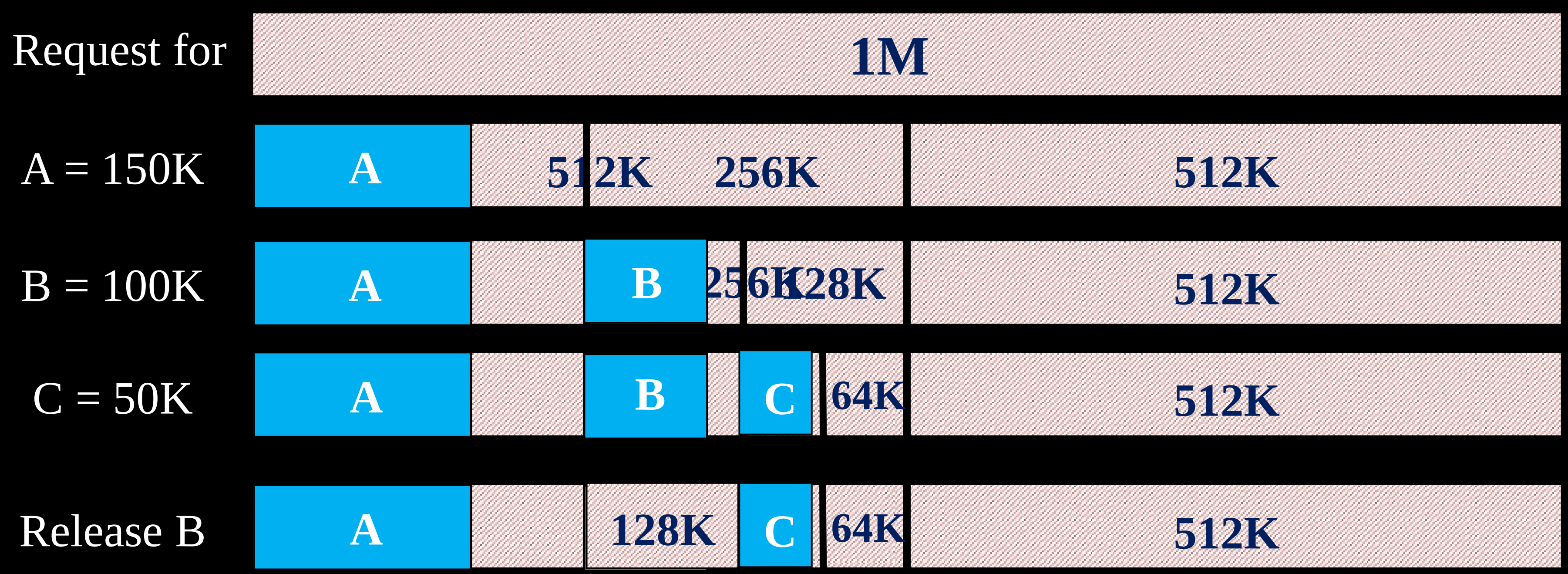
Internal: Internal fragmentation exists when the smallest available block is larger than the requested memory. It is the memory which is internal to a partition, but is not being used.



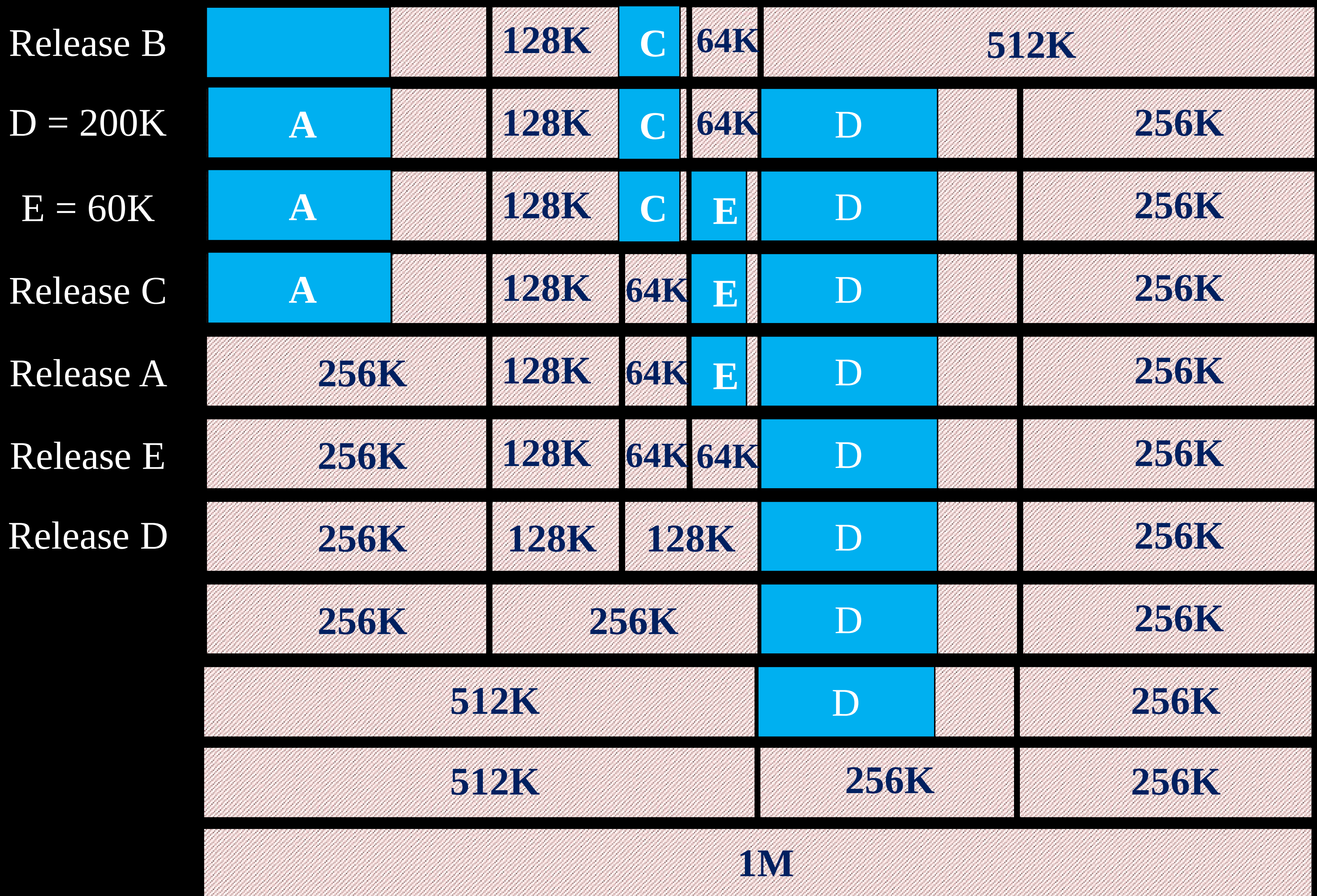
External
(Variable Partition)

Memory Management – Buddy System

In buddy system the entire space available is treated as a single block of 2^U. If a request of size S such that $2^{U-1} < S <= 2^U$, entire block is allocated. Otherwise block is split into two equal buddies. Process continues until smallest block greater than or equal to s is generated.



Memory Management – Buddy System



Memory Management – Buddy System

On a system with 1MB memory using the Buddy System, show the following allocation / deallocation sequence:
A=100K, B=240K, C=60K, D=250K, Release B, Release A, E=75, Release C, Release E, Release D

Draw the Tree representation of the above example.