N processes

var

number:array[0...n-1] of integer

(a,b)<(c,d) if a<c or if a=c and b<d

max(a0,….an-1)is a number,k such that k>=ai for i=0...n-1

number[i]=max(number[0],…….number[n-1])+1

for(j=0 to n-1)

Do

While number[j]!=0 and (number[j],j)<(number[i],i) do no_operation

end

CS

number[i]=0;

....

N processes

var

number:array[0...n-1] of integer

max(a0,….an-1)is a number,k such that k>=ai for i=0...n-1

number[i]=max(number[0],…….number[n-1])+1

for(j=0 to n-1)

Do

While number[j]!=0 and (number[j])<(number[i]) do no_operation

end

CS

number[i]=0;

....

numbers : array[0, ... (n-1)] of Integer

(a,b) < (c,d) if a<=c and b<d.

$K = max(a_i, a_{i+1}, \cdots a_{n-1})$ such that $k \geq$ all $a_i, a_{i+1}, \cdots a_{n-1}$

```
numbers[i] = max(numbers[0], numbers[1], ...
        ... numbers[n-1]) + 1;

for (j = 0 to n-1)
{
        while ( numbers[j] != 0  and
                (numbers[j], j) < numbers[i], i))
        {
                do no-op
        }
}
  c s
number[i] = 0;
```

N processes

var choosing: array[0,...n-1] of Boolean;

number:array[0...n-1] of integer

$(a,b)<(c,d)$ if $a<c$ or if $a=c$ and $b<d$

max(a0,….an-1)is a number,k such that k>=ai for i=0...n-1

```
choosing[i]=true
number[i]=max(number[0],…….number[n-1])+1
choosing[i]=false
for(j=0 to n-1)
Do
    While choosing[j] do no_operation
            While number[j]!=0 and (number[j],j)<(number[i],i) do no_operation
            end
            CS
            number[i]=0;
                ….
```

semaphores

.OS provides an tools to solve the CS problem

Semaphores.

.Semaphore is a data type like int/float/char...

.Two operations are allowed in semaphore

.Wait and signal(both atomic operation)

.S is semaphore variable

wait(s): while s<=0 do no_op signal(s):s=s+1

s=s-1

Use of semaphore

wait(s)

Critical section

signal(s)

P1's CS should execute after P2's CS

P1	p2
wait(s)	CS
CS	signal(s)

The disadvantage with the definition of semaphore given above is that it requires busy waiting.. while entering the CS p1 uses the CPU cycles to get permission and waiting for it...so CPU cycles are wasted which could be given to other processes.

solution

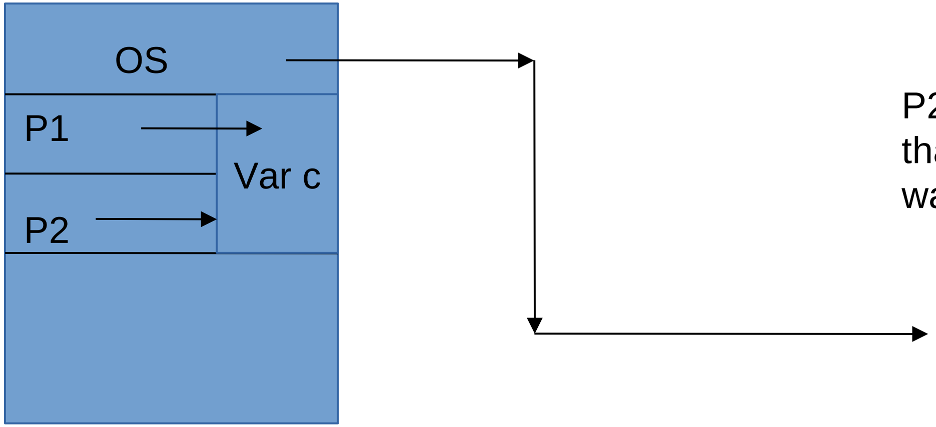
So the definition of the semaphore could be modified.

There is a waiting queue associated with each semaphore.

When a process executes the wait operation and find that the semaphore value is not positive, instead of busy waiting , it blocks itself.

The blocking operation includes

- Placing this process in the waiting queue of the semaphore.
- Switch its state to waiting state.

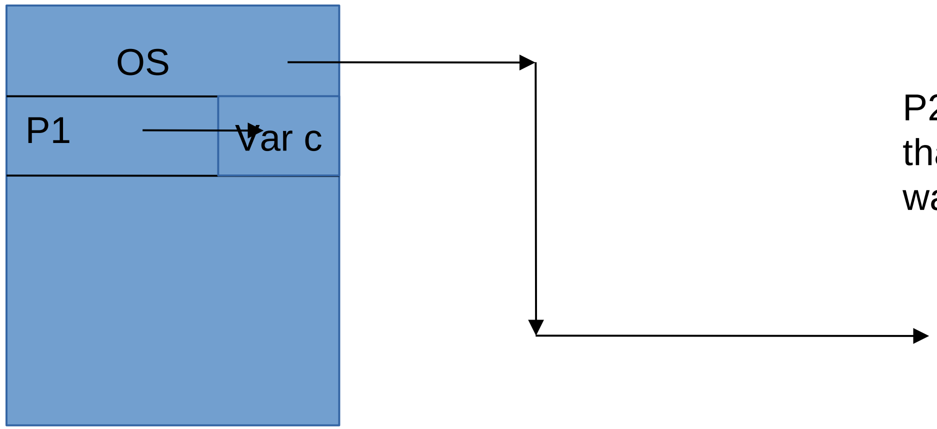


P2 is waiting for the Semaphore(s), so that it can enter the CS, but since it is waiting doing nothing

Queue of semaphore S
P2

The blocking operation includes

- Placing this process in the waiting queue of the semaphore.
- Switch its state to waiting state.



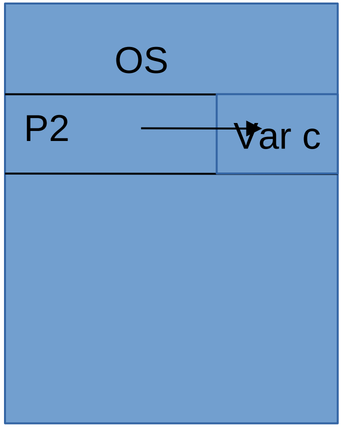
P2 is waiting for the Semaphore(s), so that it can enter the CS, but since it is waiting doing nothing

Queue of semaphore S
P2

P2 is in waiting state now

The blocking operation includes

- Placing this process in the waiting queue of the semaphore.
- Switch its state to waiting state.



P1 finish it s CS so signal operation is performed by P1, then OS wakes up P2 and P2 state becomes ready.

Dead lock and Starvation

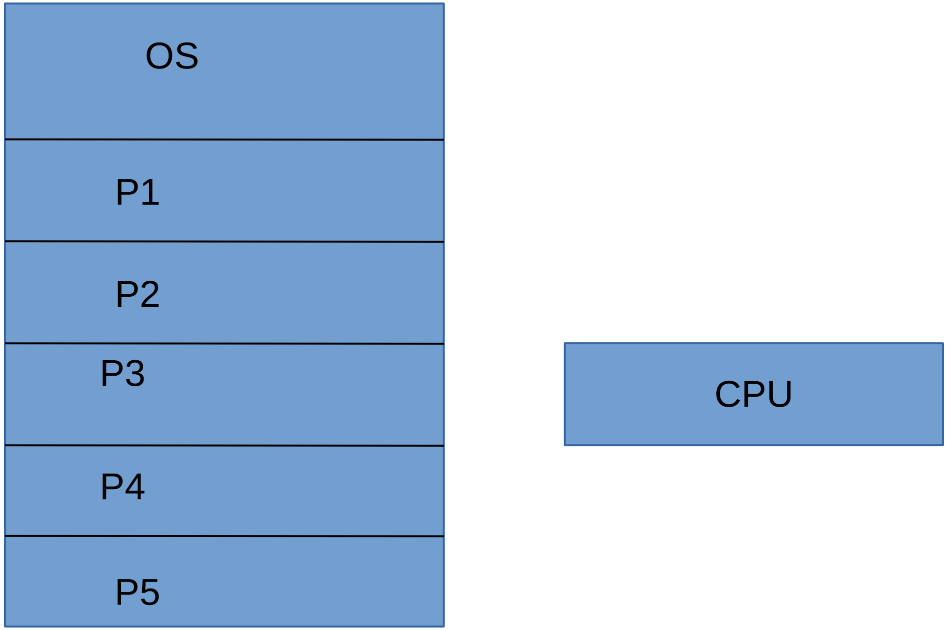
P0	P1
wait(s)	wait(q)
wait(q)	wait(s)
signal(s)	signal(q)
signal(q)	signal(s)

Both processes P0 and P1 are waiting for each other......
This kind of situation is known as DEAD LOCK.

Indefinite blocking or starvation is a situation in which processes wait within the semaphore.

It may occur if the order in which processes are removed from the list associated with a semaphore is lIFO.

CPU Scheduling(short term scheduler)



P1
P2
P3
P4
P5
Ready processes

But CPU can execute one process at a time

CPU scheduling algo will decide which process gets the CPU next.

CPU-I/O Burst Cycle

Process execution consist of a cycle of CPU execution (CPU burst) and I/O wait(I/O burst) .

Processes alternate back and forth between these two states.

The last CPU burst will end with a system request to terminate execution rather than with an I/O burst.

#include<stdio.h>	CPU
Int main()	Burst
{	|
Int x=1,y,fact=1;	|
printf(“\n enter the limit”);	I/O
Scanf(“%d”,&y);	|
for(x=1;x<y;x++)	CPU
{	Burst
fact=fact*x;	|
}	|
printf(“\n fact=%d”,fact);	I/O Burst
}	CPU burst [terminate process]

Pre-emptive and Non Pre-emptive scheduling

Pre-emptive scheduling

CPU scheduling decision may take place under the following circumstances:

1. when a process switches from the running state to the waiting state.
2. when a process switches from the running state to the ready state.
3. when a process switches from the waiting state to the ready state.
4. when a process terminate.

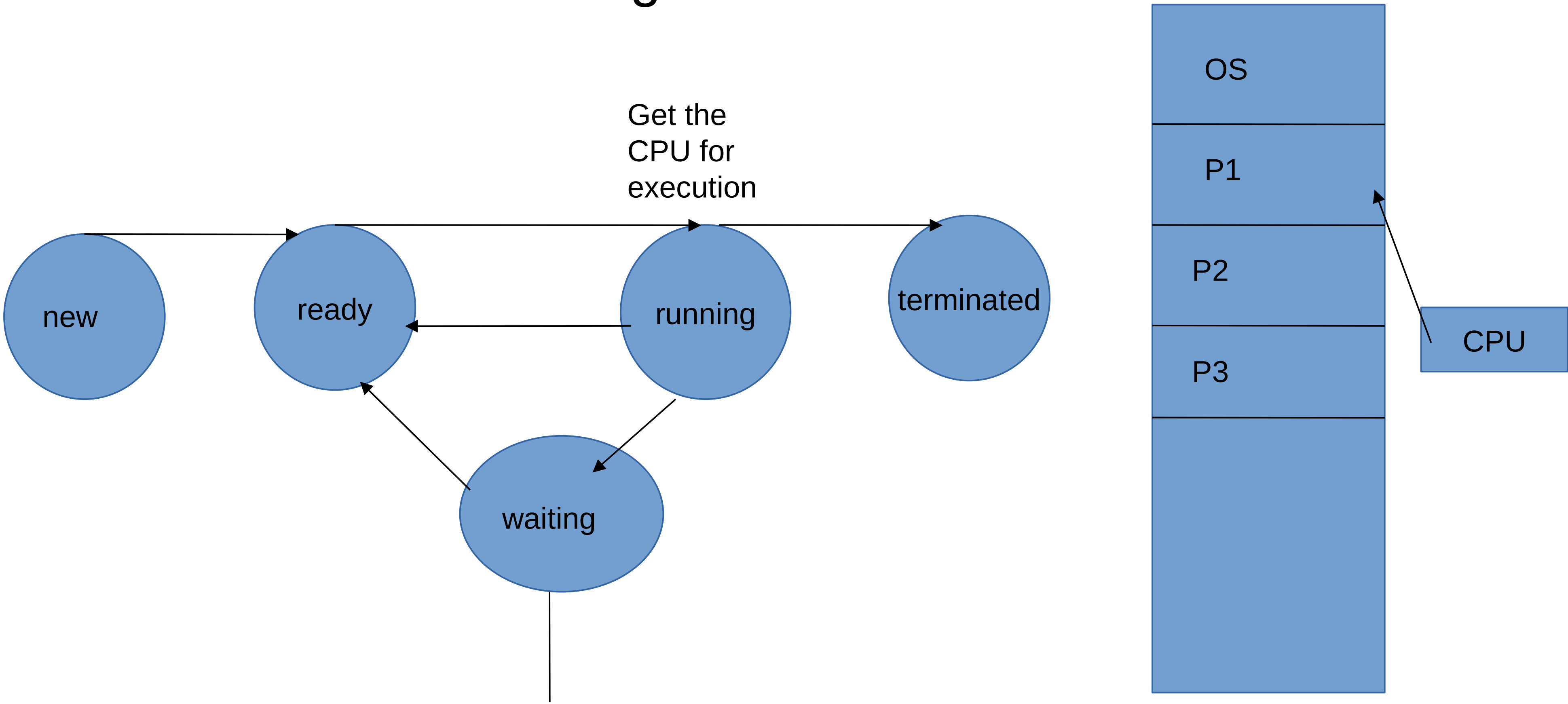
Pre-emptive and Non Pre-emptive scheduling

Pre-emptive scheduling

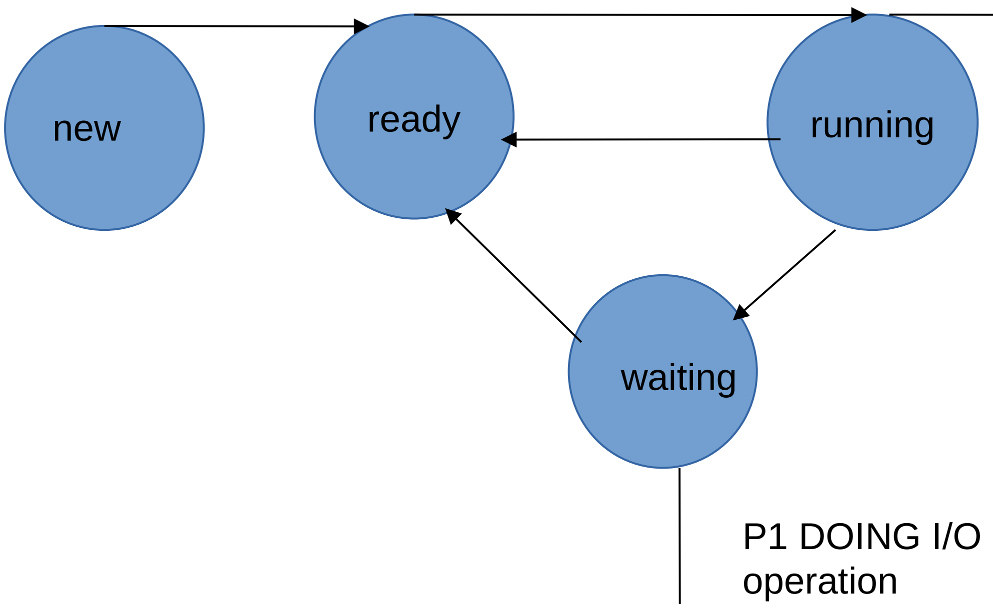
CPU scheduling decision may take place under the following circumstances:

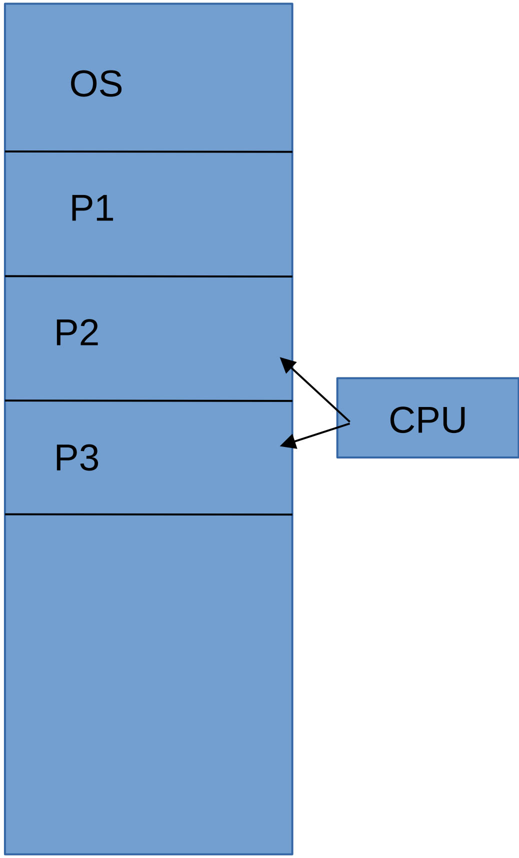
1. when a process switches from the running state to the waiting state.
2. when a process switches from the running state to the ready state.
3. when a process switches from the waiting state to the ready state.
4. when a process terminate.

when a process switches from the running state to the waiting state.

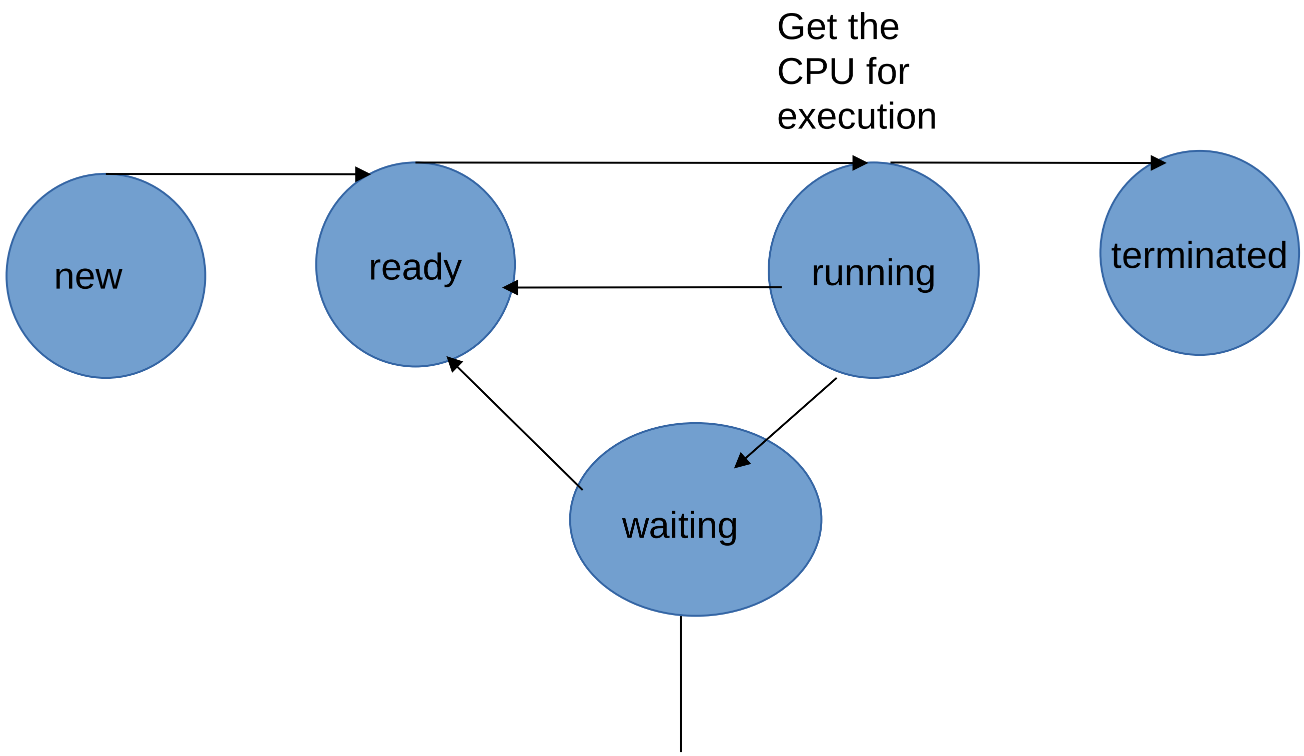


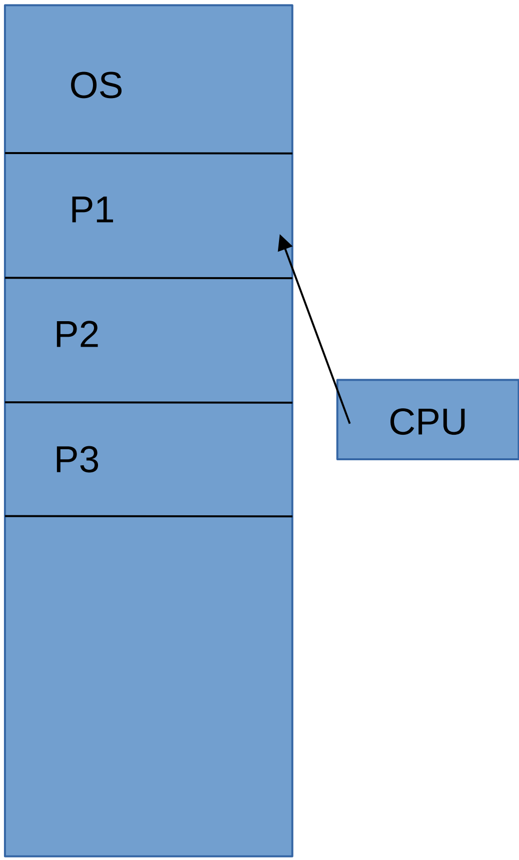
when a process switches from the running state to the waiting state.



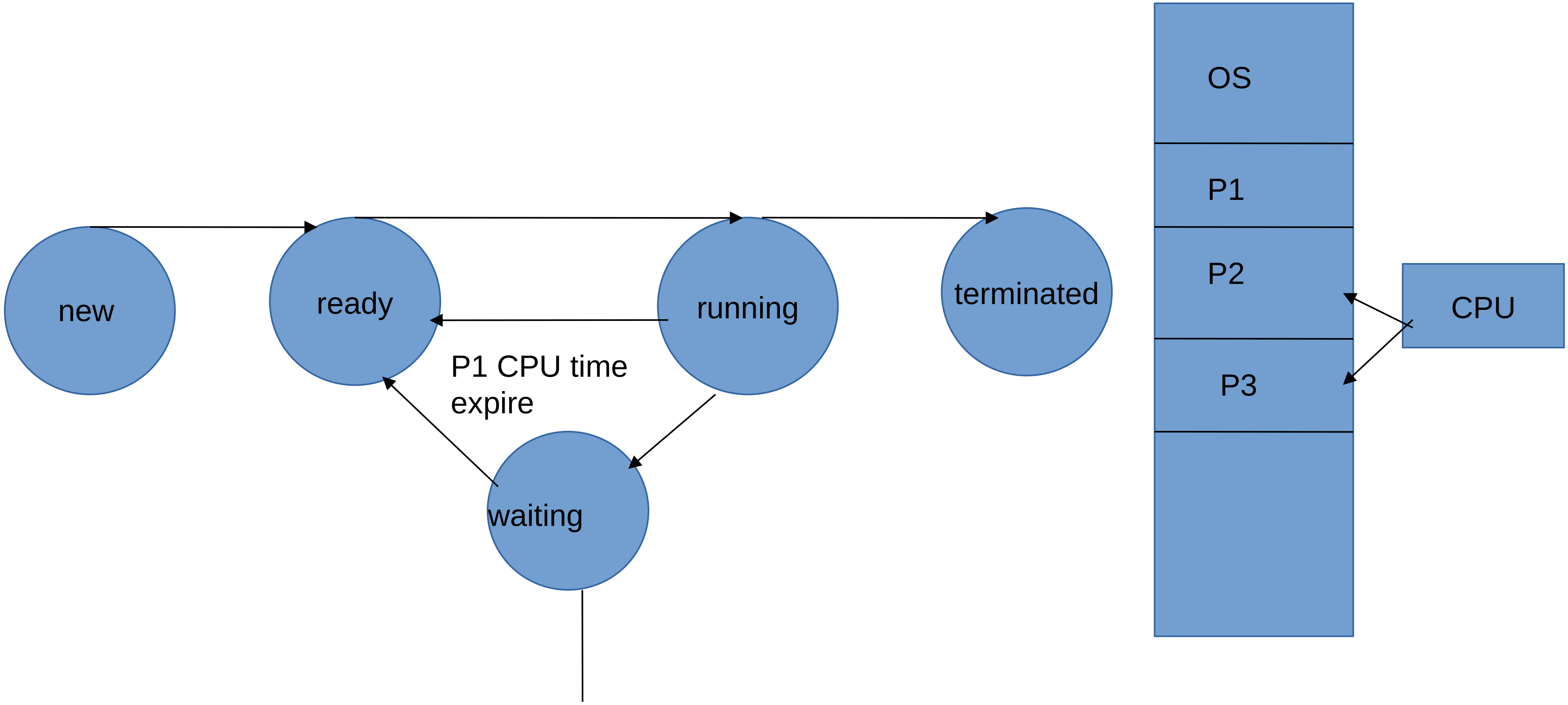


when a process switches from the running state to the ready state.





when a process switches from the running state to the ready state.



Pre-emptive and Non Pre-emptive scheduling

Non **Pre-emptive scheduling**

CPU scheduling decision may take place under the following circumstances:

1. when a process switches from the running state to the waiting state.

2. when a process terminate.

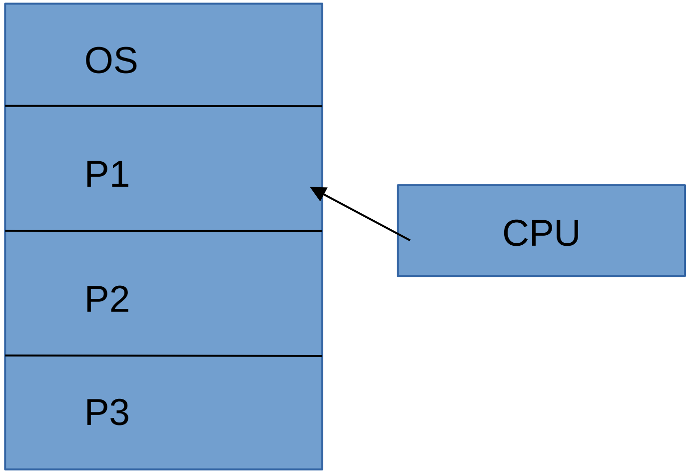
Disadvantage of Pre-emptive scheduling

1. Data inconsistency may arise. E.g. When two processes share data , one may be in the midst of updating the shared data when it is pre-empted and the second process is run. The second process may try to read the data which are currently in an inconsistent state.

2. During the processing of a system call, the kernel may be busy updating some kernel data structure. If it is pre-empted before the complete updating is done , they are no longer in a consistent state.

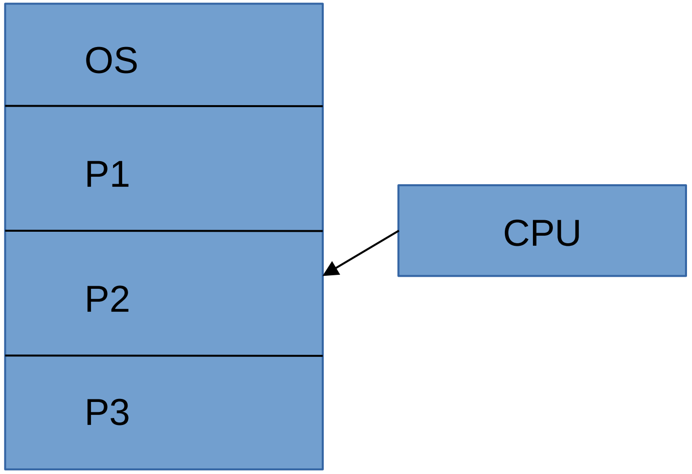
Dispatcher

Dispatcher is a module that gives control of the CPU to the process selected by the short-term scheduler.



Dispatcher

Dispatcher is a module that gives control of the CPU to the process selected by the short-term scheduler.



Short term scheduler selects p2

Dispatcher functions:

1. switching context.

2. switching to user mode.

3. jumping to the proper location in the user program to restart the program.

The time it takes for the dispatcher to stop one process and then start another running is known as dispatcher latency.

Scheduling criteria

1. CPU utilization. 0%-100%
2. Throughput : number of processes completed per unit time.
3. turnaround time: the interval from the time of submission to the time of completion for a process.
4. waiting time: sum of the periods spent waiting in the ready queue.
5. response time: from the submission of a request until the first response is produced.

CPU scheduling Algorithm

First come first served(FCFS)

Non-preemptive

The process that request the CPU first is allocated the CPU first.

It is implemented easily with a FIFO queue.

The average waiting time under FCFS is quite long

Consider the following situation with arrival time for each process is 0

Process	cpu time
P1	24
P2	3
P3	3

Average waiting time

Waiting time for P1=0

Waiting time for p2=24

Waiting time for p3= 27

Average waiting time= (0+24+27)/3=17

Consider the following situation with arrival time for each process is 0

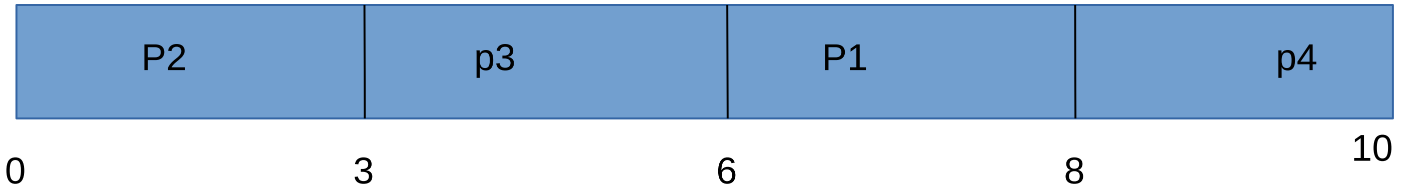
Process	cpu time
P2	3
P3	3
P1	24

What is the average waiting time?

Consider the following situation

Process	cpu time	arrival time
P2	3	0
P3	3	1
P1	2	2
P4	5	2

What is the average waiting time?



Convey effect

Consider a situation where FCFS is used and there is a CPU bound process and many I/O bound processes.

The CPU bound process will get the CPU and hold it.

During this time all the other processes will finish their I/O operation and move into the ready queue.

Now the I/O devices are idle .

When the Processes finish their CPU operation they come back to the I/O queue to performed I/O operation.

Now the CPU sit idle.

This is called convey effect.

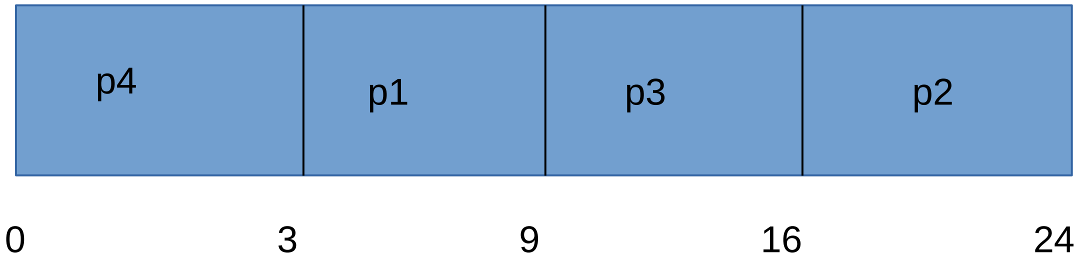
Shortage job first scheduling(SJF)

This algorithm associates with each process the length of the process next CPU time.

When the CPU is available all the processes in the ready queue are examined and the CPU is given to the process that has the smallest next CPU time.

Consider the processes with arrival time 0.

Process	CPU time
P1	6
P2	8
P3	7
P4	3



Average waiting time

Waiting time for P1=3

Waiting time for p2=16

Waiting time for p3=9

Waiting time for p4=0

Average waiting time=(3+16+9+0)/4=7

What is the average waiting time for the above problem using FCFS?????

Advantage & Disadvantage

Adv: It gives the the minimum average waiting time.

Dis adv:

It is difficult to get the length of the next CPU time.

Pre-emptive SJF Algorithm

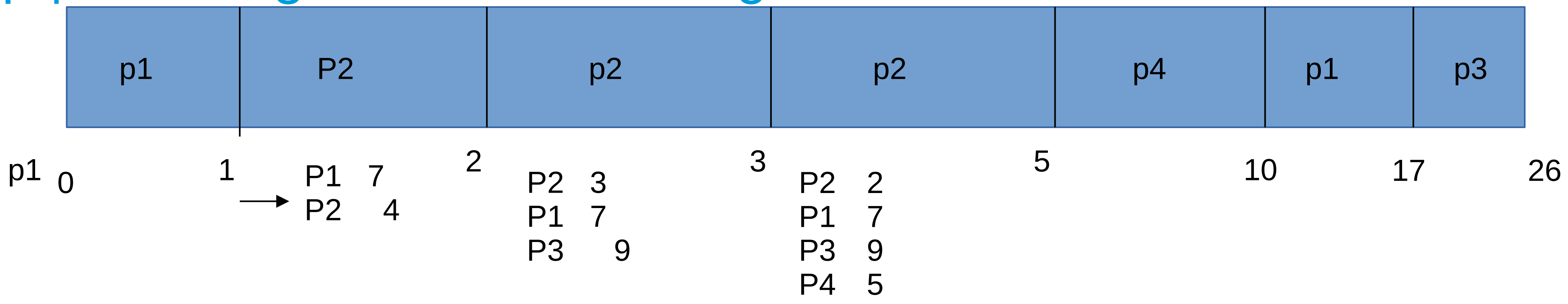
When a new process arrives at the ready queue while a previous process is executing, the new process may have a shorter CPU time than what is left of the currently executing process.

A pre-emptive SJF will pre-empt the currently executing process and the CPU will be given to the new process.

But a non pre-emptive SJF algorithm will allow the currently running process to finish its CPU time.

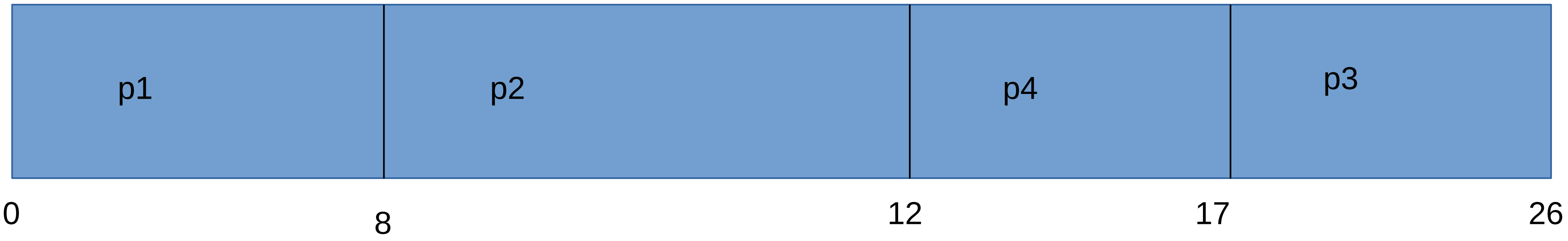
Consider the following situation

Process	arrival time	CPU time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



Average waiting time?

Apply Non pre emptive SJF in the same data and find out average waiting time



Pre-emptive SJF Algorithm

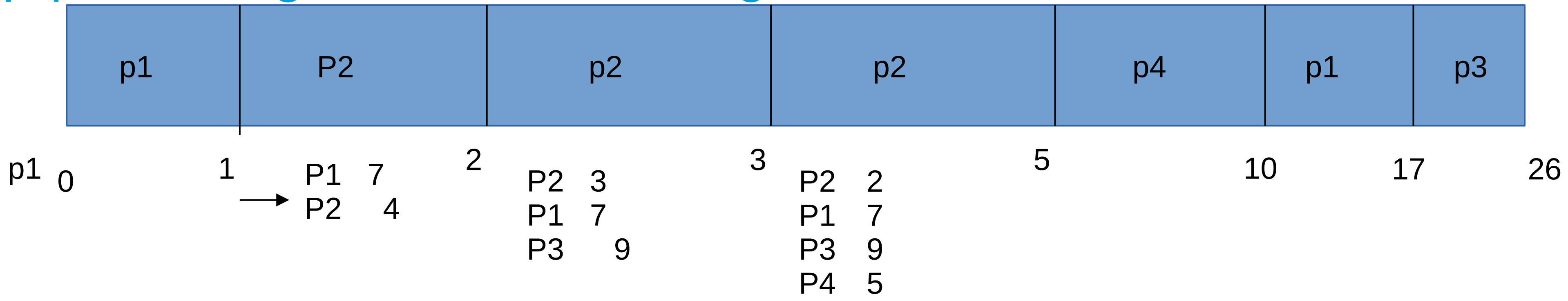
When a new process arrives at the ready queue while a previous process is executing, the new process may have a shorter CPU time than what is left of the currently executing process.

A pre-emptive SJF will pre-empt the currently executing process and the CPU will be given to the new process.

But a non pre-emptive SJF algorithm will allow the currently running process to finish its CPU time.

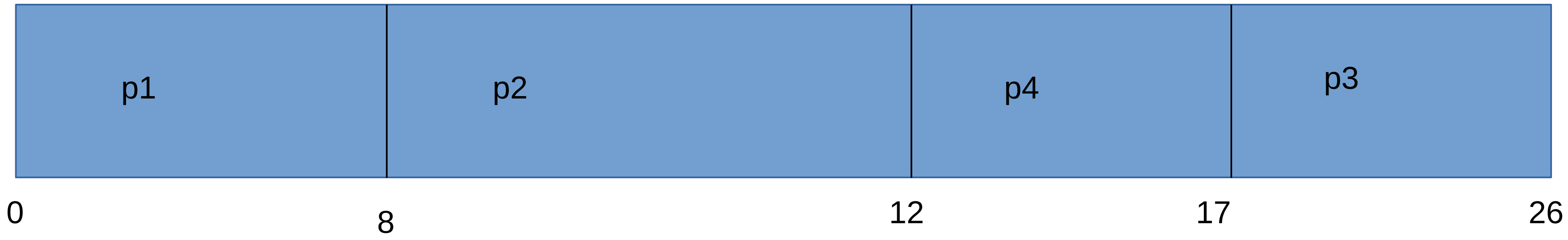
Consider the following situation

Process	arrival time	CPU time
P1	0	8
P2	1	4
P3	2	9
P4	3	5



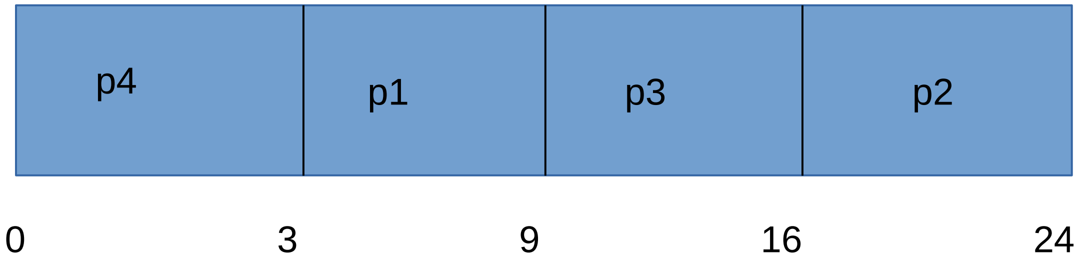
Average waiting time?

Apply Non pre emptive SJF in the same data and find out average waiting time



Consider the processes with arrival time 0.

Process	CPU time
P1	6
P2	8
P3	7
P4	3



Average waiting time

Waiting time for P1=3

Waiting time for p2=16

Waiting time for p3=9

Waiting time for p4=0

Average waiting time=(3+16+9+0)/4=7

What is the average waiting time for the above problem using FCFS?????

Advantage & Disadvantage

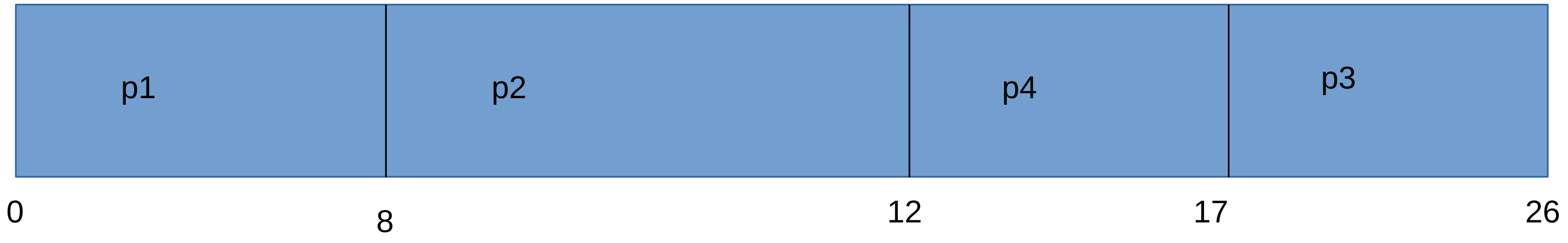
Adv: It gives the the minimum average waiting time.

Dis adv:

It is difficult to get the length of the next CPU time.

Average waiting time?

Apply Non pre emptive SJF in the same data and find out average waiting time

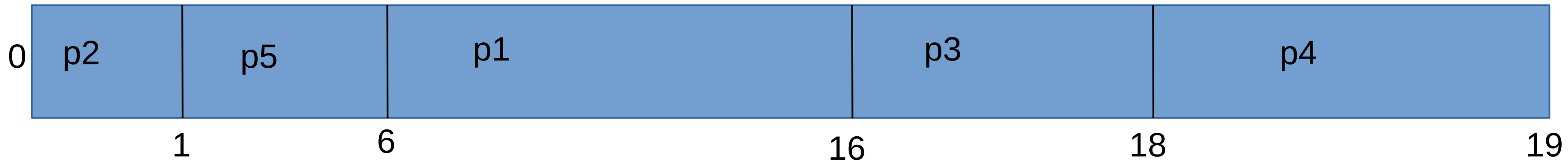


Priority scheduling

In this algorithm a priority is associated with each process and the CPU is assigned to the process with the highest priority .

Consider the following situation with arrival time 0

Process	CPU time	priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2



Average waiting time?

8.2

Priority scheduling may be either pre-emptive or non pre-emptive

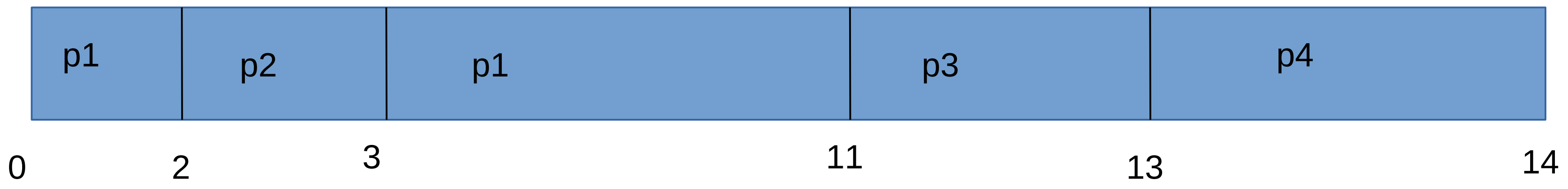
When a new process arrives at the ready queue while a previous process is executing, the new process may have a higher priority than the currently executing process.

A pre-emptive priority scheduling algorithm will pre-empt the currently executing process, and the cpu will be given to the new process.

But a non pre-emptive priority scheduling algorithm will allow the currently running process to finish its CPU time

Consider the following situation

Process	CPU time	priority	arrival time
P1	10	3	0
P2	1	1	2
P3	2	3	2
P4	1	4	3



disadvantage

Blocking

A process is ready to run but it is not able to get the CPU is said to be blocked.

It could happened that a low priority process wait indefinitely for the CPU.

This is true in heavily loaded systems.

It is rumoured that when they shut down the IBM 7094 at MIT in 1973 they found a low priority process that had been submitted in 1967 and not yet been run.

One solution to this problem is aging.

Gradually increase the priority of the processes that wait in the system for a long time.

It is rumoured that when they shut down the IBM 7094 at MIT in 1973 they found a low priority process that had been submitted in 1967 and not yet been run.

One solution to this problem is aging.

Gradually increase the priority of the processes that wait in the system for a long time.

Round robin algorithm

This algorithm is designed specifically for time sharing system.

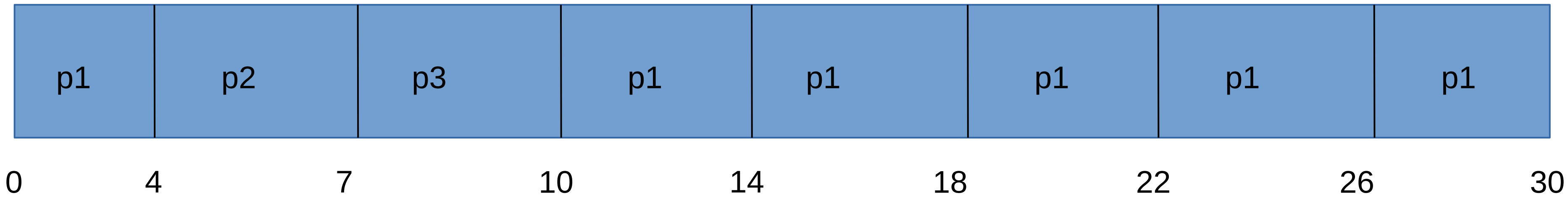
A small unit of time called a time slice or quantum is defined.

The ready queue is treated as a circular queue and the CPU goes around this queue, allocating the CPU to each process for a time interval of up to 1 time quantum .

New processes are added to the tail of the queue

Consider the following situation with arrival time 0

Process	CPU time	
P1	24	time slice =4
P2	3	
P3	3	



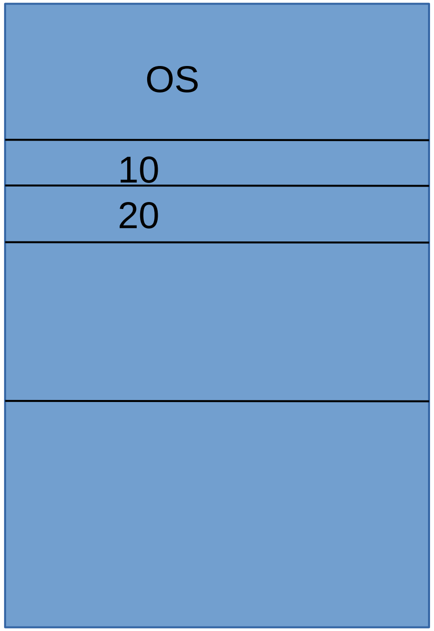
5.66

Time slice high and low???

Memory Management

```
Int main()
{
    Int A,B;
    A=10;
    B=20;
    B=A+B;
    }
```

A	0X1234
B	0x1238



A and B are logical addresses and
0x1234 and 0x1238 are absolute addresses

Memory management unit of OS maps A---->
0x1234 and
B---->0x1238

Both the CPU and I/o system interact with the memory.

Before a user program is finally loaded into the memory and executed it goes through several stages.

- Generally addresses in the source program are symbolic (such as A).
- A compiler typically binds these symbolic addresses to relocatable addresses .
- The linkage editor/loader will bind these relocatable addresses to the absolute addresses (such as 0x1234).
- Each binding is nothing but a mapping from one address space to another.

Whatever may be the number of address bindings that may be required, the user program must finally be mapped to absolute address and loaded into memory to be executed.

A typical instruction execution cycle will fetch an instruction from memory.

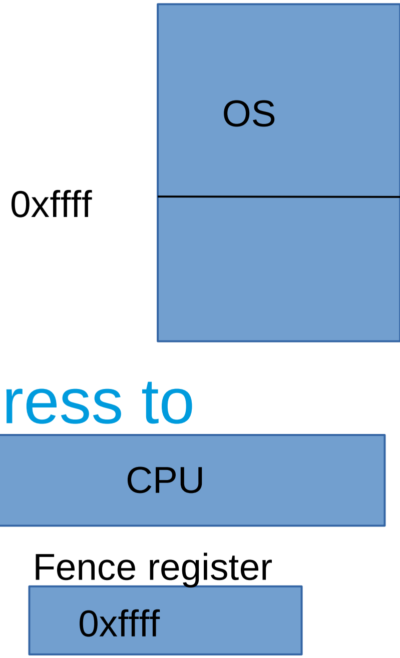
The instruction will be decoded and it may result in fetching the operands from the memory.

After executing the instruction it may also require storing of the result in the memory.

To protect the user from accessing the system memory (part of the memory where the os is stored) a fence register is used.

The fence register stores the fence address.

Every memory reference is checked against this this address to verify that it is indeed a legal memory reference.

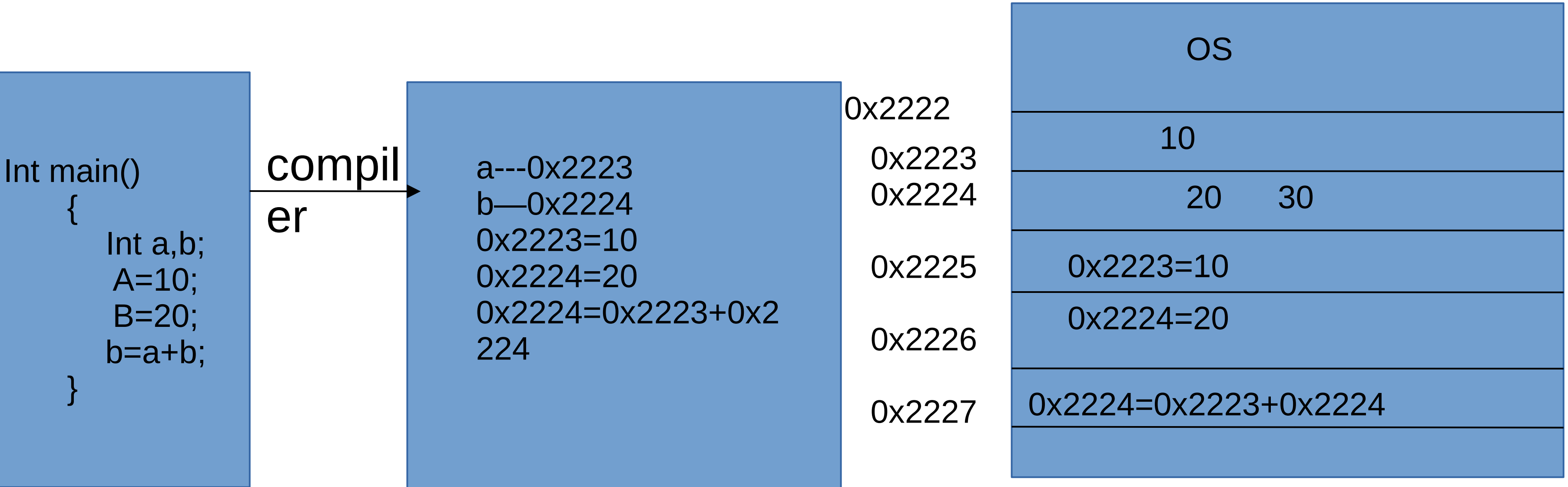


There may be 3 ways of mapping the user program to absolute address

– **Binding at compile time**

If the fence address is known at compile time , absolute code can be generated then

It only requires for the code to be loaded and executed.



Any disadvantage

If the fence address changes it is required to recompile.

Binding at load time

Here the compiler produces only relocatable code

Binding to absolute address is done during load time .

In case of changing in fence address the program only needs to be reloaded.

```
Int main()
{
        Int A,B;
        A=10;
        B=20;
        B=A+B;
}
```

Compiler

Relocatable code
Lets take a dummy code(start)
A—start+0
B---start+1
Start=10
Start+1=20
start+1=start+start+1

LOAD

Start-0x333
A-0x333
B-0x333+1
0x333=10
0x334=20
0x334=0x333
+0x334

0x333	10
0x334	20 30
0x335	0x333=10
0x336	0x334=20
0x336	0x334=0x333+0x334

Binding at load time

Here the compiler produces only relocatable code

Binding to absolute address is done during load time .

In case of changing in fence address the program only needs to be reloaded.

```
Int main()
{
        Int A,B;
        A=10;
        B=20;
        B=A+B;
}
```

Compiler

Relocatable code
Lets take a dummy code(start)
A—start+0
B---start+1
Start=10
Start+1=20
start+1=start+start+1

LOAD

Start-0x333
A-0x333
B-0x333+1
0x333=10
0x334=20
0x334=0x333+0x334

0x333	10
0x334	20 30
0x335	0x333=10
0x336	0x334=20
0x336	0x334=0x333+0x334

Any Disadvantages

The fence address can not change when program is executing.

Binding at execution time
dynamic binding

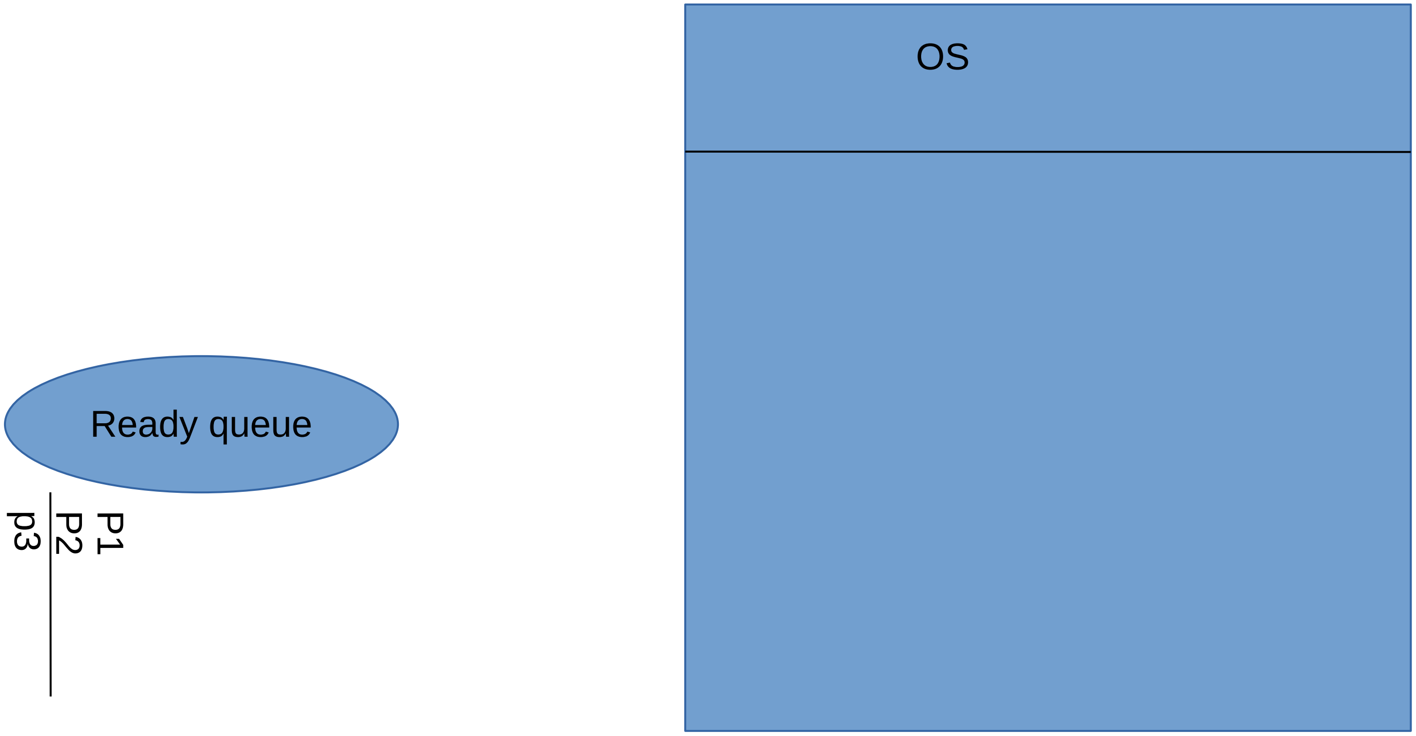
The fence register is now called base register and every address generated by a user process at the time it is sent to memory.

The user never sees the real physical addresses.

The user program deals with logical addresses. The memory management unit converts logical addresses into physical addresses.

The user generates only logical addresses.

Allocation of memory partition



Fixed partition (MFT)

Also known as multiprogramming with a fixed number of tasks.

Divide the memory into a number of fixed size partitions.

Each partition may contain exactly one process.

There fore the degree of multi programming is bound by the number of partitions.

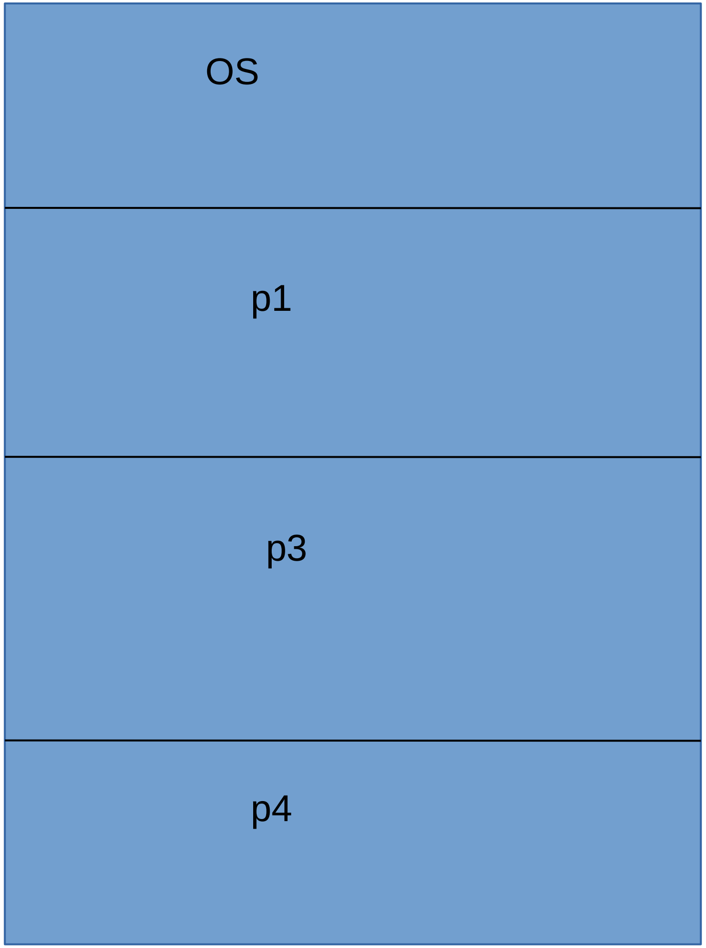
When a partition is free a process is selected from the ready queue and is loaded into the free partition.

When a process terminates the partition becomes available for another process.

Ready queue

P2
P5
p6

Degree of multiprogramming is 3



Fixed partition (MFT)

Also known as multiprogramming with a fixed number of tasks.

Divide the memory into a number of fixed size partitions.

Each partition may contain exactly one process.

There fore the degree of multi programming is bound by the number of partitions.

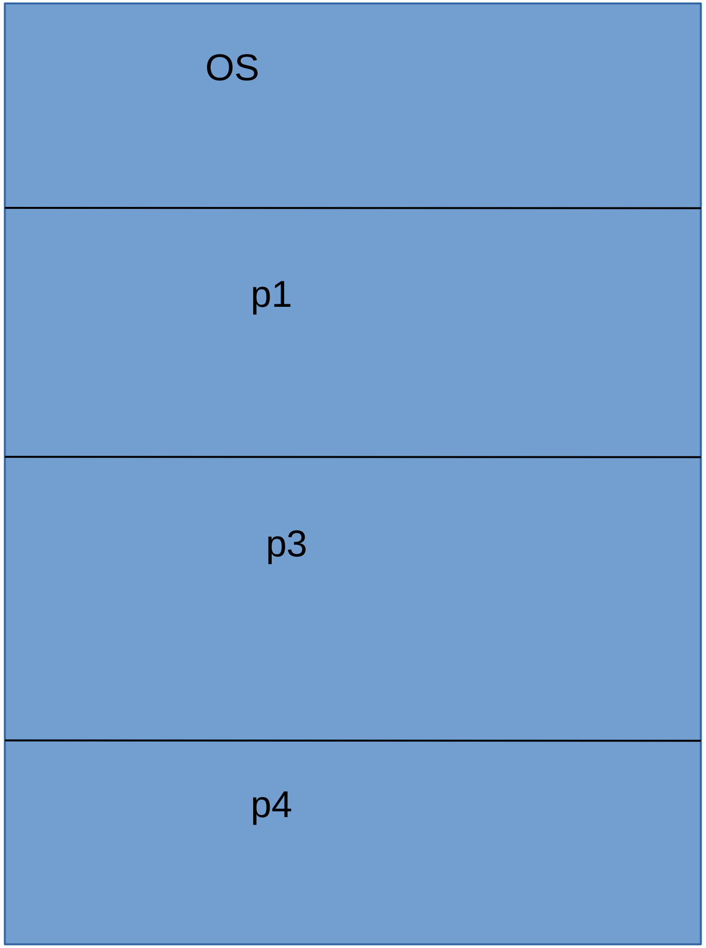
When a partition is free a process is selected from the ready queue and is loaded into the free partition.

When a process terminates the partition becomes available for another process.

Ready queue

P2
P5
p6

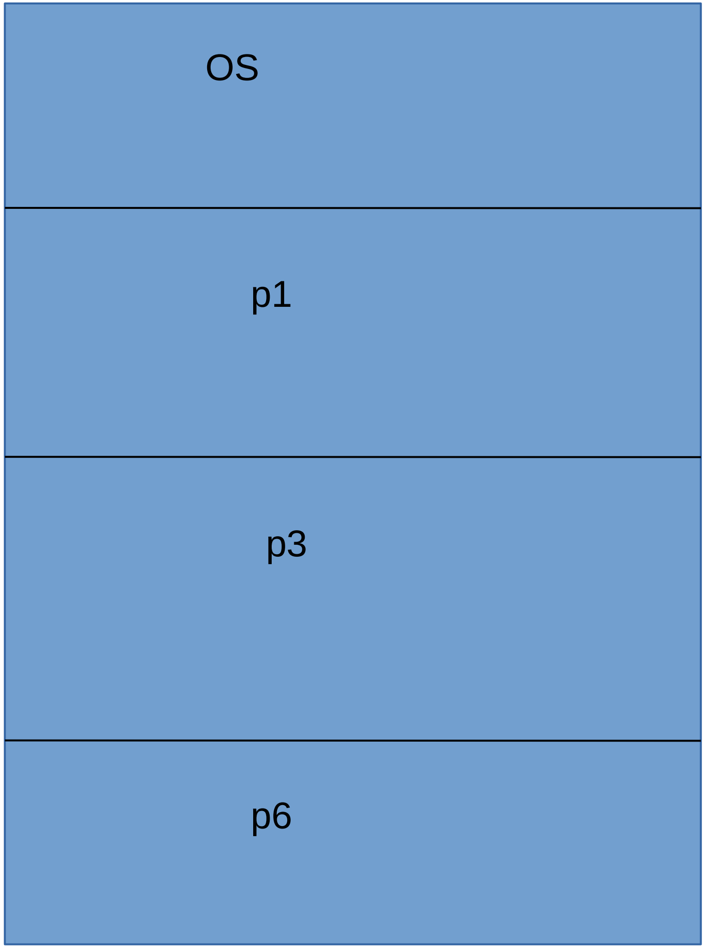
Degree of multiprogramming is 3



Ready queue

P2
P5

Degree of multiprogramming is 3



32 kB memory can be divided into regions of the following sizes:

Os----10KB

Very large programs----12 KB

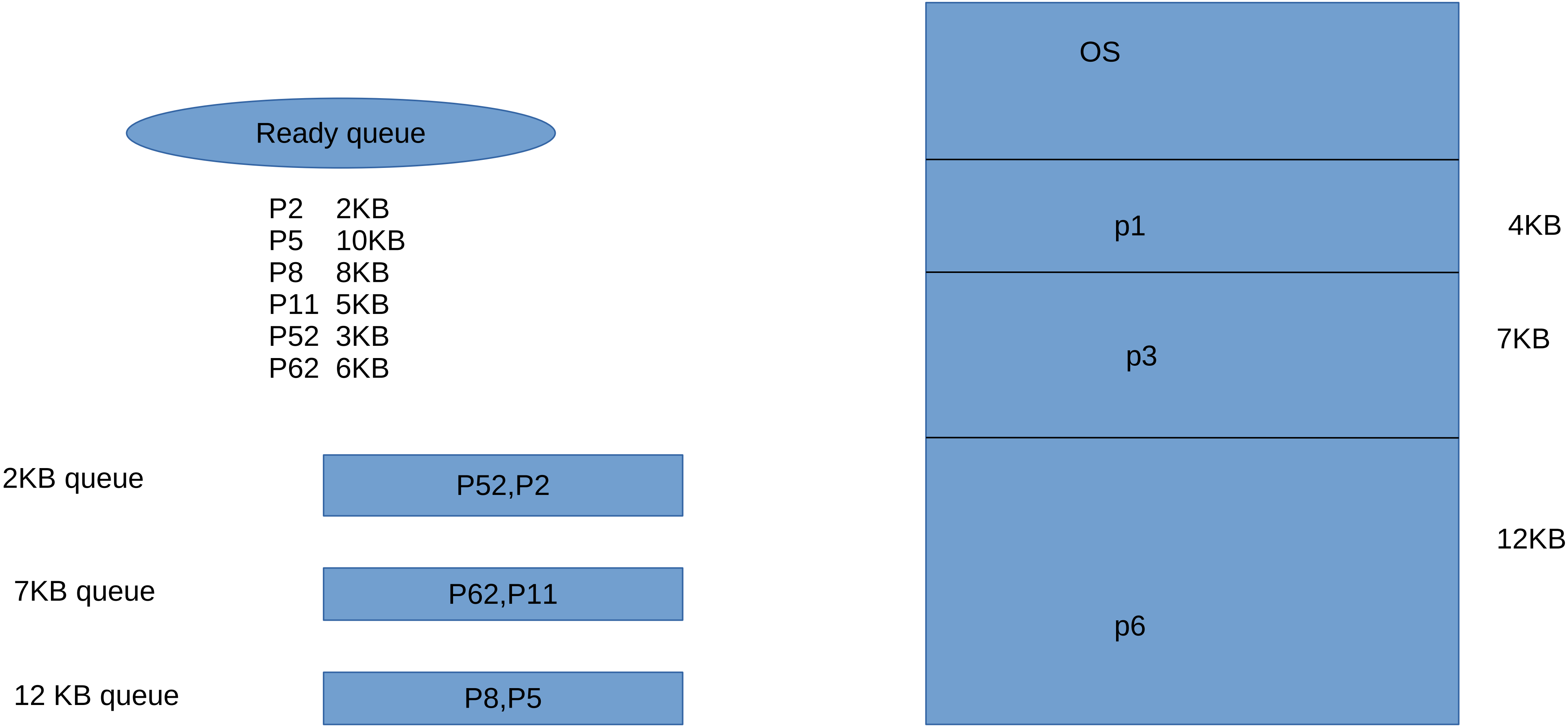
Average programs-------6 KB

Small programs----------4 KB

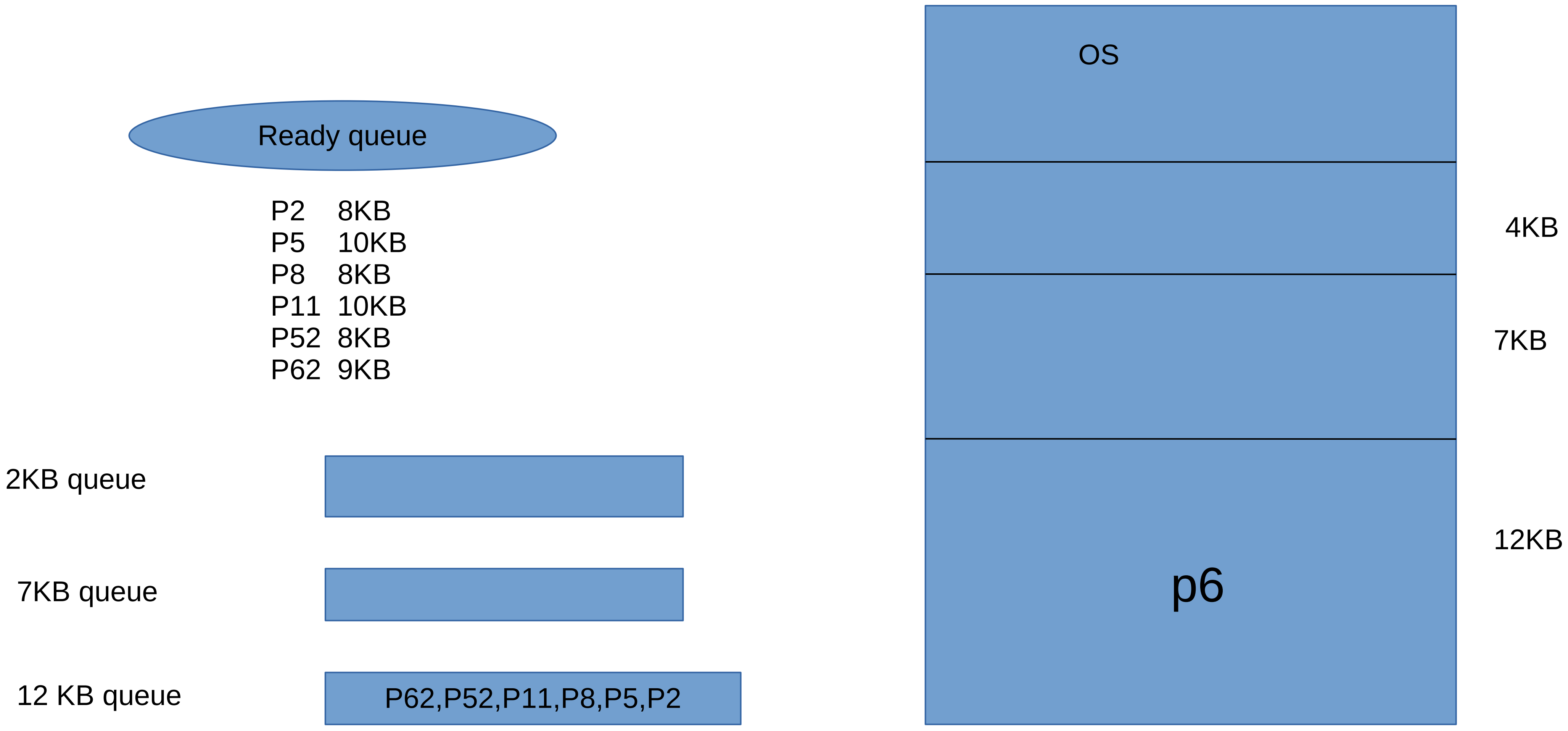
One variation is have multiple queues, a queue for each region.

The user specifies the maximum amount of memory required or the OS can attempt to determine the memory requirements automatically.

Accordingly a process could be assigned to queue.



Ready queue
P2 2KB
P5 10KB
P8 8KB
P11 5KB
P52 3KB
P62 6KB
2KB queue
P52,P2
7KB queue
P62,P11
12 KB queue
P8,P5
OS
p1
4KB
p3
7KB
p6
12KB



Ready queue
P2 8KB
P5 10KB
P8 8KB
P11 10KB
P52 8KB
P62 9KB
2KB queue
7KB queue
12 KB queue
P62,P52,P11,P8,P5,P2
OS
4KB
7KB
12KB
p6

Disadvantage

Internal and external fragmentation may occur.

Dynamic Allocation
Variable partition Allocation (MVT)

The OS keeps a table indicating which part of memory are available (called holes) are available and which are occupied.

Initially all memory is available for user processes.

When a process arrives we select a hole which is large enough to hold this process.

We allocate as much memory is required for the process and the rest is kept as a hole which can be used for later requests.

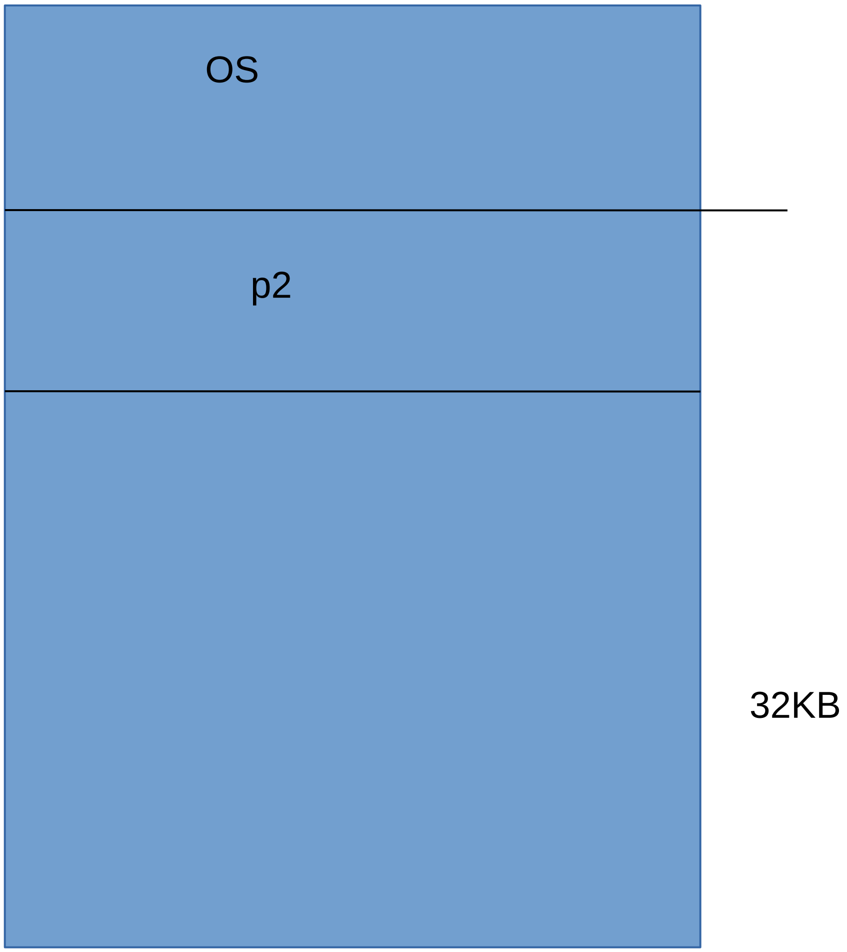
If a hole large enough for this process cannot be found, this process waits until some other process(es) finishes and a large enough hole is available.

If a new hole is adjacent to other holes OS merge the adjacent holes to form a large hole.

Ready queue

P2 8KB
P5 10KB
P8 8KB
P11 10KB
P52 8KB
P62 9KB

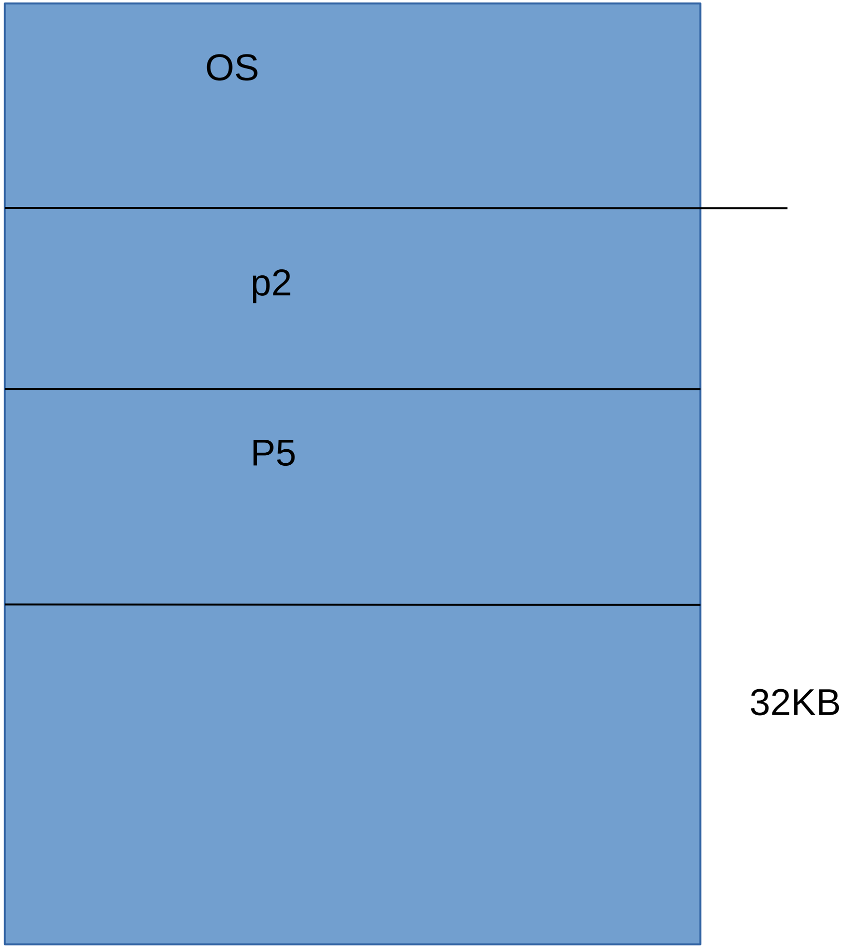
Free---24KB



Ready queue

P5 10KB
P8 8KB
P11 10KB
P52 8KB
P62 9KB

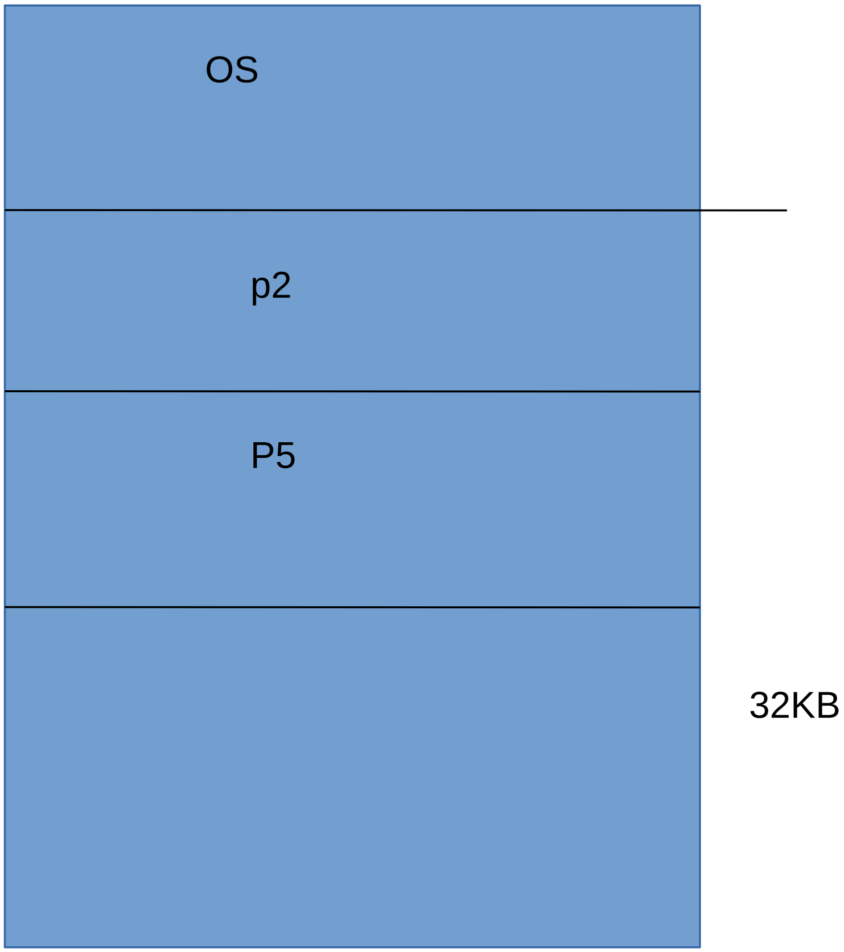
Free---24KB



Ready queue

P8 8KB
P11 10KB
P52 8KB
P62 9KB

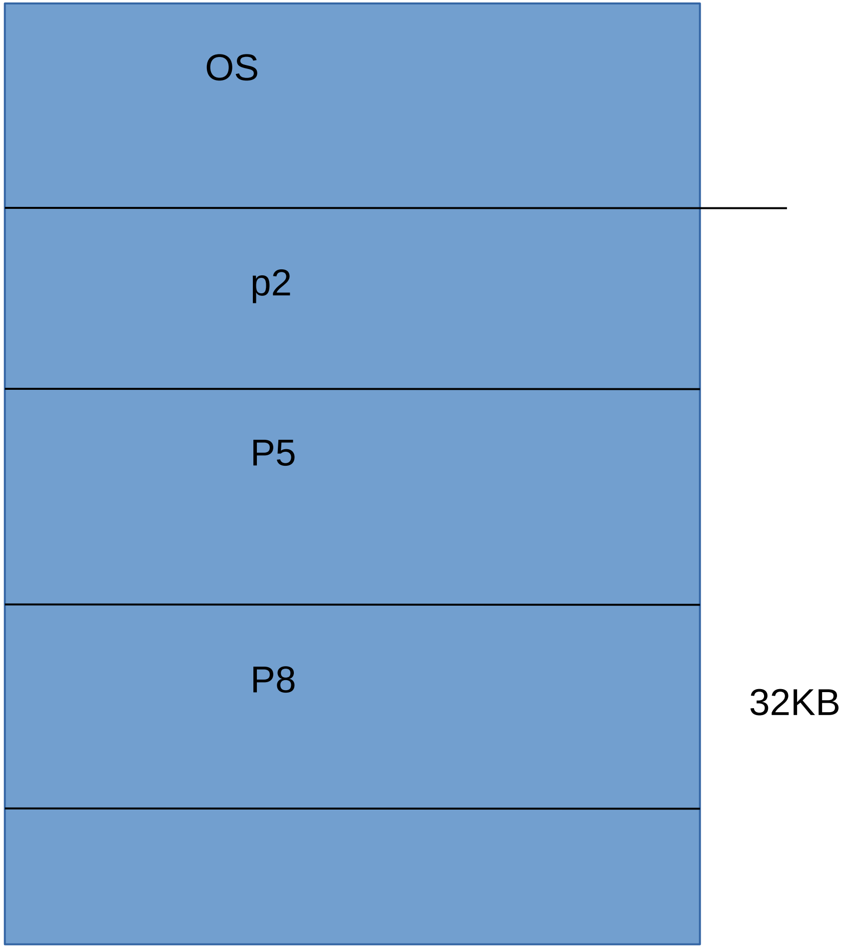
Free---14KB



Ready queue

P8 8KB
P11 10KB
P52 8KB
P62 9KB

Free---14KB

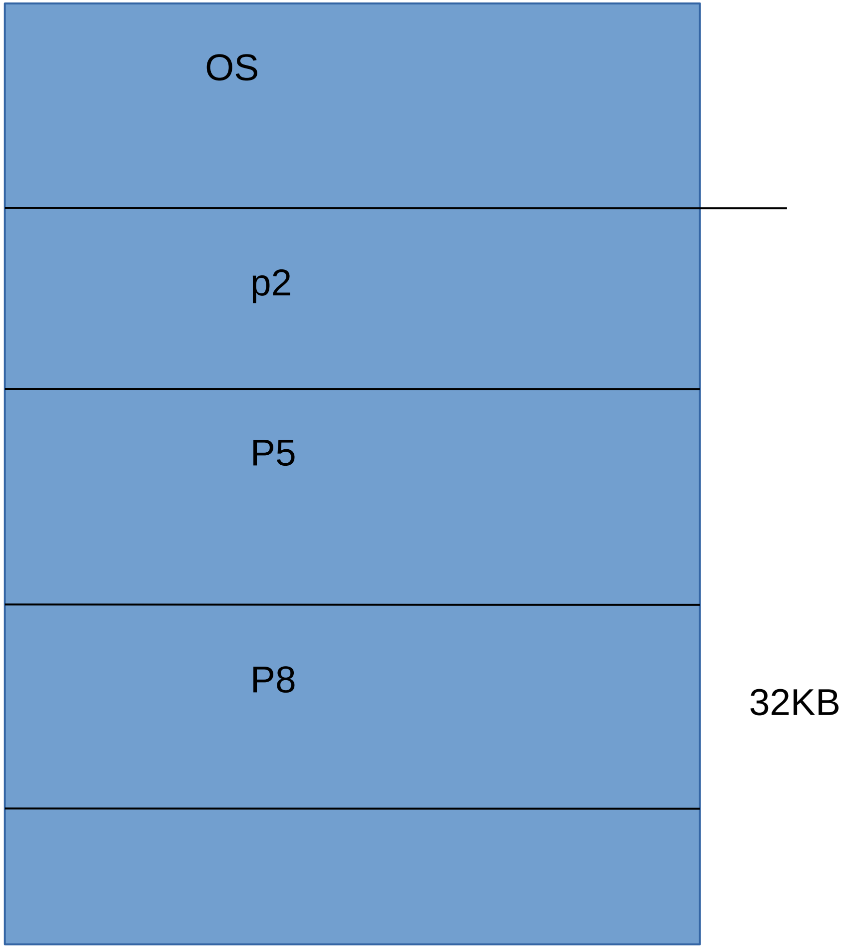


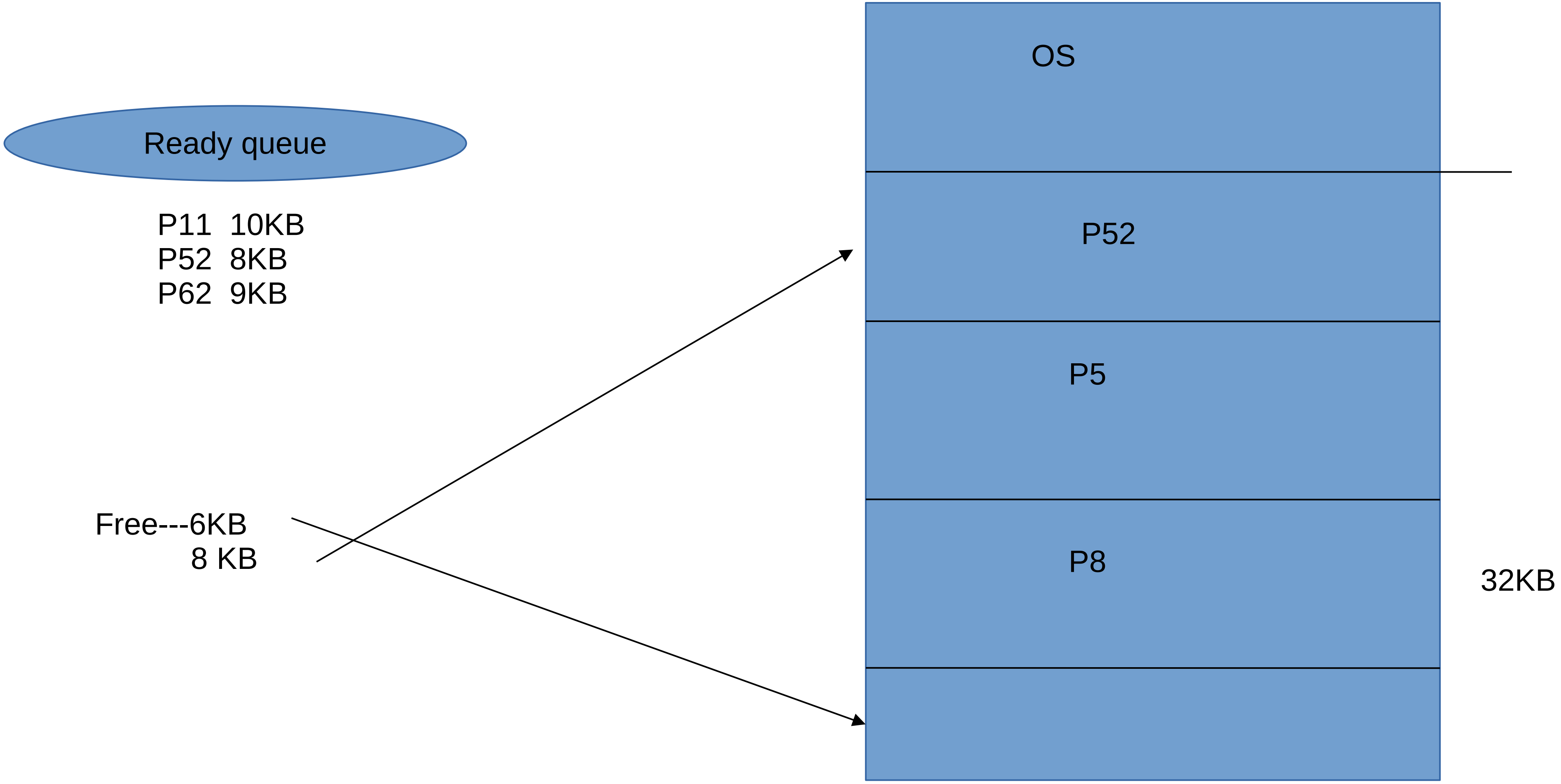
Ready queue

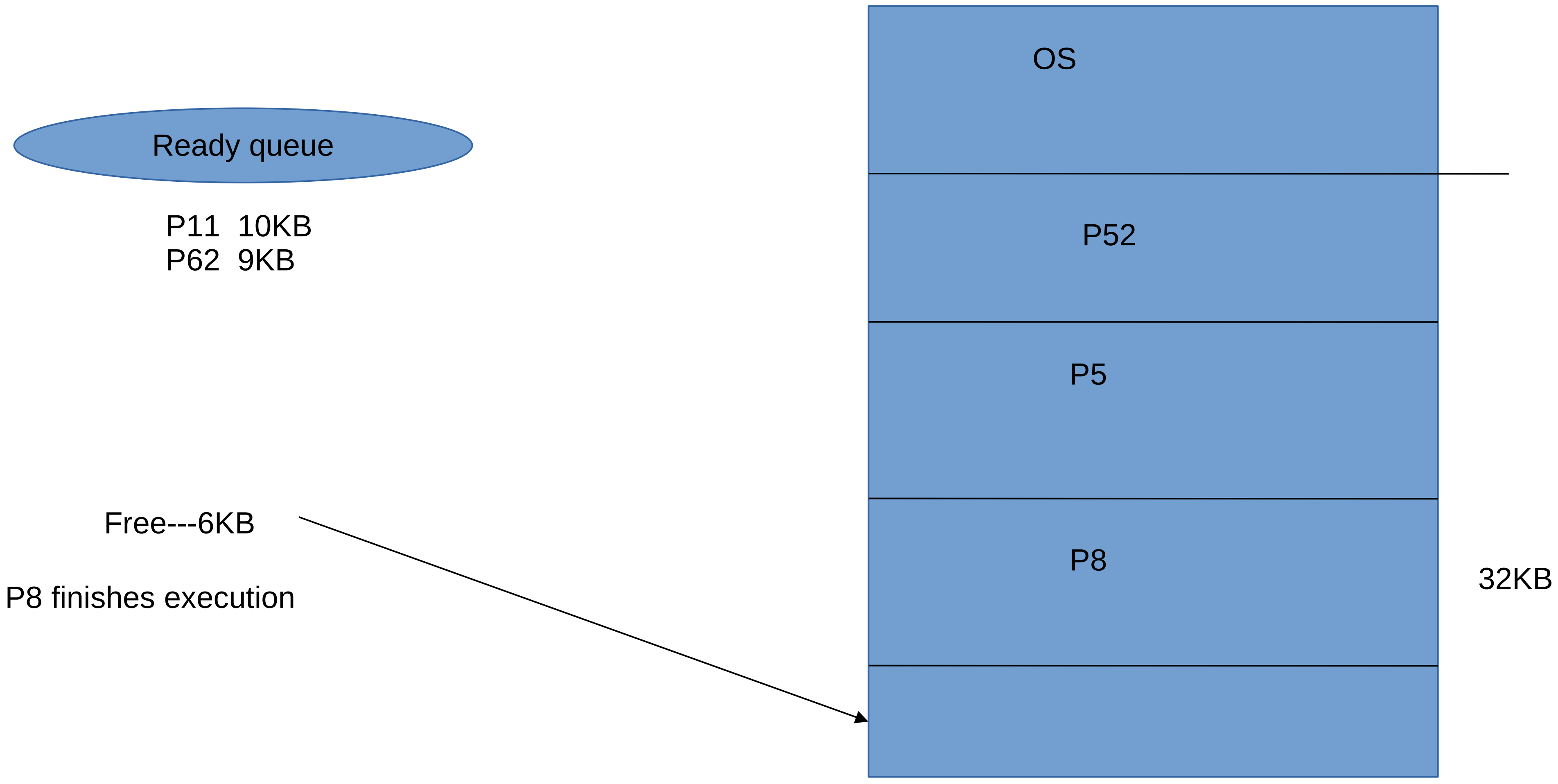
P11 10KB
P52 8KB
P62 9KB

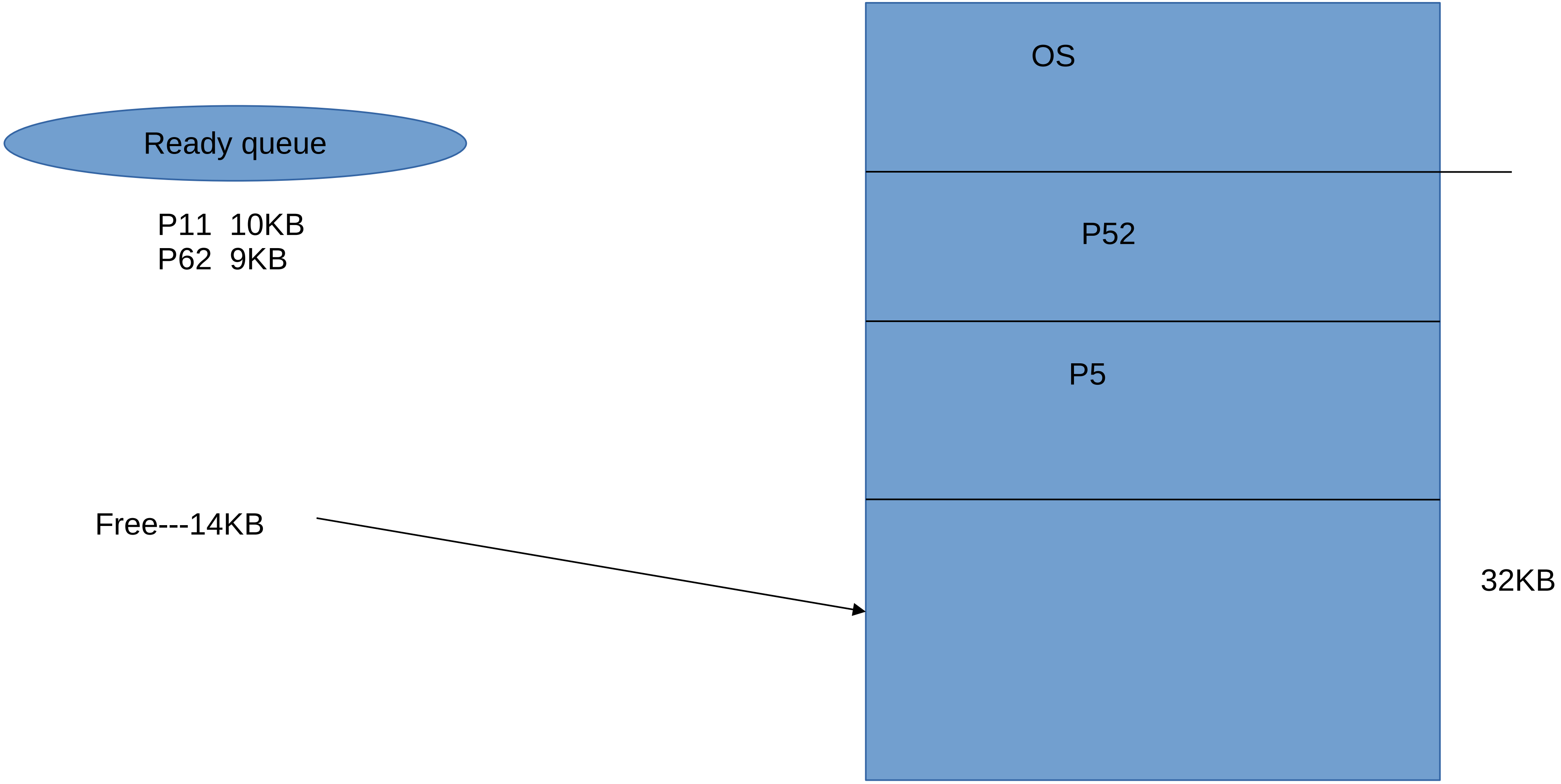
Free---6KB

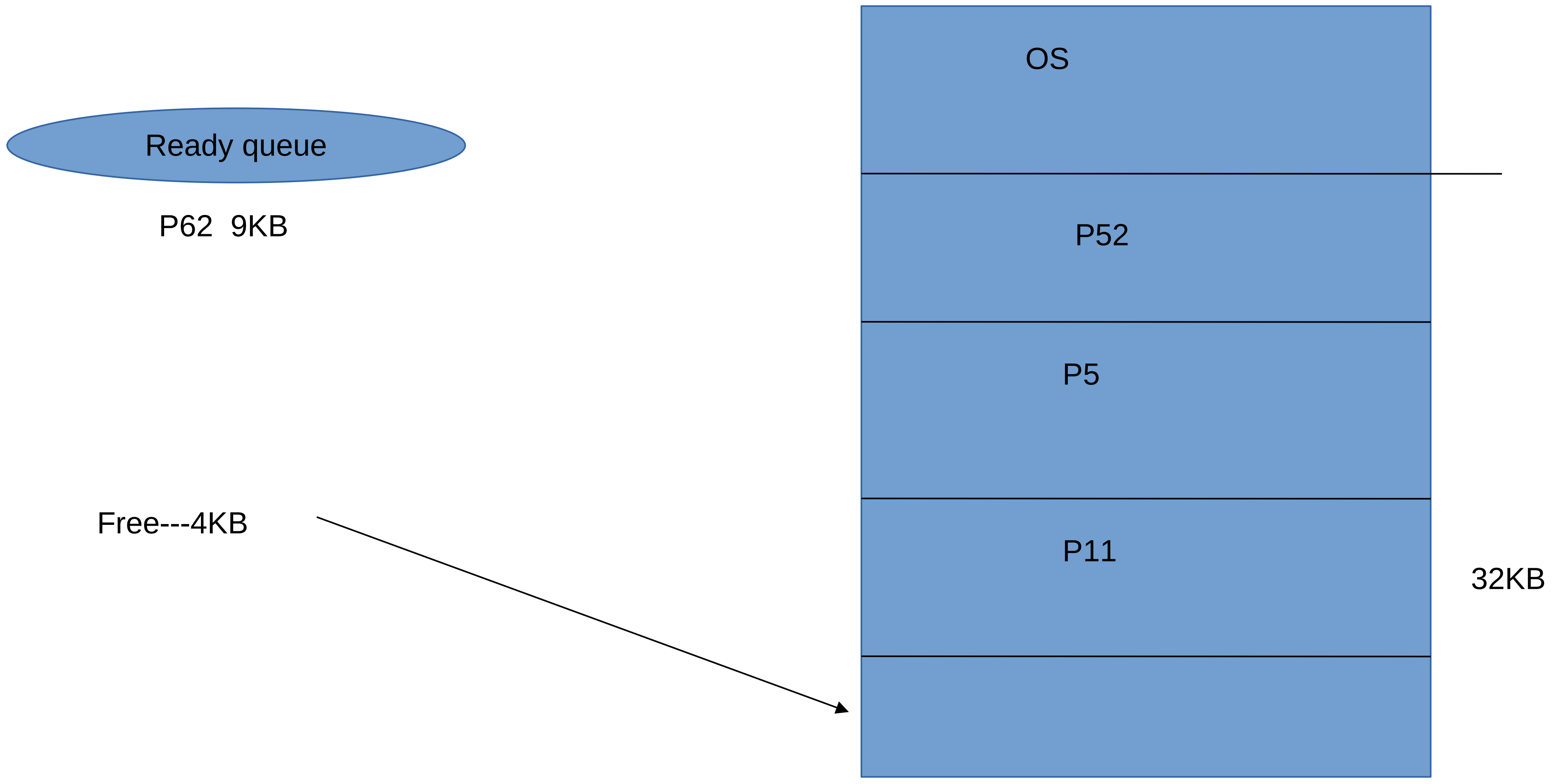
P2 finishes execution

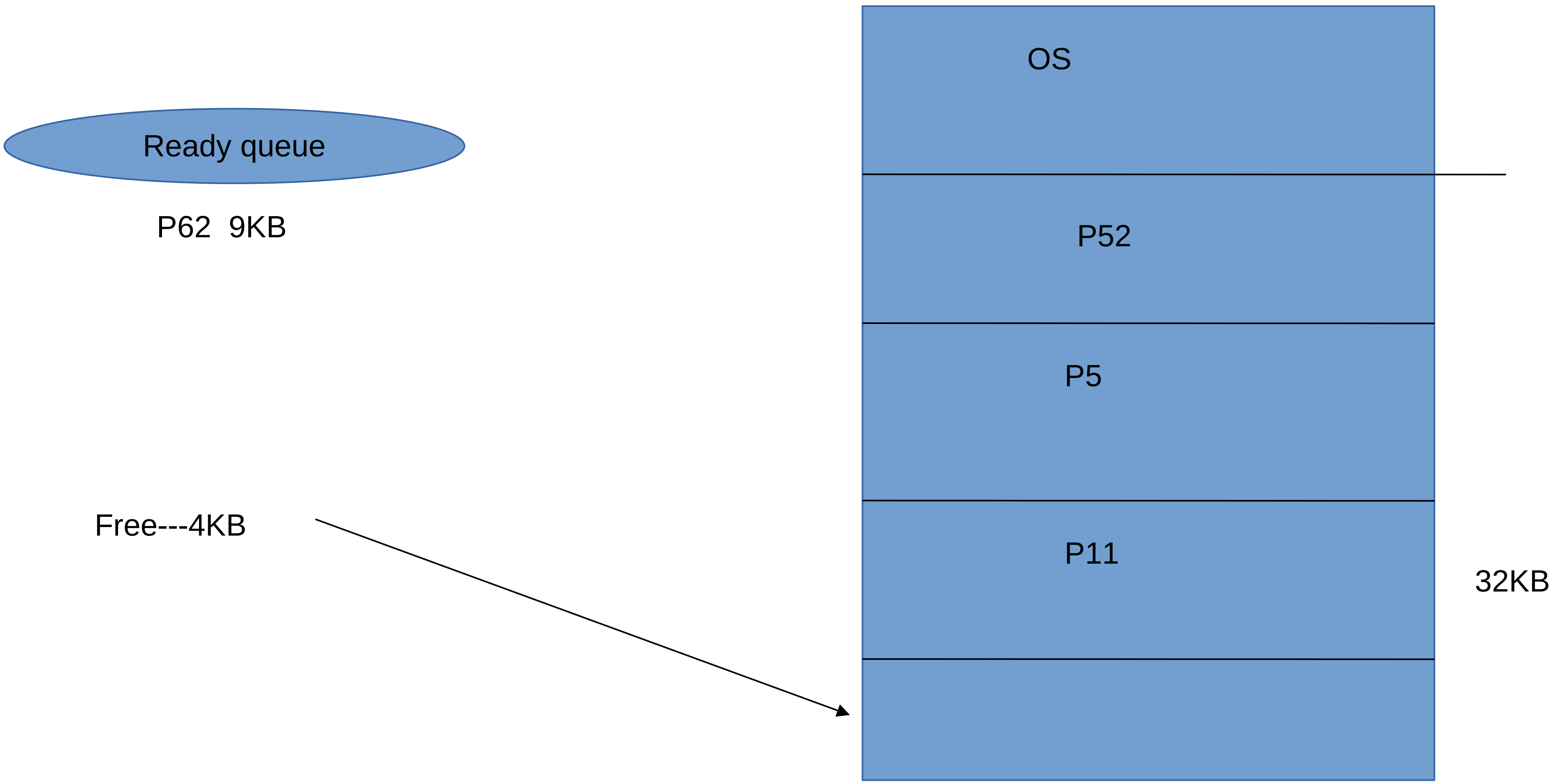



Ready queue
P11 10KB
P52 8KB
P62 9KB
Free---6KB
8 KB
OS
P52
P5
P8
32KB


Ready queue
P11 10KB
P62 9KB
Free---6KB
P8 finishes execution
OS
P52
P5
P8
32KB

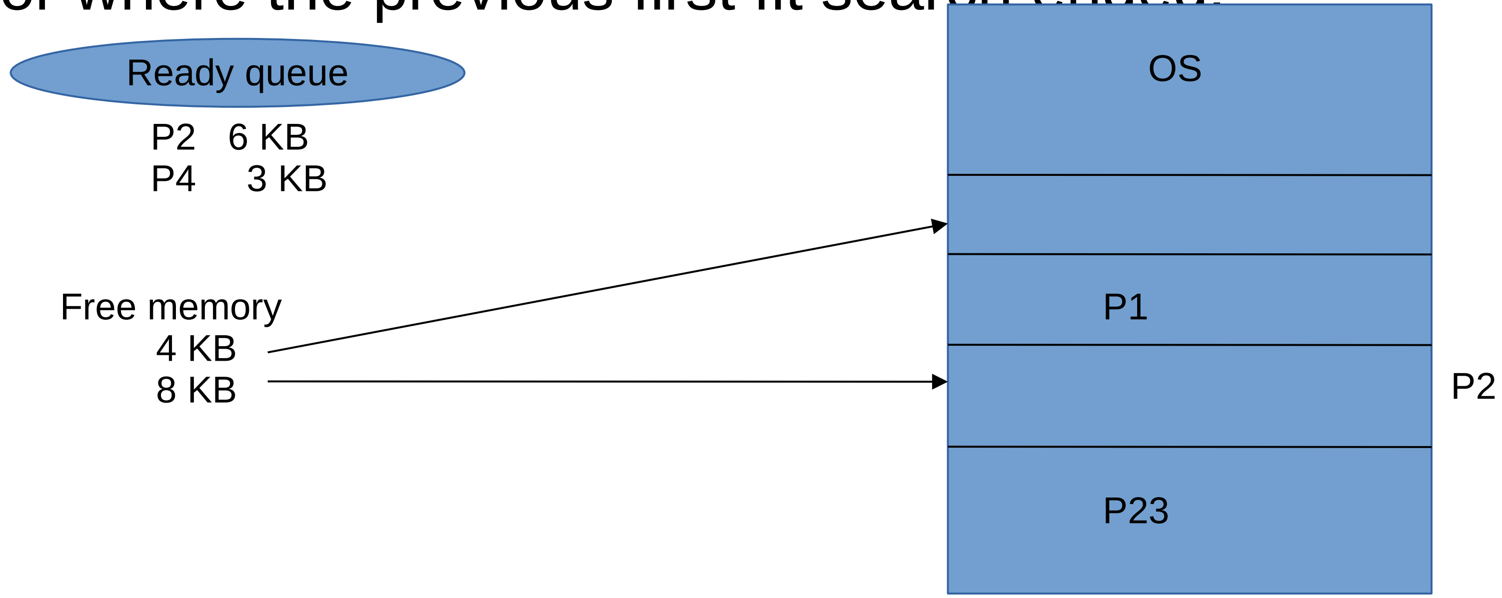

Ready queue
P11 10KB
P62 9KB
Free---14KB
OS
P52
P5
32KB


Ready queue
P62 9KB
Free---4KB
OS
P52
P5
P11
32KB


Ready queue
P62 9KB
Free---4KB
OS
P52
P5
P11
32KB

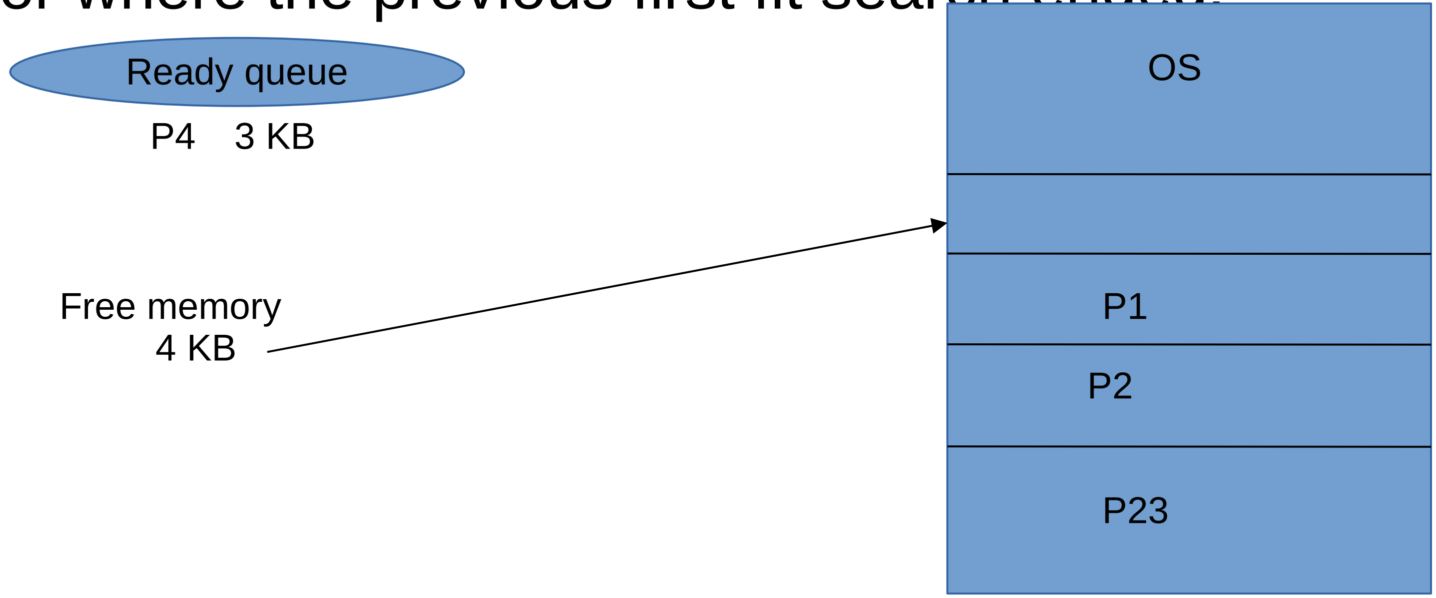
Selection of a hole to hold a process

First fit: allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or where the previous first fit search ended.



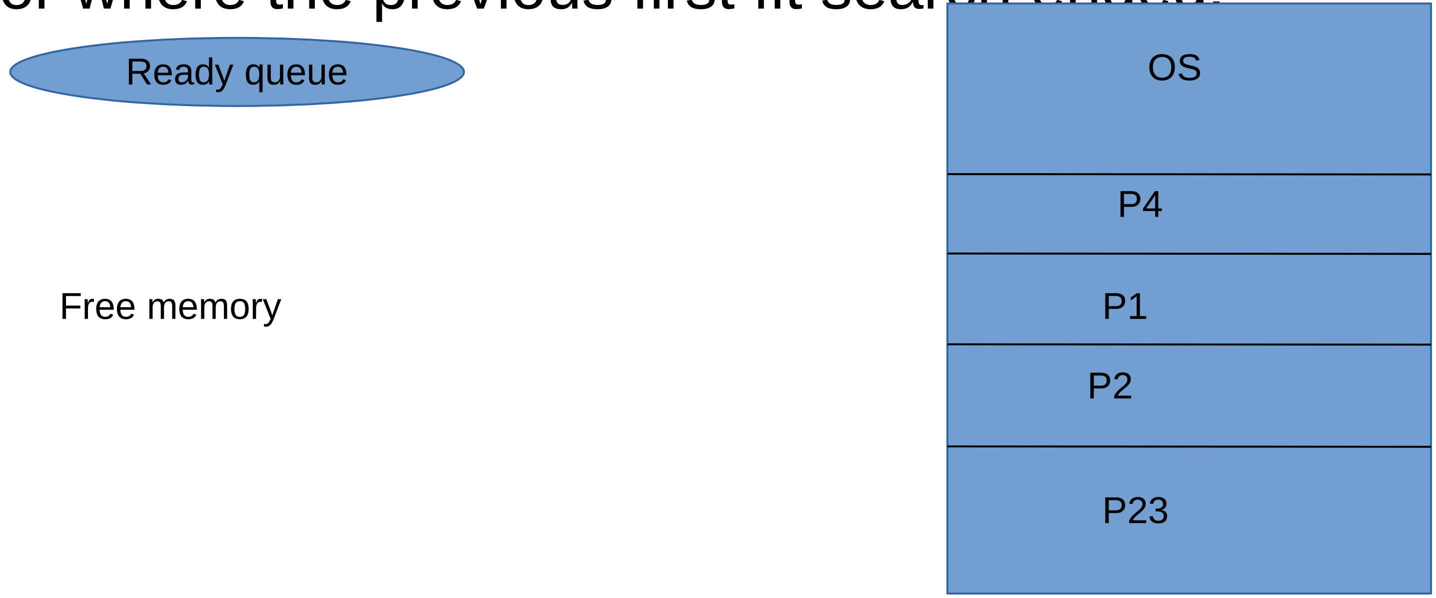
Selection of a hole to hold a process

First fit: allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or where the previous first fit search ended.



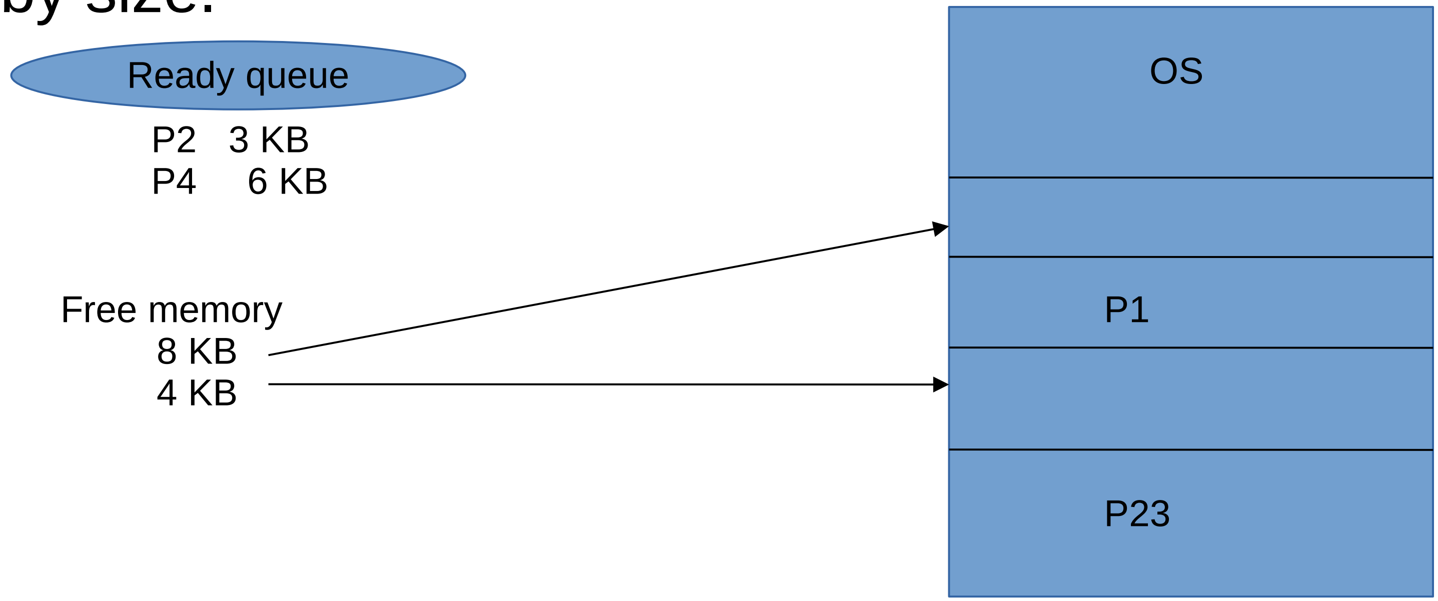
Selection of a hole to hold a process

First fit: allocate the first hole that is big enough. Searching can start at the beginning of the set of holes or where the previous first fit search ended.



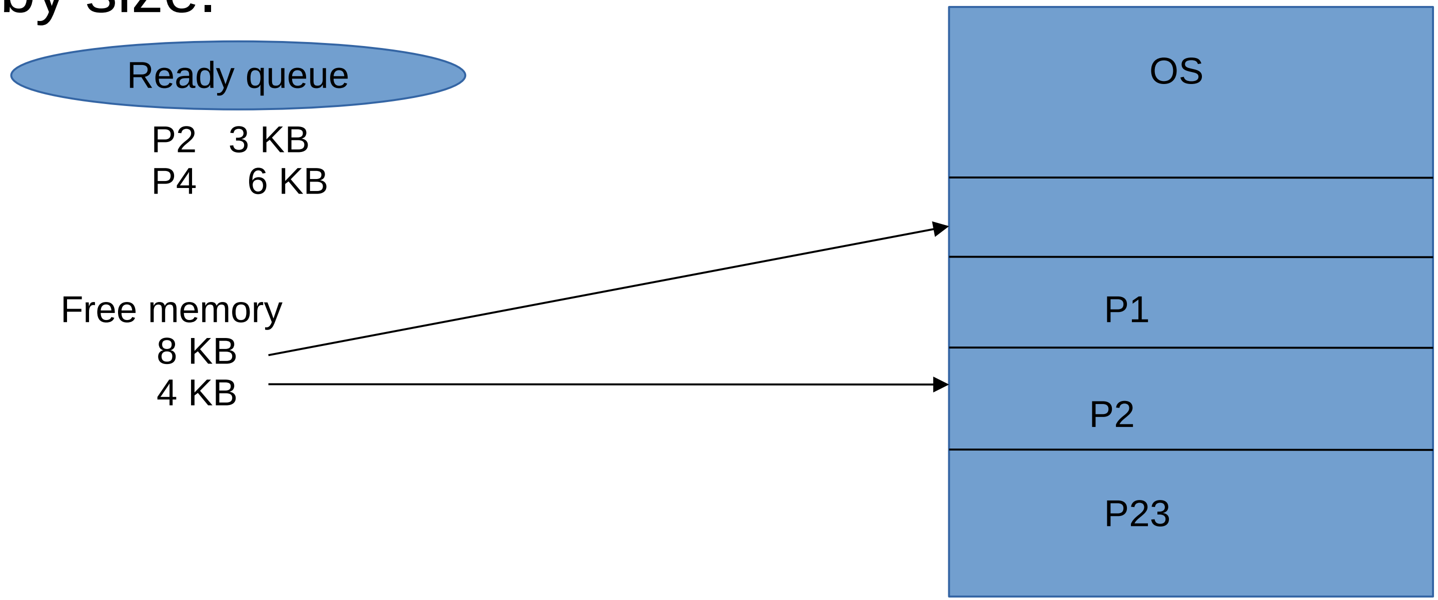
Selection of a hole to hold a process

best fit: allocate the smallest hole that is big enough. We must Search the entire list, unless the list is kept ordered by size.



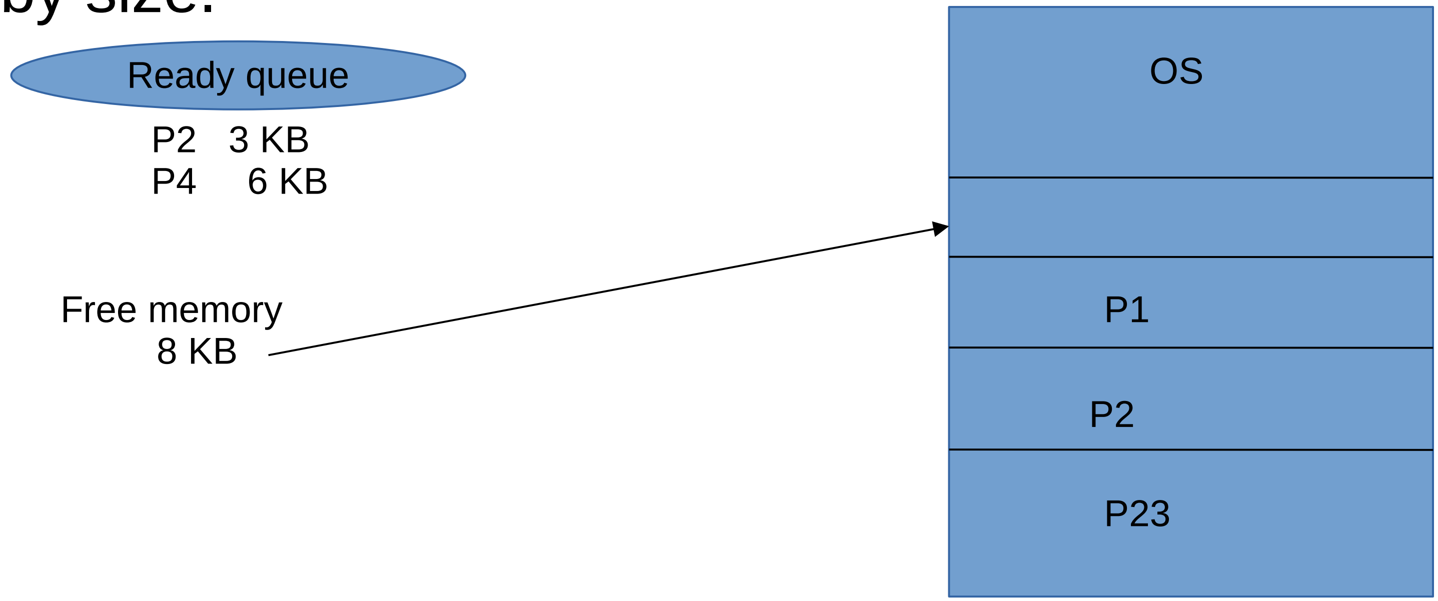
Selection of a hole to hold a process

best fit: allocate the smallest hole that is big enough. We must Search the entire list, unless the list is kept ordered by size.



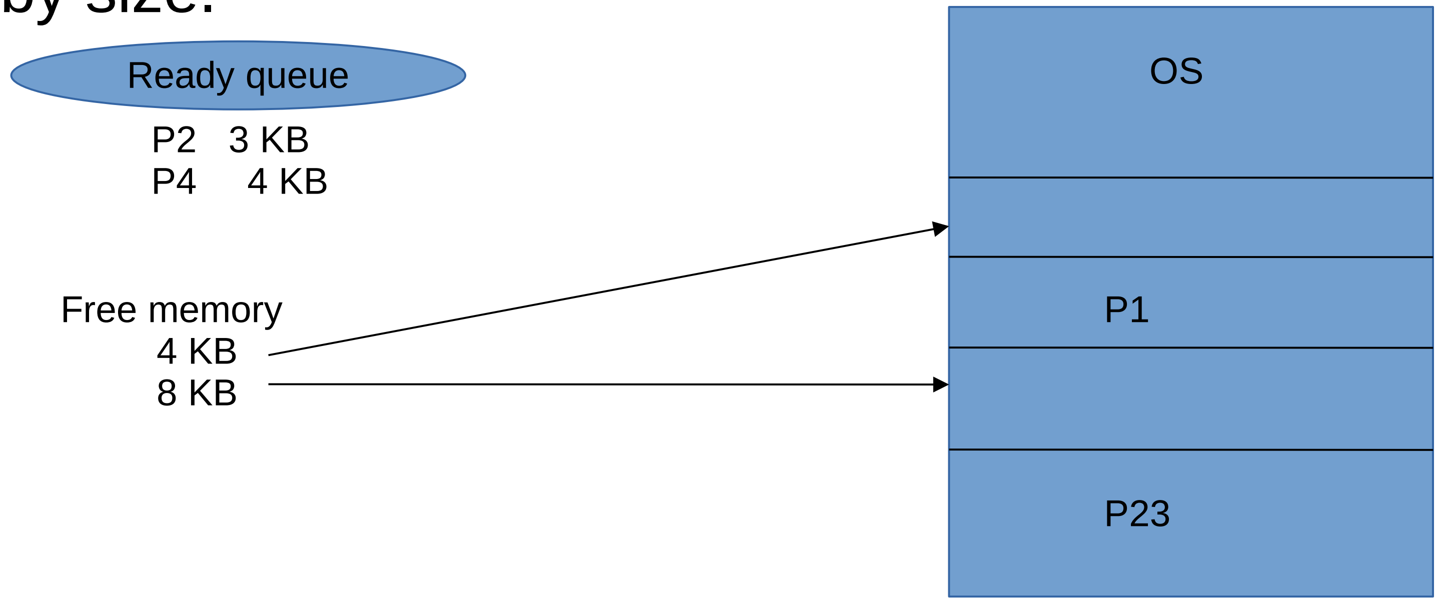
Selection of a hole to hold a process

best fit: allocate the smallest hole that is big enough. We must Search the entire list, unless the list is kept ordered by size.



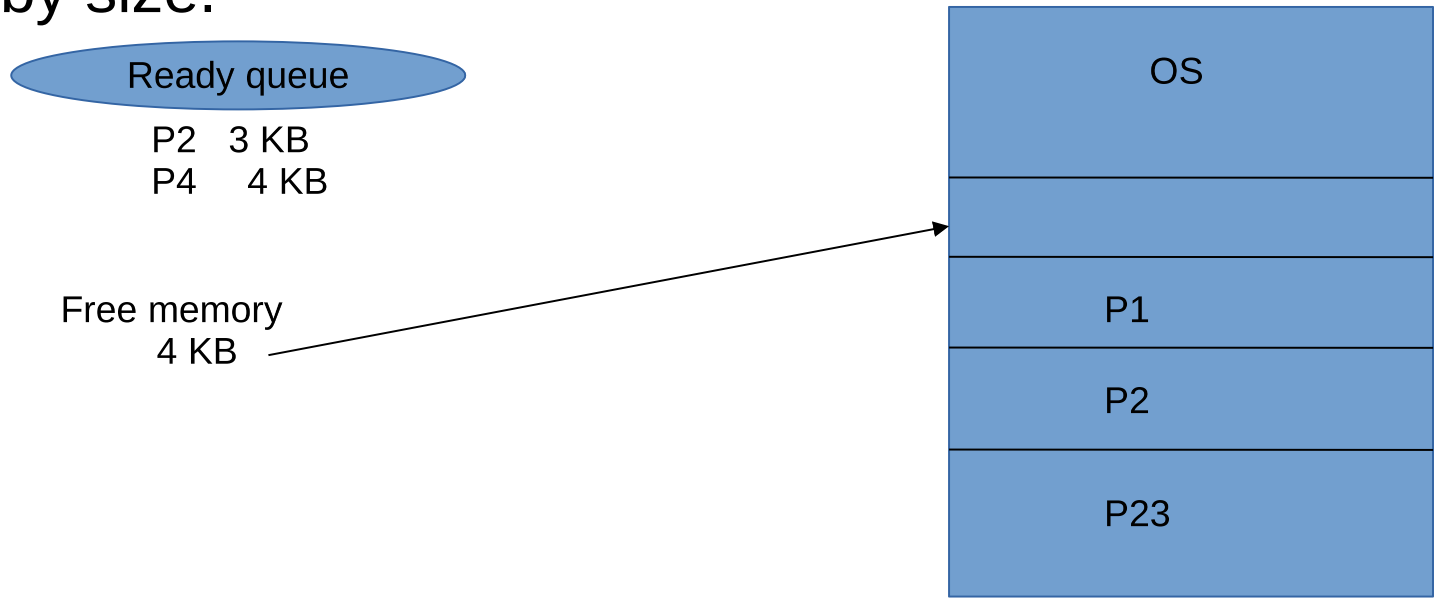
Selection of a hole to hold a process

Worst fit: allocate the largest hole that is big enough. We must Search the entire list, unless the list is kept ordered by size.



Selection of a hole to hold a process

Worst fit: allocate the largest hole that is big enough. We must Search the entire list, unless the list is kept ordered by size.



Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing both time and storage utilization.

Neither first fit nor best fit is clearly better in terms of storage utilization but first fit is generally faster.

External Fragmentation

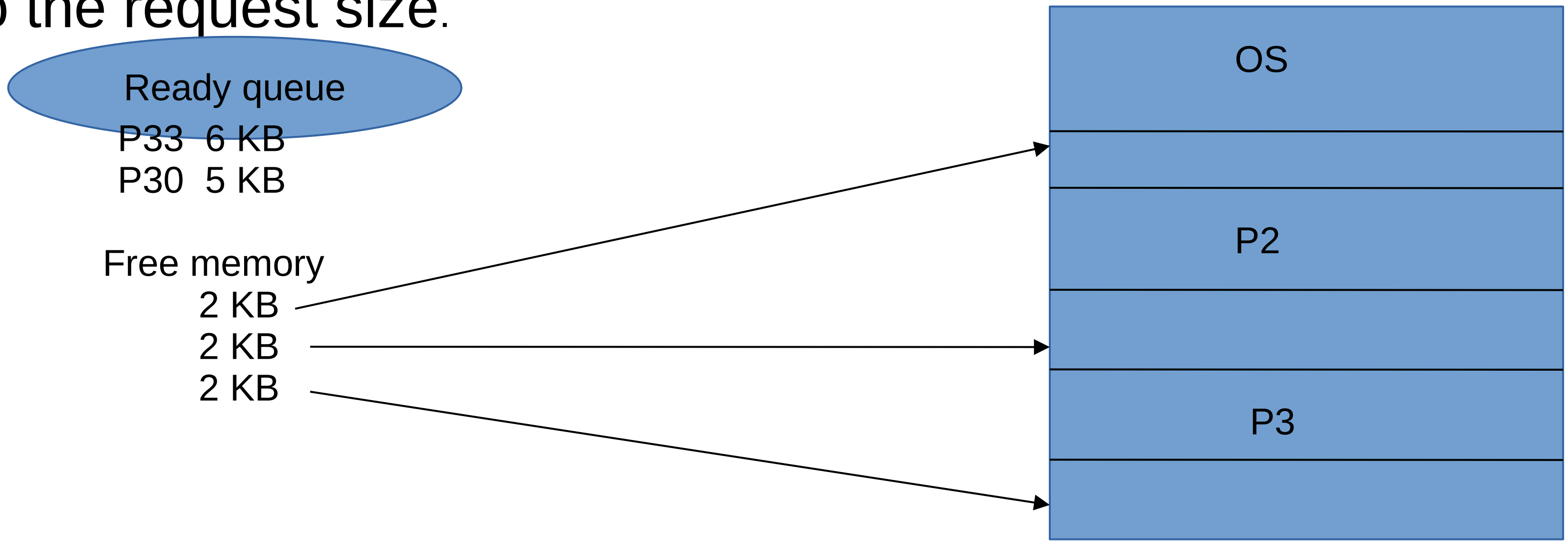
External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous.

In other words there are a number of small holes none of which is large enough to satisfy a request, but the sum of their sizes is greater than or equal to the request size.

External Fragmentation

External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous.

In other words there are a number of small holes none of which is large enough to satisfy a request, but the sum of their sizes is greater than or equal to the request size.



Solution?

Compaction

The goal is to shuffle the memory contents to place all the free memory together in one large block.

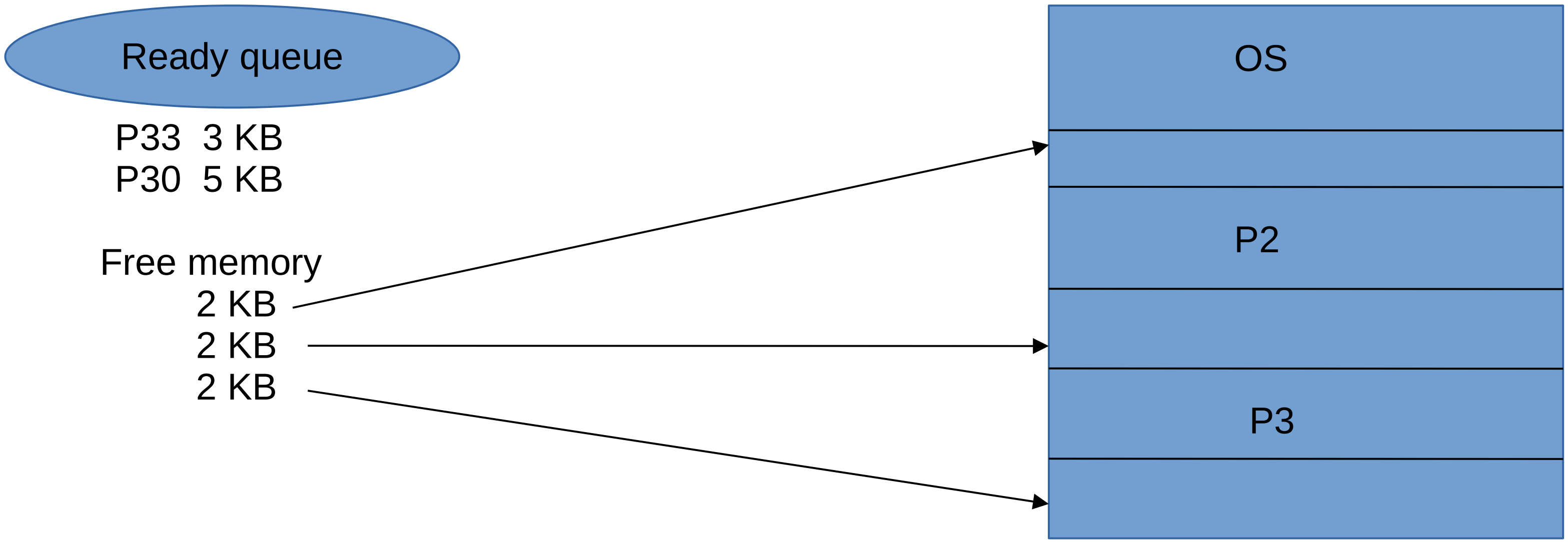
For a relocatable process to be able to execute in its new location, all internal addresses must be relocated.

If the relocation is static (Binding at compile time) compaction can not be done.

If relocation is dynamic (Binding at load time) compaction can be done.

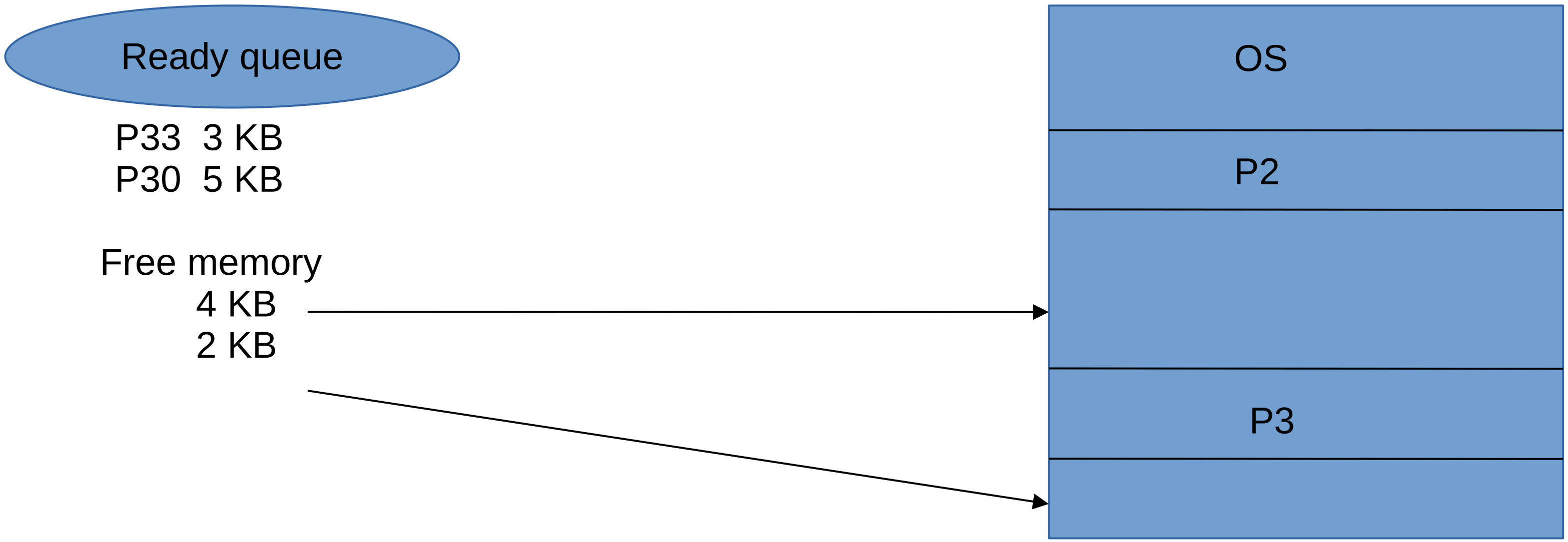
Compaction Technique

A. simply move all processes towards one end of memory all holes move in the other direction producing one large hole of available memory.



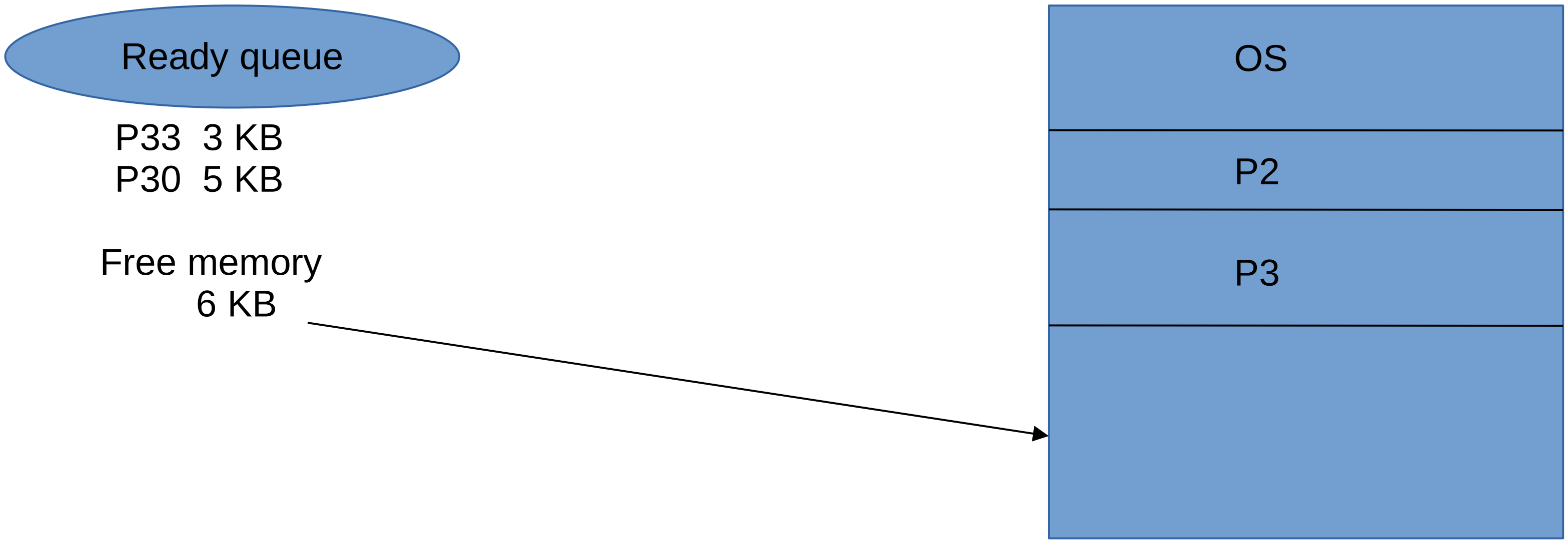
Compaction Technique

A. simply move all processes towards one end of memory all holes move in the other direction producing one large hole of available memory.



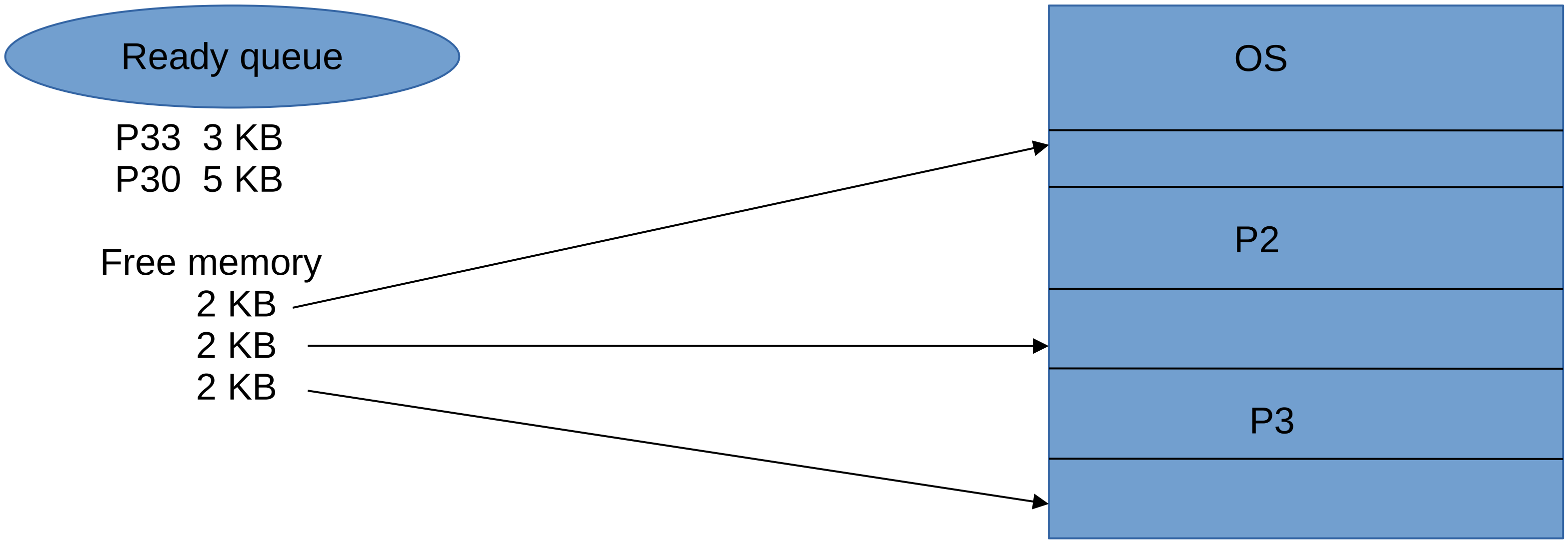
Compaction Technique

A. simply move all processes towards one end of memory all holes move in the other direction producing one large hole of available memory.



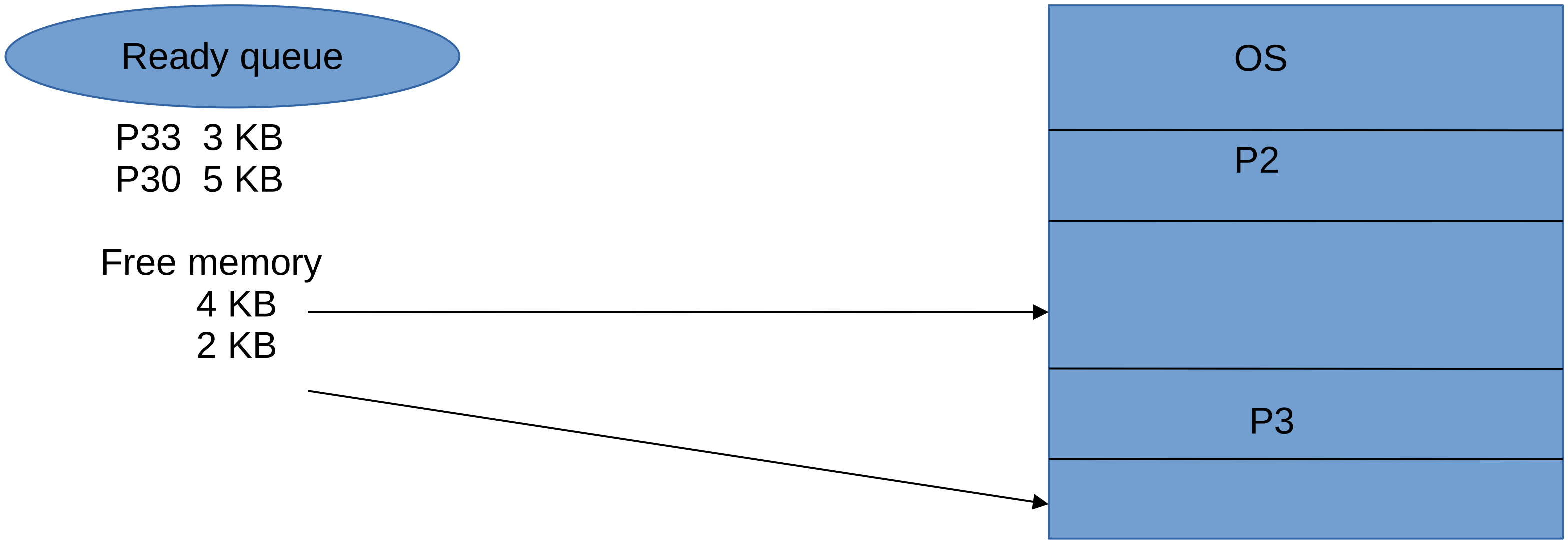
Compaction Technique

B. Create a large hole big enough anywhere to satisfy the request.



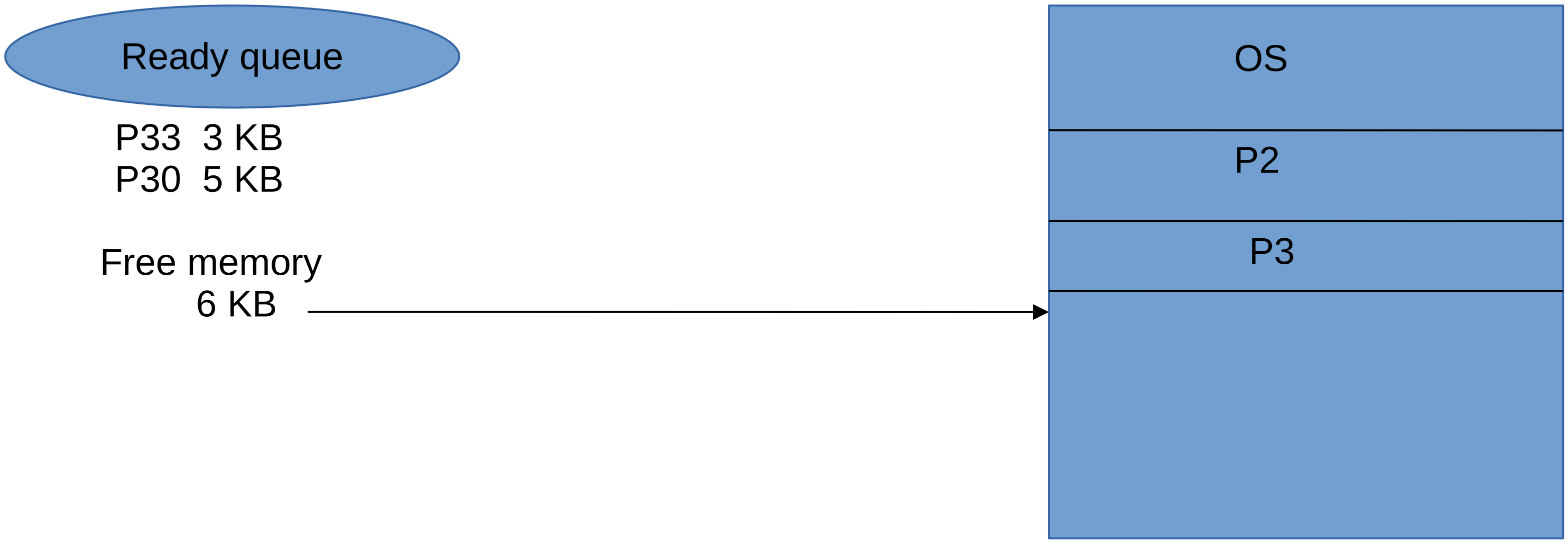
Compaction Technique

B. Create a large hole big enough anywhere to satisfy the request.



Compaction Technique

B. Create a large hole big enough anywhere to satisfy the request.



Internal Fragmentation

Consider a hole of 2002 bytes and let us say this is the only available at the moment.

Suppose the next process requests 2000 bytes.

If we allocate exactly the requested block, we are left with a hole of 2 bytes.

The overhead to keep track of this hole will be substantially larger than the hole itself.

The general idea is to allocate this hole as part of the larger request.

There fore the allocated memory is slightly larger than the request. So a little amount of memory is wasted.

So internal fragmentation exist.

Compaction is really tough ……. solution?

Paging

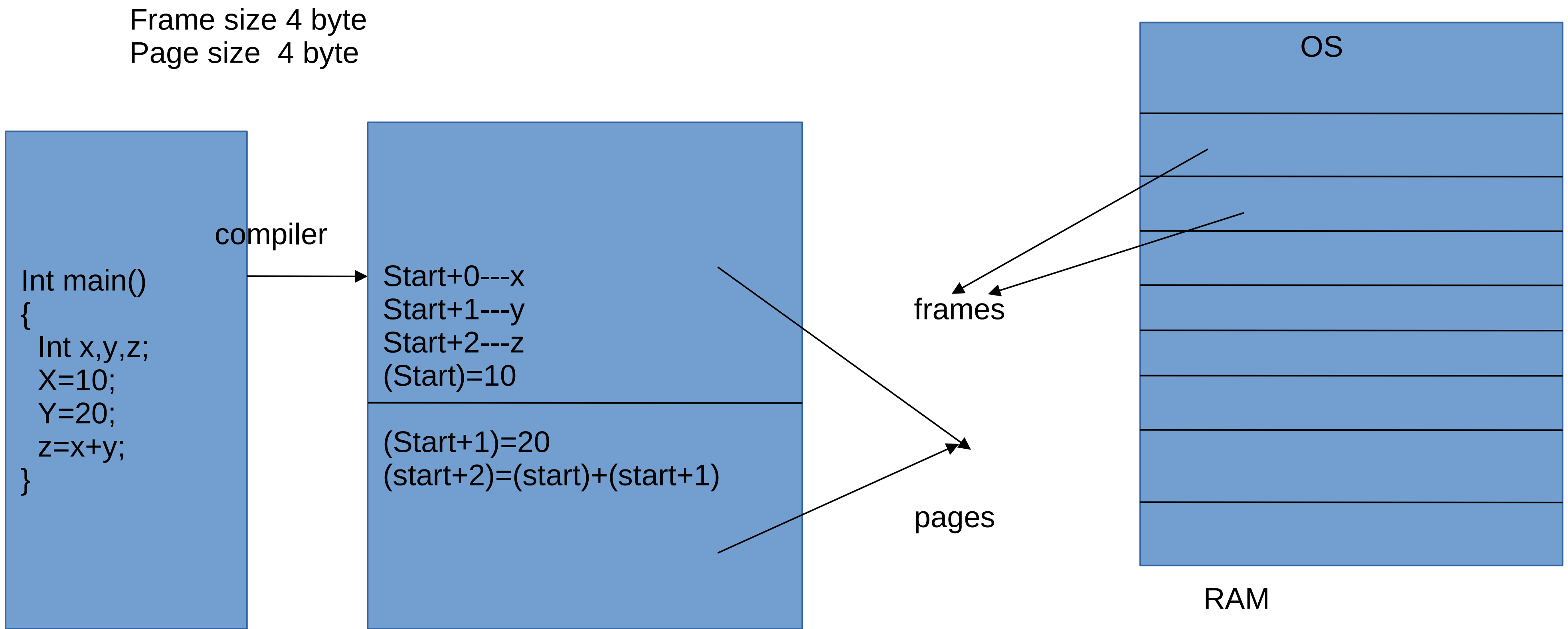
An address generated by the CPU is commonly refereed to as a logical address, whereas an address seen by the memory unit(the one loaded in to the MAR) is commonly referred to as a physical address.

The set of all logical addresses generated by a program is referred to as a logical address space.

Whereas the set of all physical addresses corresponding to these logical addresses is referred to as a physical address space.

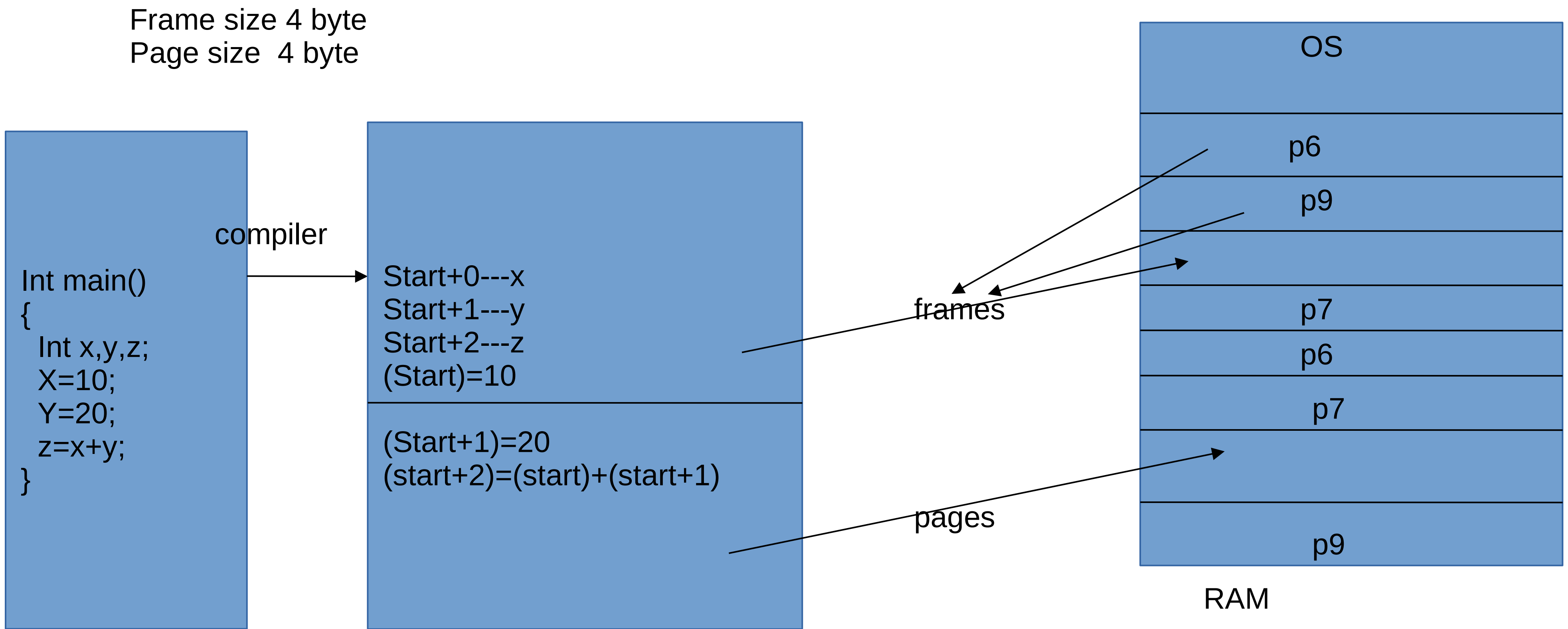
Physical memory is broken into fixed size blocks called **frames** .

Logical memory is also broken into blocks of the same size called **pages.**



When a process is to be executed, its pages (which are in the backing storage) are loaded into any available memory frames.

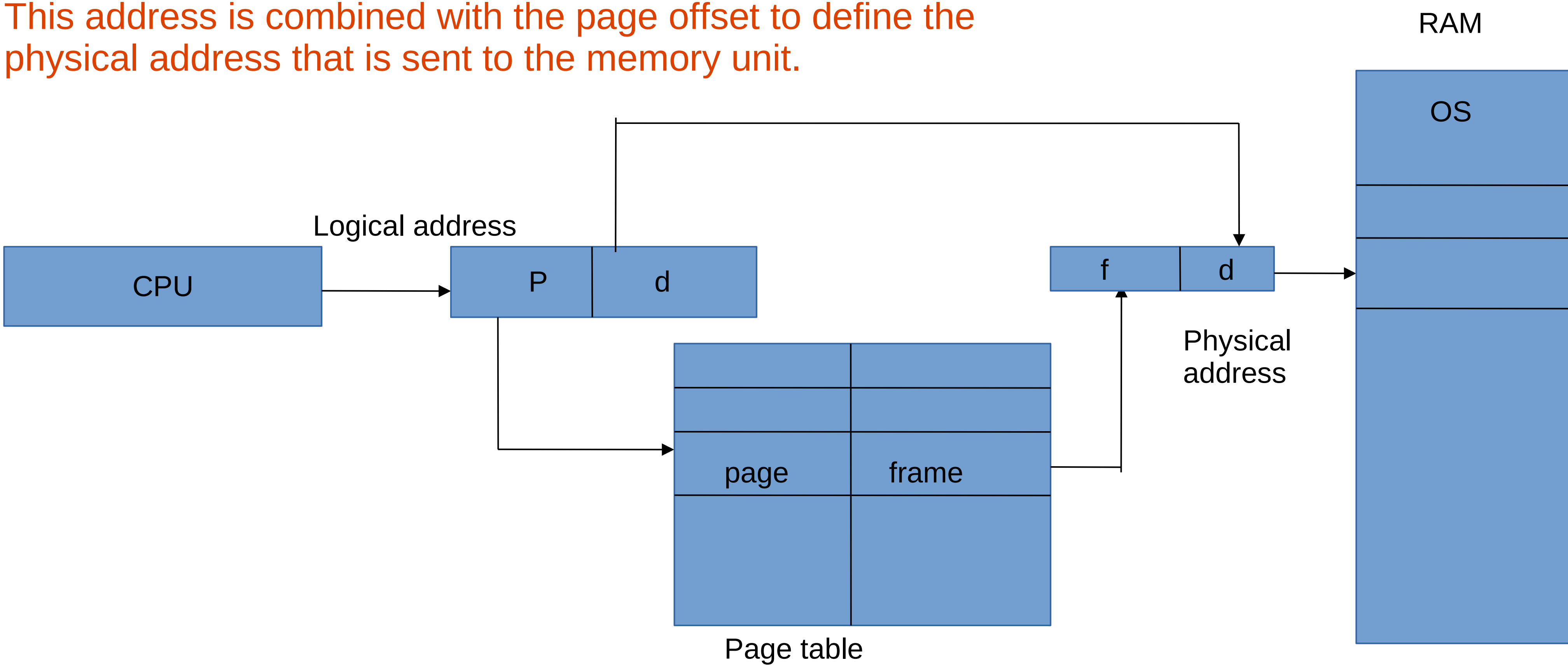
Therefore the pages of a process may not be contiguous.



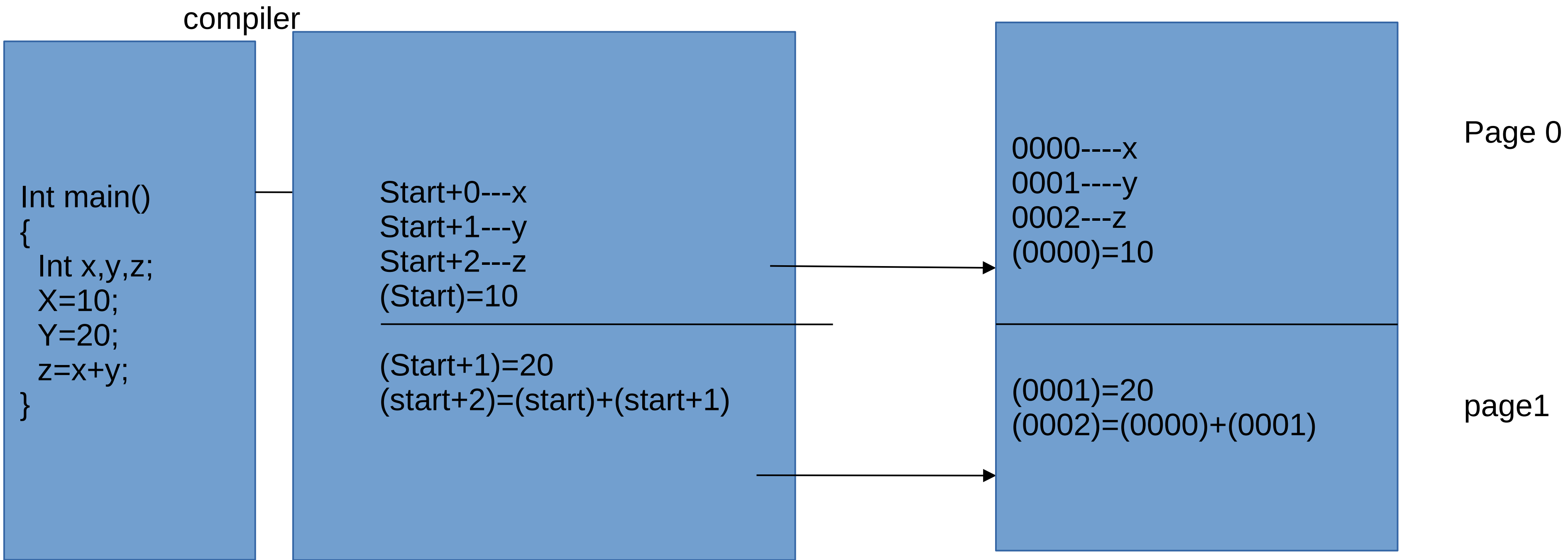
Every address generated by the CPU is divided into 2 parts, a page number(p) and a page offset(d).

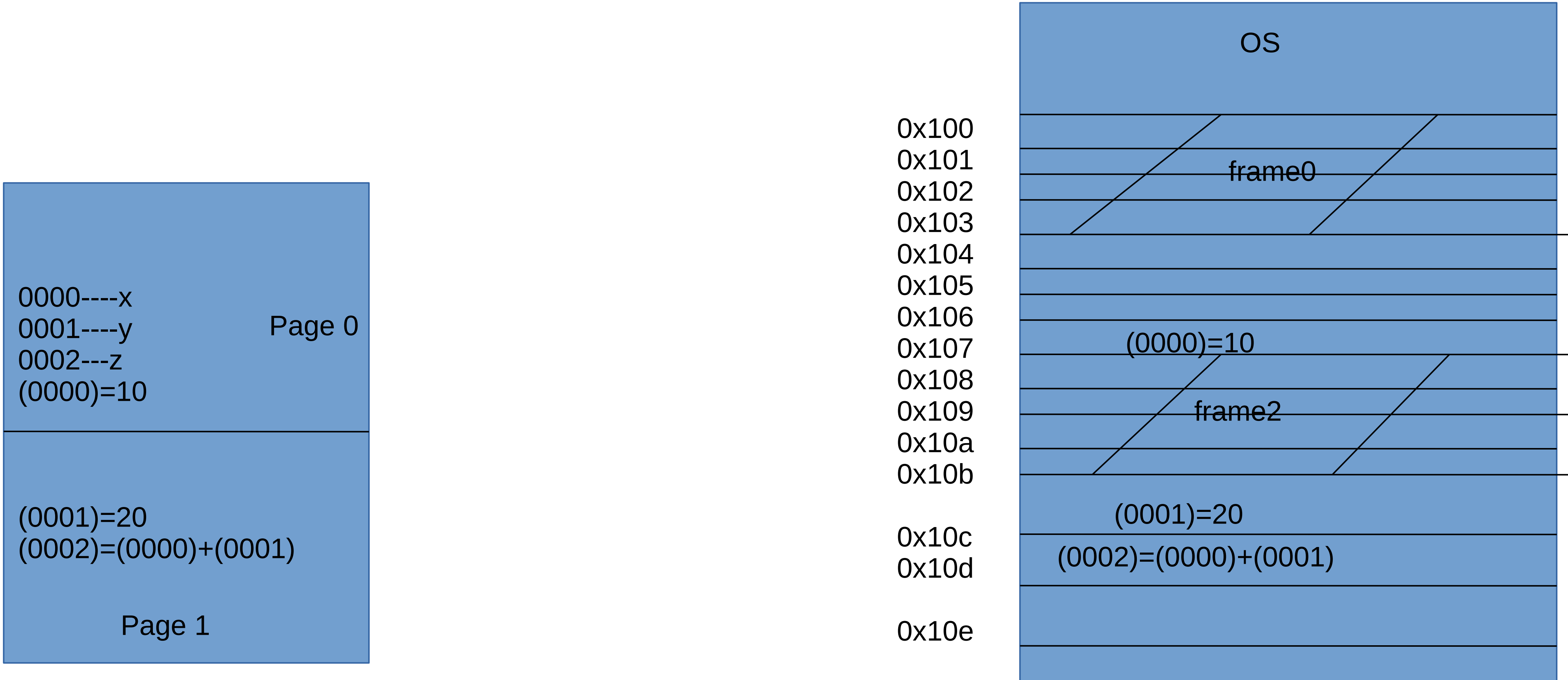
The page number p is used as index to a page table which keeps the starting address of each page in the physical memory(frame).

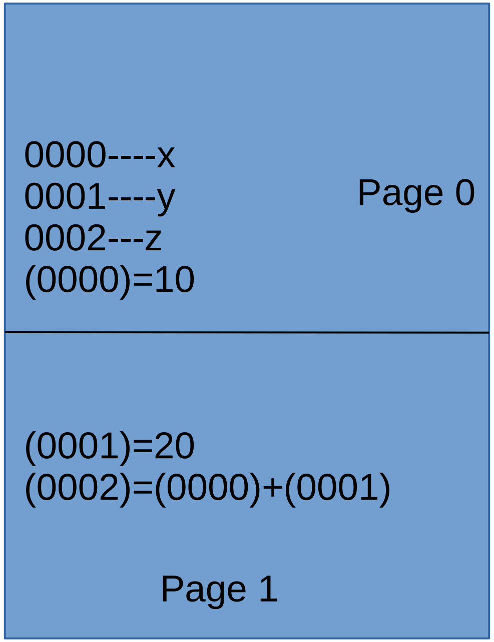
This address is combined with the page offset to define the physical address that is sent to the memory unit.



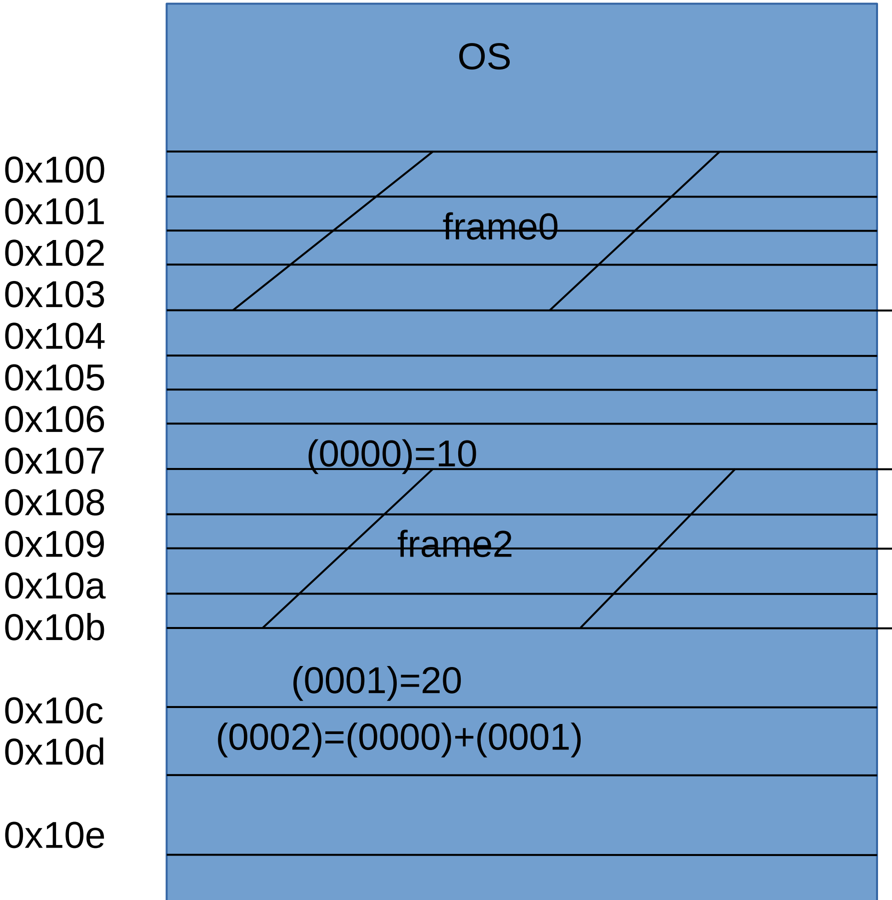
Frame size 4 byte
Page size 4 byte

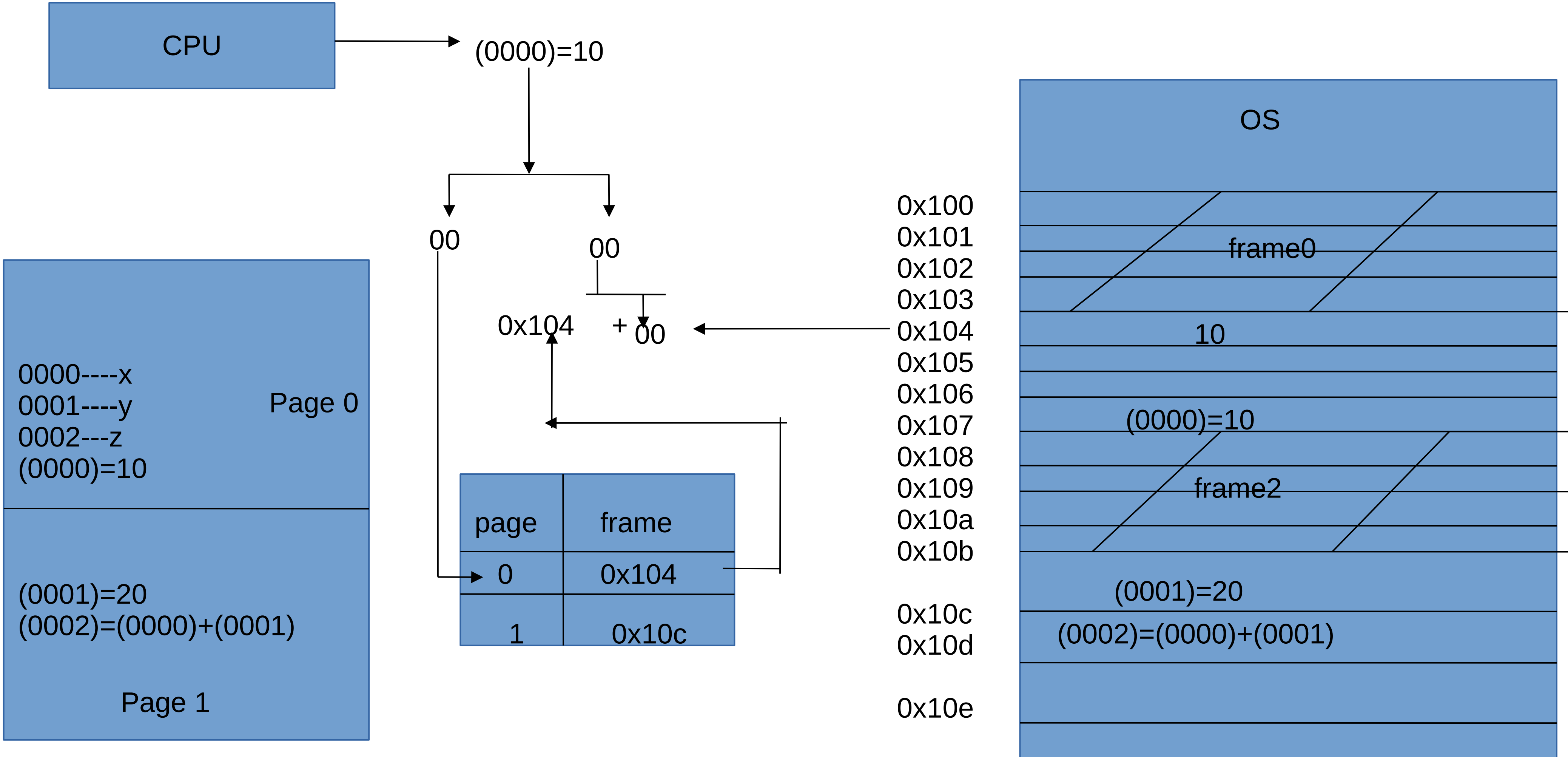



0000----x
0001----y
0002---z
(0000)=10
Page 0
(0001)=20
(0002)=(0000)+(0001)
Page 1
OS
0x100
0x101
0x102
0x103
0x104
0x105
0x106
0x107
0x108
0x109
0x10a
0x10b
0x10c
0x10d
0x10e
frame0
(0000)=10
frame2
(0001)=20
(0002)=(0000)+(0001)



page	frame
0	0x104
1	0x10c




CPU
(0000)=10
00
00
0x104
+ 00
0000----x
0001----y
0002---z
(0000)=10
Page 0
(0001)=20
(0002)=(0000)+(0001)
Page 1
page
frame
0
0x104
1
0x10c
0x100
0x101
0x102
0x103
0x104
0x105
0x106
0x107
0x108
0x109
0x10a
0x10b
0x10c
0x10d
0x10e
OS
frame0
10
(0000)=10
frame2
(0001)=20
(0002)=(0000)+(0001)

How logical address is divided into page number and offset?

The page size which is defined by the hardware is typically a power of 2 varying between 512 bytes and 8192 bytes.

If the size of a logical address space is pow(2,m) and a page size is pow(2,n) addressing units then the high order m-n bits of a logical address designate the page number and low order bits designate the page offset.

There fore the logical address is as follows:

Page number	Page offset
P	d
m-n	n

Paging eliminates external fragmentation altogether but there may be a little internal fragmentation

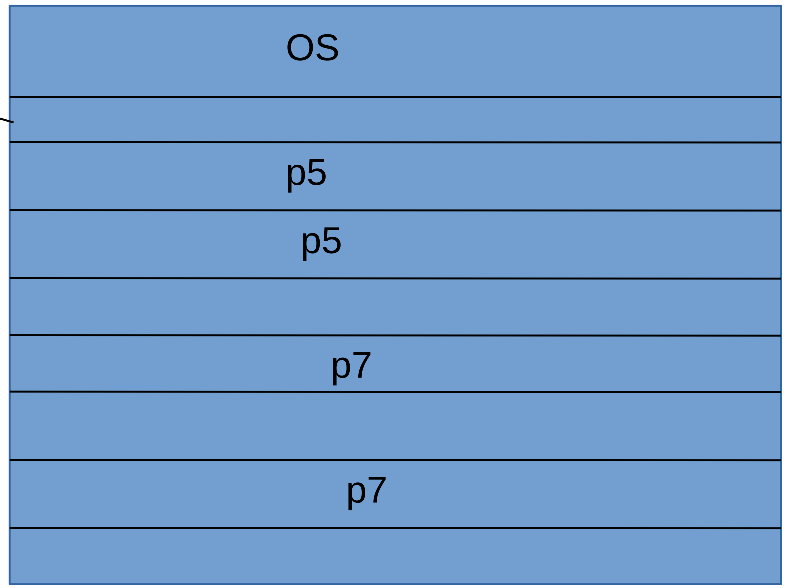
1. The logical address produced by the user program are translated into physical addresses.
2. This translation is controlled by the OS.
3. When a process arrives to be executed its size expressed in pages is examined.
4. Each user page need one frame.
5. If the required number of frames are available they are allocated to this process.
6. The first page of process is loaded into one of the allocated frames and the frame number is put into the page table for this process.
7. The next page is loaded into another frame and the frame number is similarly put into the page table and so on.

To keep track of the frames which are free and which are not the OS maintains a frame table, which has an entry for each of the physical frames.

The entry for a frame tells whether this frame is allocated or free and if it is allocated to which page of which process or processes.

Frame table

0	free
1	p5
2	p5
3	free
4	p7
5	free
6	p7
7	free



The OS maintains a copy of the page table for each process just as it maintain a copy for the instruction counter and instruction register.

Consider page size = frame size = 4 bytes

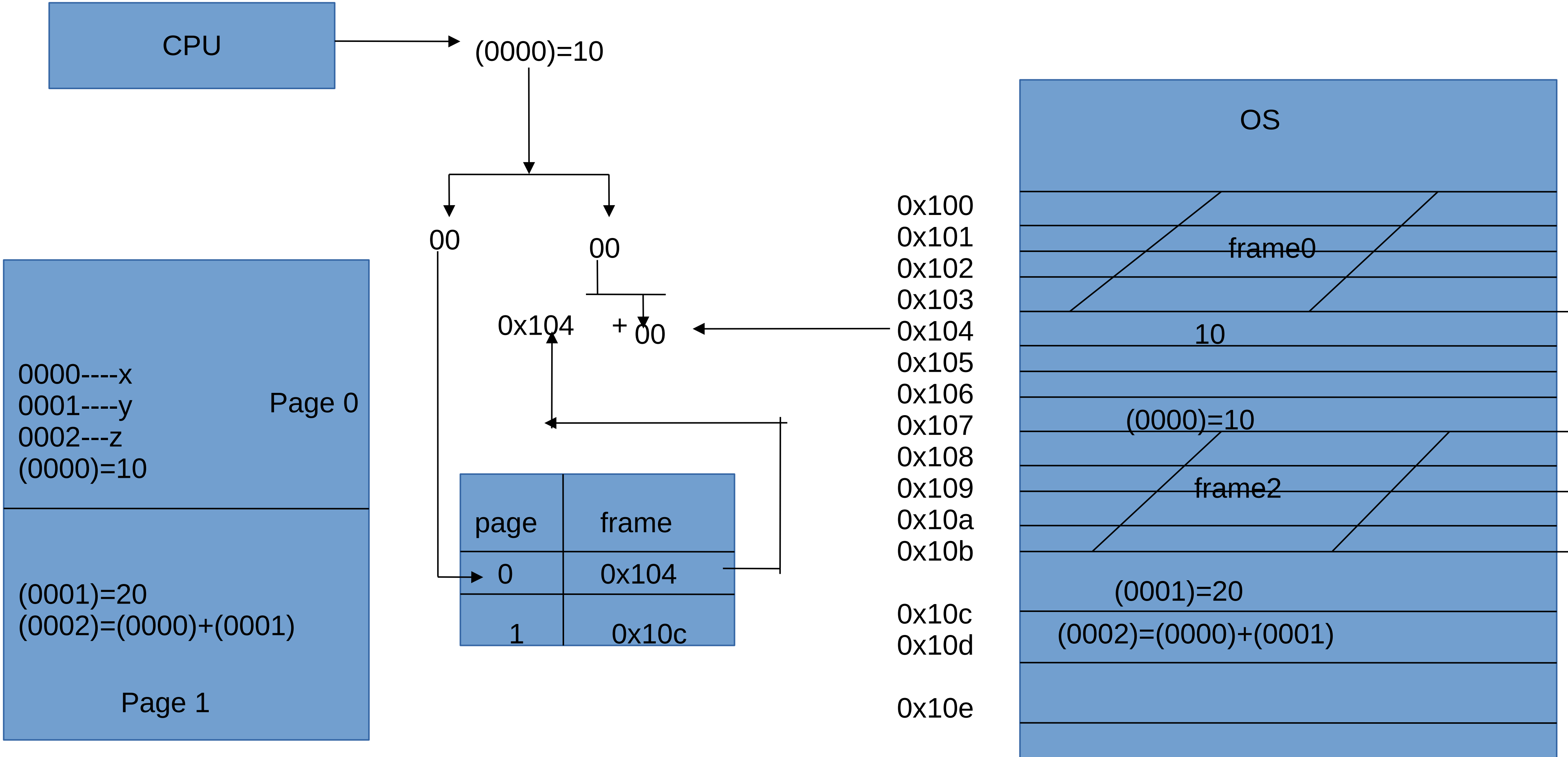
Suppose the logical address space for a process is 20 bytes.

How many pages are there in this process?

Draw the page table for this process.....

Solution ?
Multi level paging

One way of getting around this problem is to use a two level paging scheme in which the page table itself is also paged.


CPU
(0000)=10
00
00
0x104
+ 00
0000----x
0001----y
0002---z
(0000)=10
Page 0
(0001)=20
(0002)=(0000)+(0001)
Page 1
page
frame
0
0x104
1
0x10c
OS
0x100
0x101
0x102
0x103
0x104
0x105
0x106
0x107
0x108
0x109
0x10a
0x10b
0x10c
0x10d
0x10e
frame0
10
(0000)=10
frame2
(0001)=20
(0002)=(0000)+(0001)

Consider page size = frame size = 4 bytes

Suppose the logical address space for a process is 20 bytes.

How many pages are there in this process?

Draw the page table for this process.....

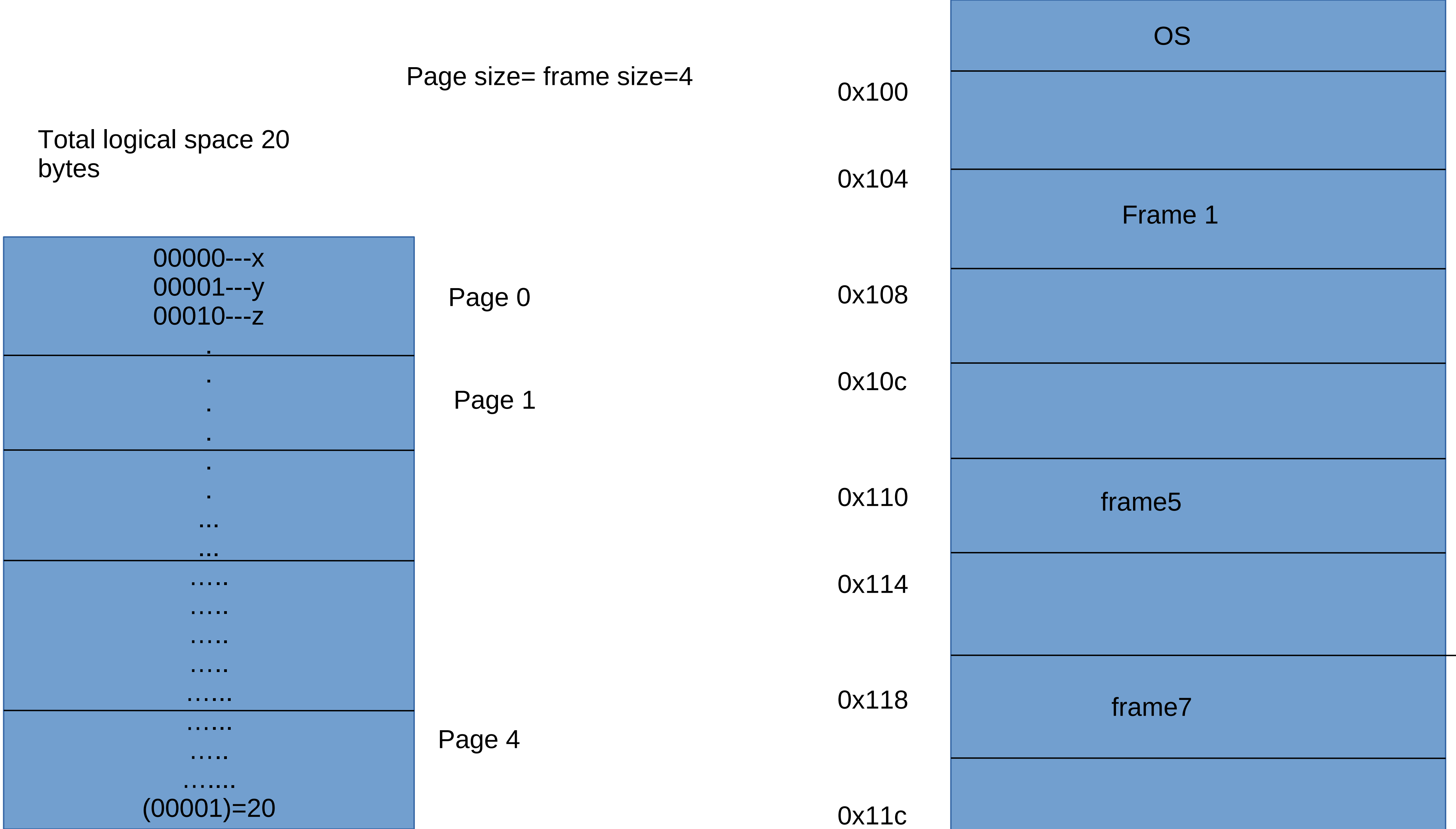
Solution ?

Multi level paging

One way of getting around this problem is to use a two level paging scheme in which the page table itself is also paged.

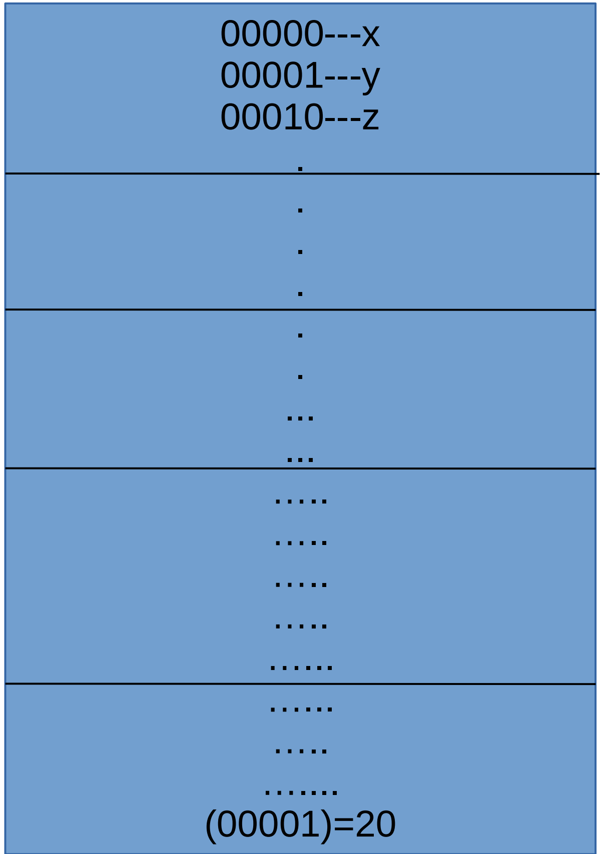
Page size= frame size=4

Total logical space 20 bytes



Page size= frame size=4

Total logical space 20 bytes

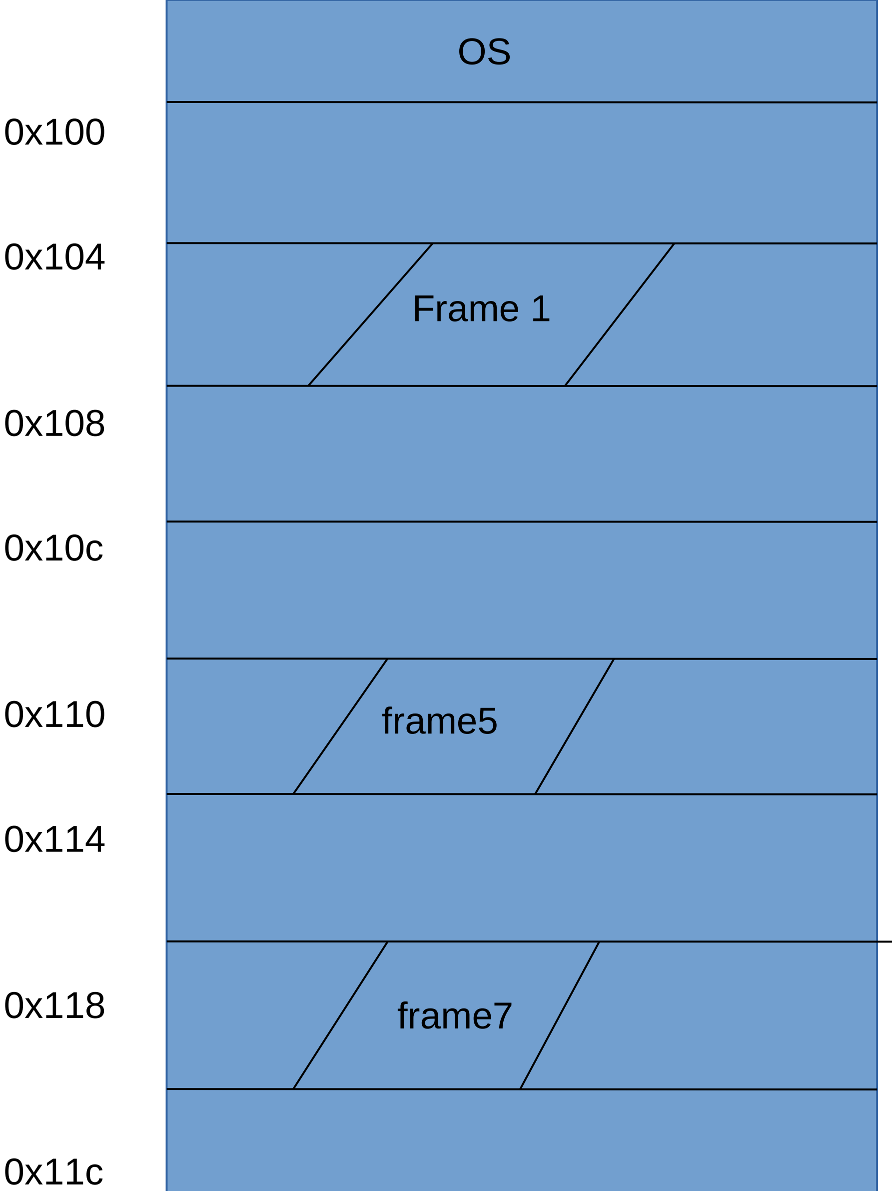


Page 0

Page 1

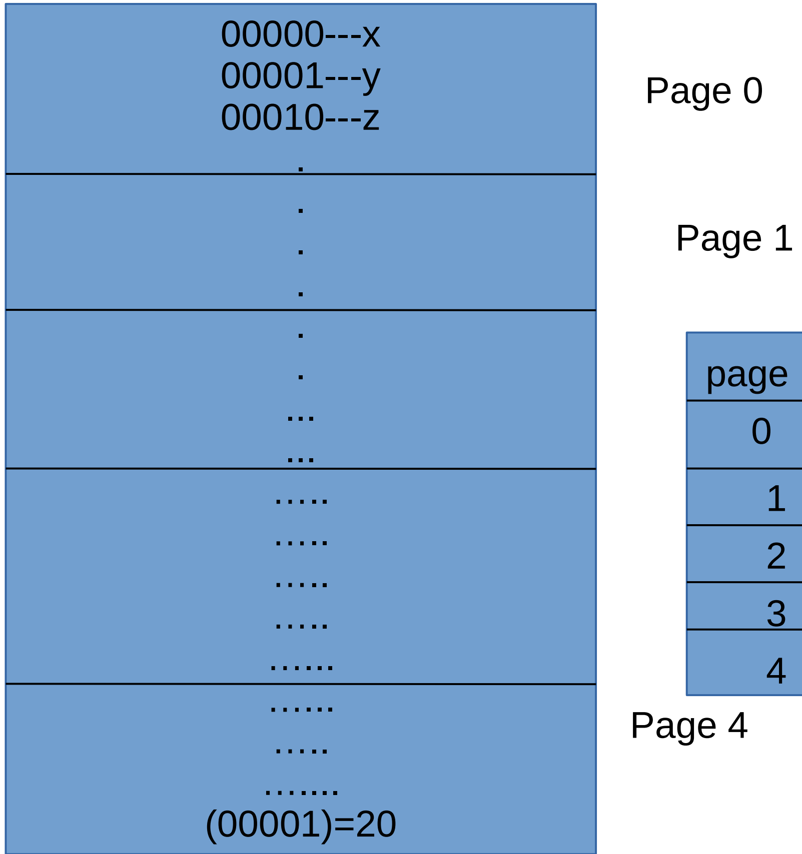
page	frame
0	0x100
1	0x108
2	0x10c
3	0x114
4	0x11c

Page 4

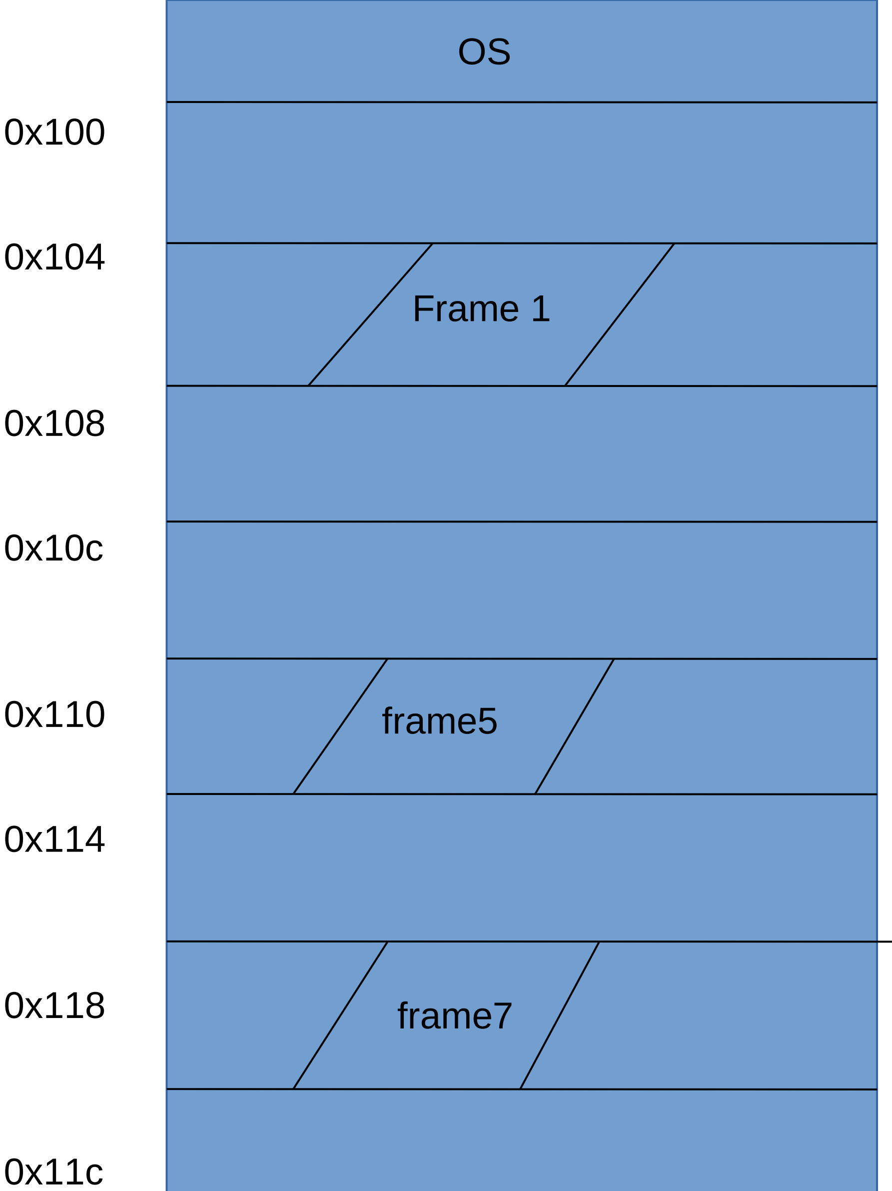


Total logical space 20 bytes m=5

Page size= frame size=4 n=2

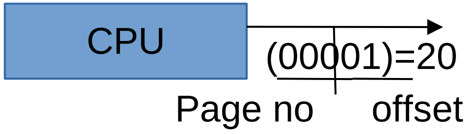


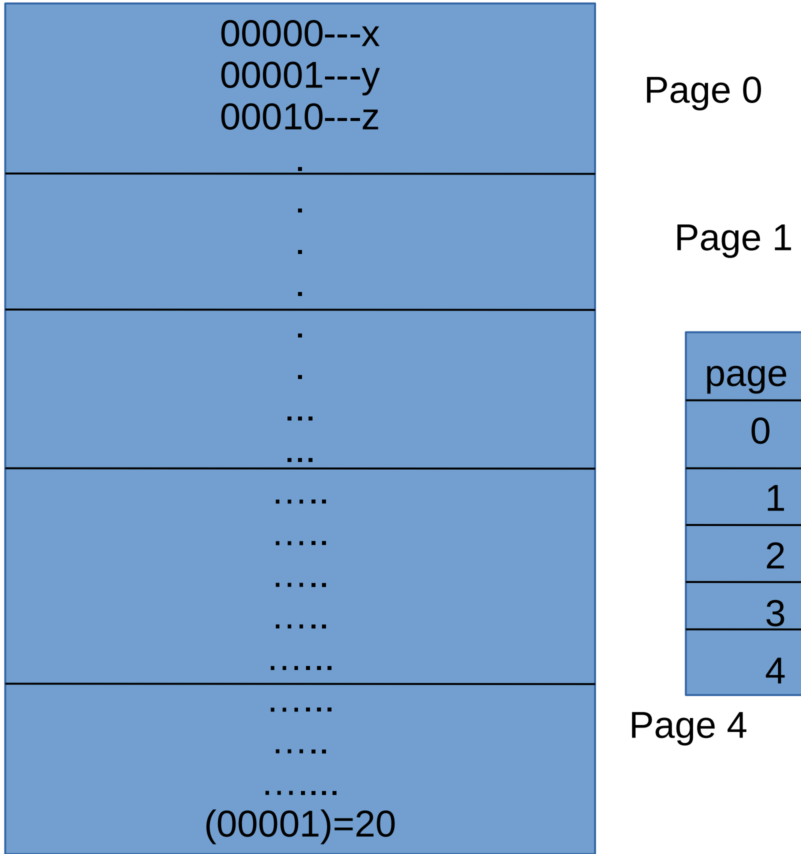
page	frame
0	0x100
1	0x108
2	0x10c
3	0x114
4	0x11c



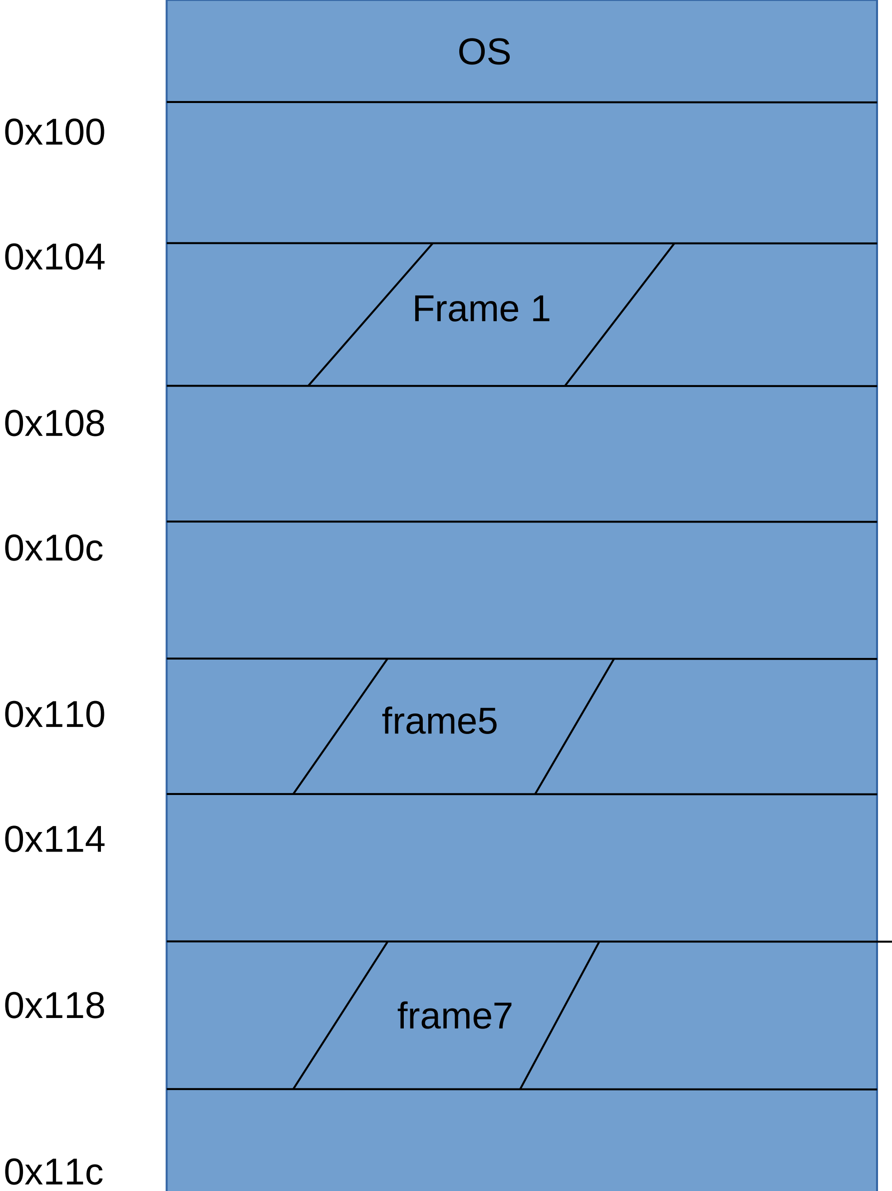
Total logical space 20 bytes m=5

Page size= frame size=4
n=2



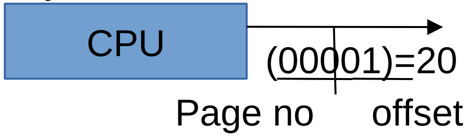


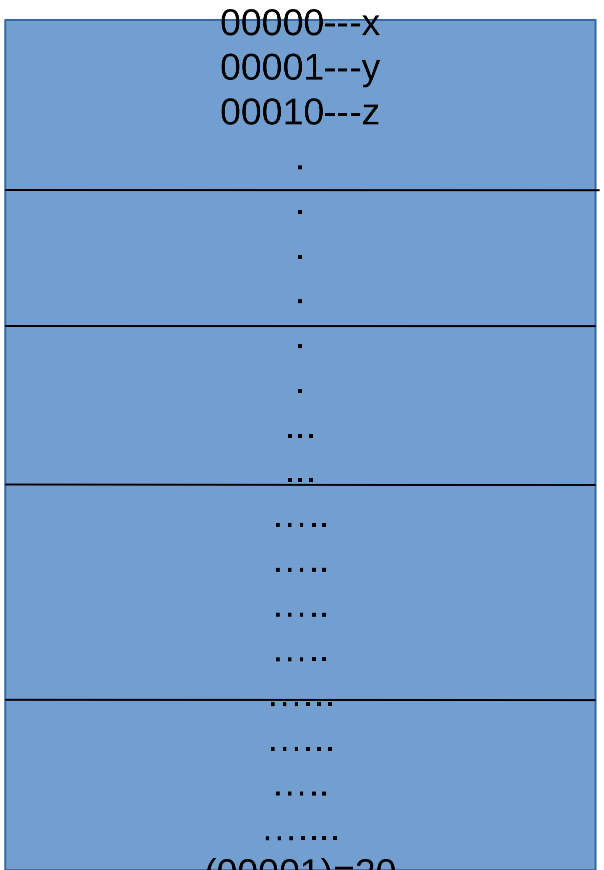
page	frame
0	0x100
1	0x108
2	0x10c
3	0x114
4	0x11c



Total logical space 20 bytes m=5

Page size= frame size=4
n=2



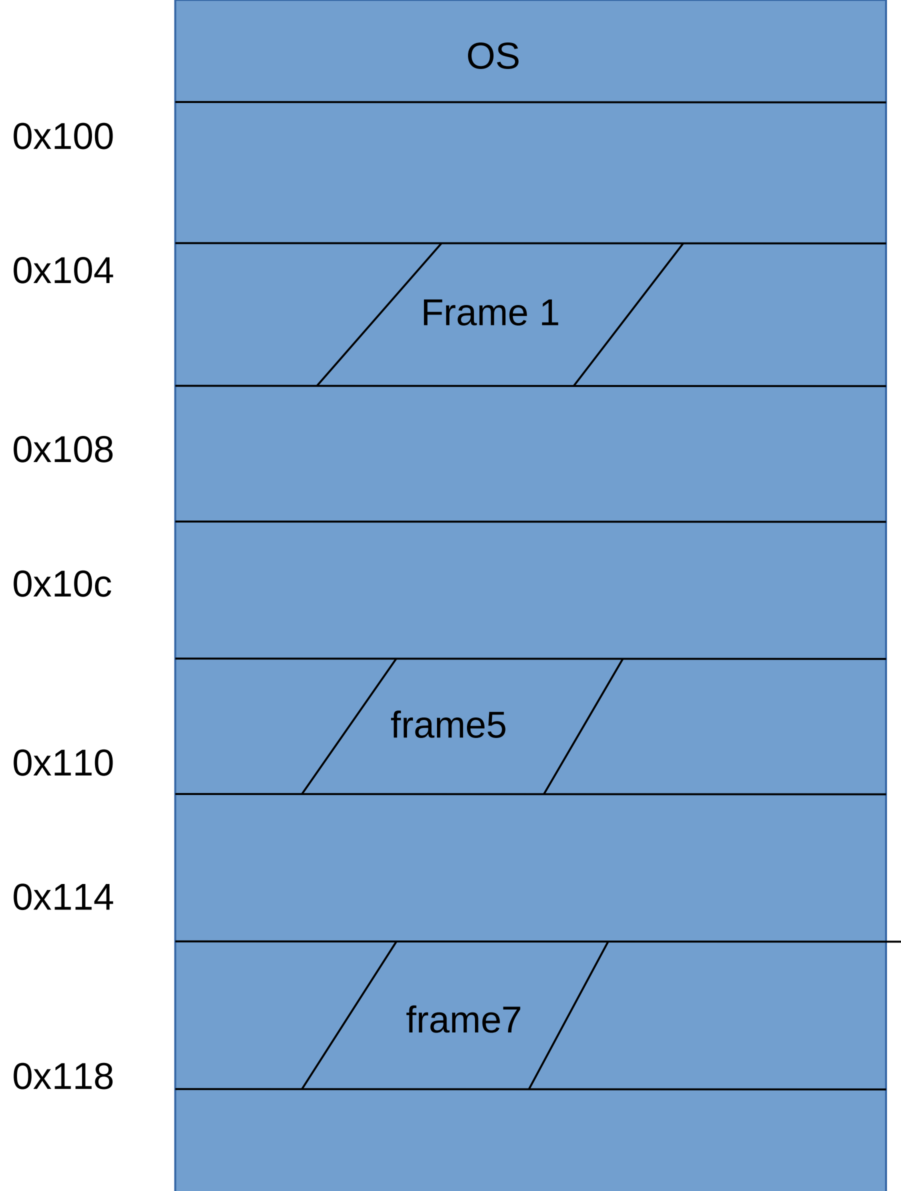


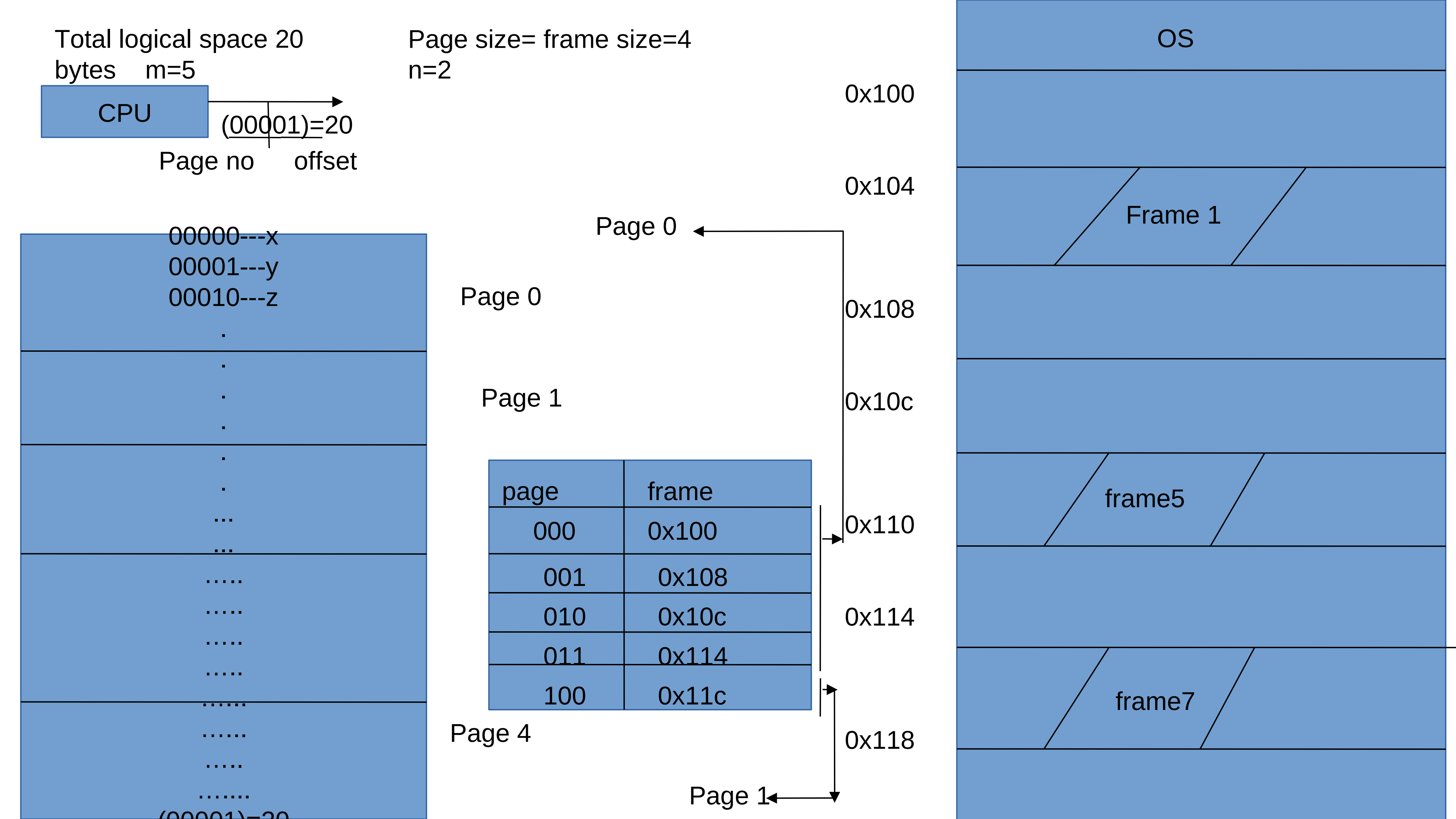
Page 0

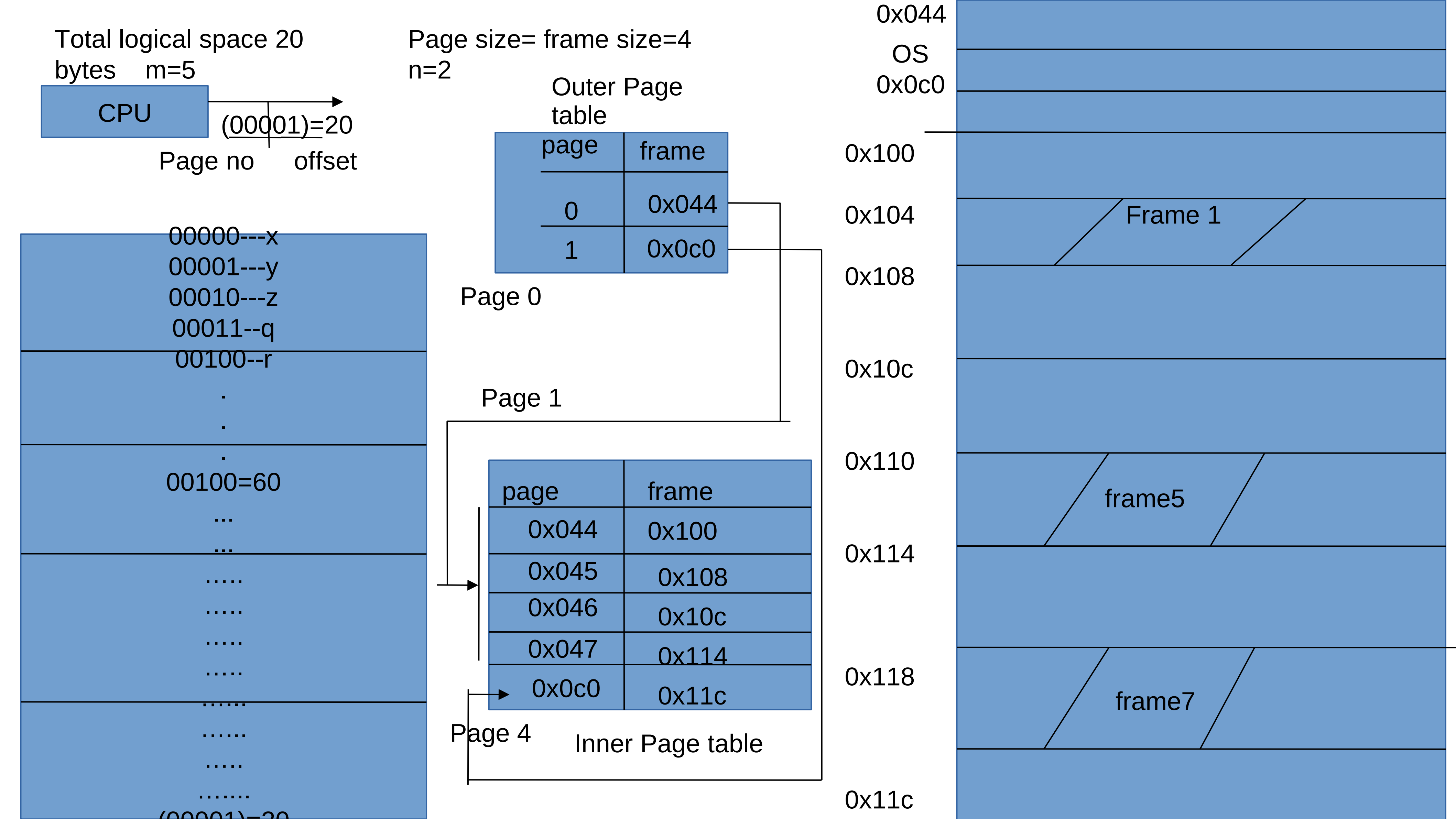
Page 1

page	frame
00	0x100
01	0x108
10	0x10c
11	0x114
100	0x11c

Page 4




Total logical space 20 bytes m=5
Page size= frame size=4 n=2
CPU
(00001)=20
Page no
offset
00000---x
00001---y
00010---z
Page 0
Page 1
Page 4
Page 0
Page 1
page | frame
000 | 0x100
001 | 0x108
010 | 0x10c
011 | 0x114
100 | 0x11c
(00001)=20
0x100
0x104
0x108
0x10c
0x110
0x114
0x118
OS
Frame 1
frame5
frame7



Total logical space 20 bytes m=5
Page size= frame size=4 n=2
CPU
(00001)=20
Page no
offset
Outer Page table
page
frame
0
0x044
1
0x0c0
Page 0
00000---x
00001---y
00010---z
00011--q
00100--r
00100=60
Page 1
page
frame
0x044
0x100
0x045
0x108
0x046
0x10c
0x047
0x114
0x0c0
0x11c
Page 4
Inner Page table
(00001)=20
0x044
OS
0x0c0
0x100
0x104
Frame 1
0x108
0x10c
0x110
frame5
0x114
0x118
frame7
0x11c

Disadvantage ?

Page table for each process consume lots of memory......

Inverted page table

To overcome the disadvantages mention in the previous slides an inverted page table is used.

Inverted page table has one entry for each frame of memory.

Each entry consists of the logical(or virtual) address of the page stored in that memory location. With information about the process that owns it.

Therefore there is only one inverted page table in the system and it has only one entry for each frame of physical memory.

Each logical address in the system is consist of a triplet.

< process id, page number, offset>

each inverted page table entry is a pair<process id,page number>.

When a memory reference occurs part of the logical address consisting of <process id,page number> is presented to memory subsystem .

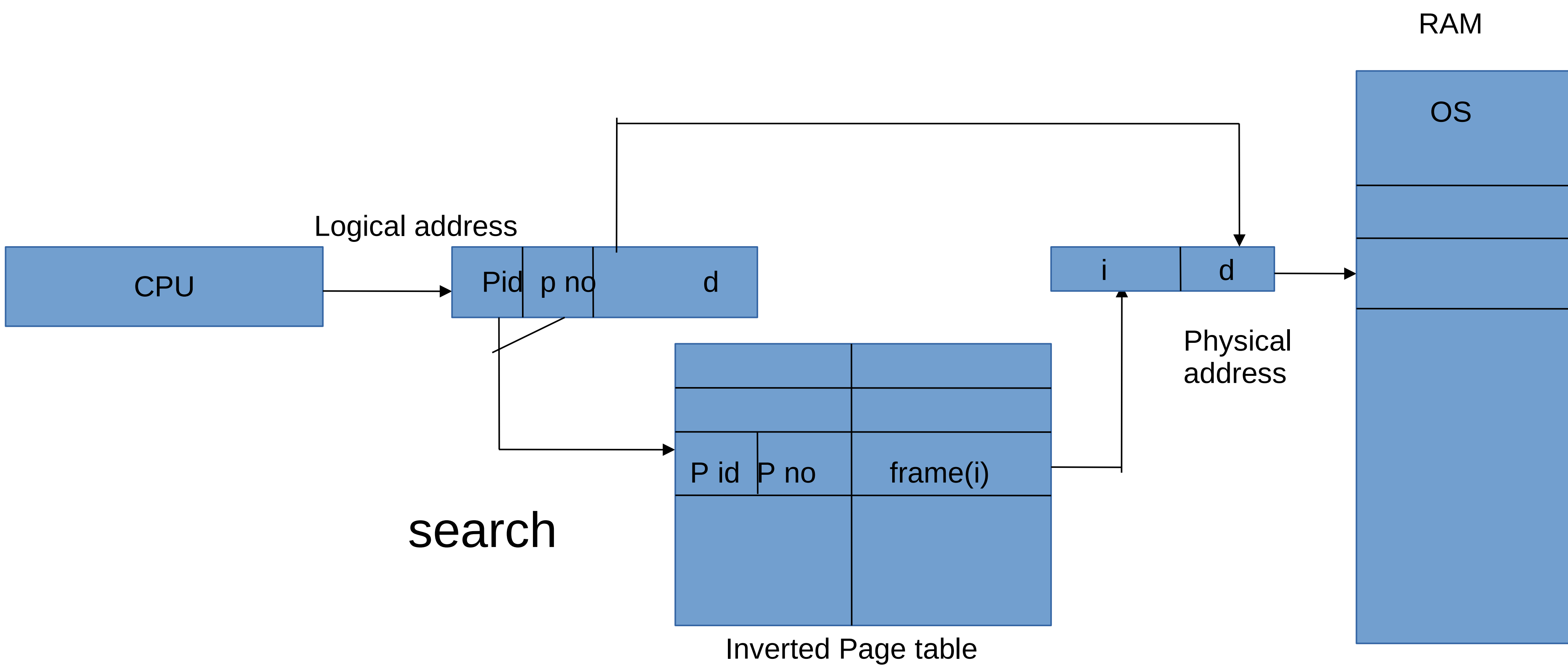
The inverted page table is then searched for this

<process id,page number>.

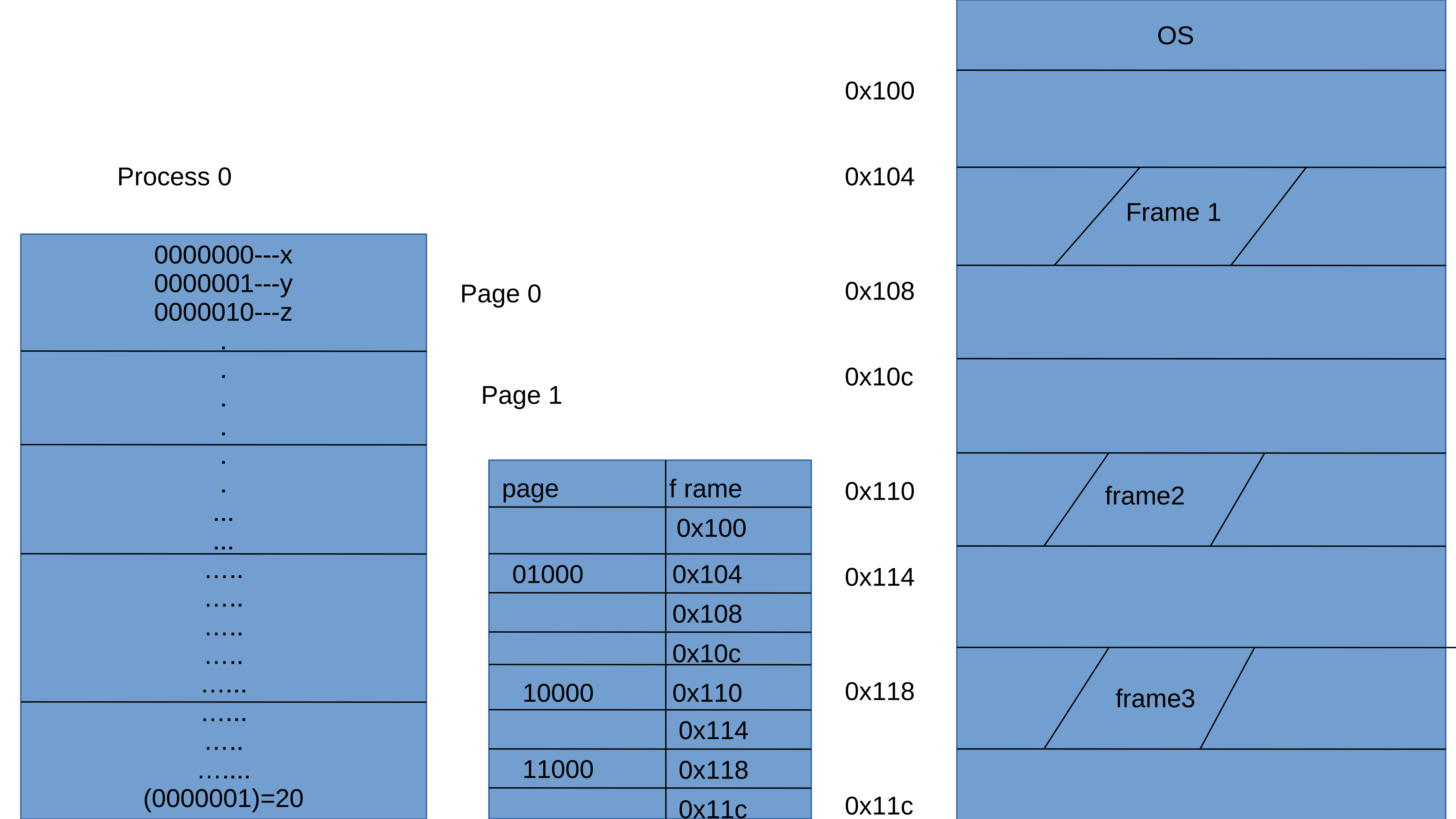
If the match is found say at entry I then the physical address

<i, offset> is generated.

If no match is found an illegal address access has been attempted .



RAM
OS
Logical address
CPU
Pid
p no
d
i
d
Physical address
P id
P no
frame(i)
search
Inverted Page table


Process 0
0000000---x
0000001---y
0000010---z
. . . . . .
...
...
.....
.....
.....
.....
......
......
.....
....
(0000001)=20
Page 0
Page 1
page
f rame
0x100
01000
0x104
0x108
0x10c
10000
0x110
0x114
11000
0x118
0x11c
OS
0x100
0x104
Frame 1
0x108
0x10c
0x110
frame2
0x114
0x118
frame3
0x11c

Total logical space 20 bytes m=5+2

Page size= frame size=4 n=2

CPU → (0000001)=20

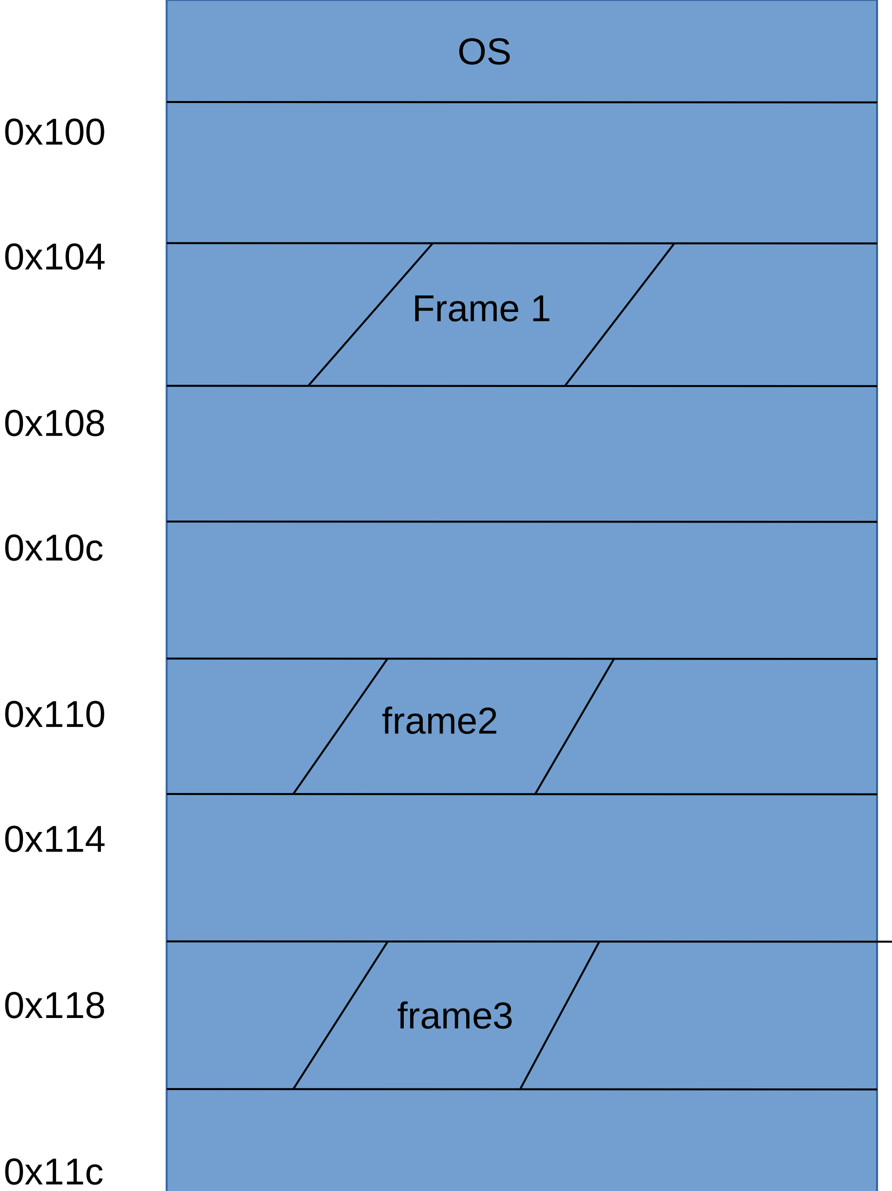
Page no offset

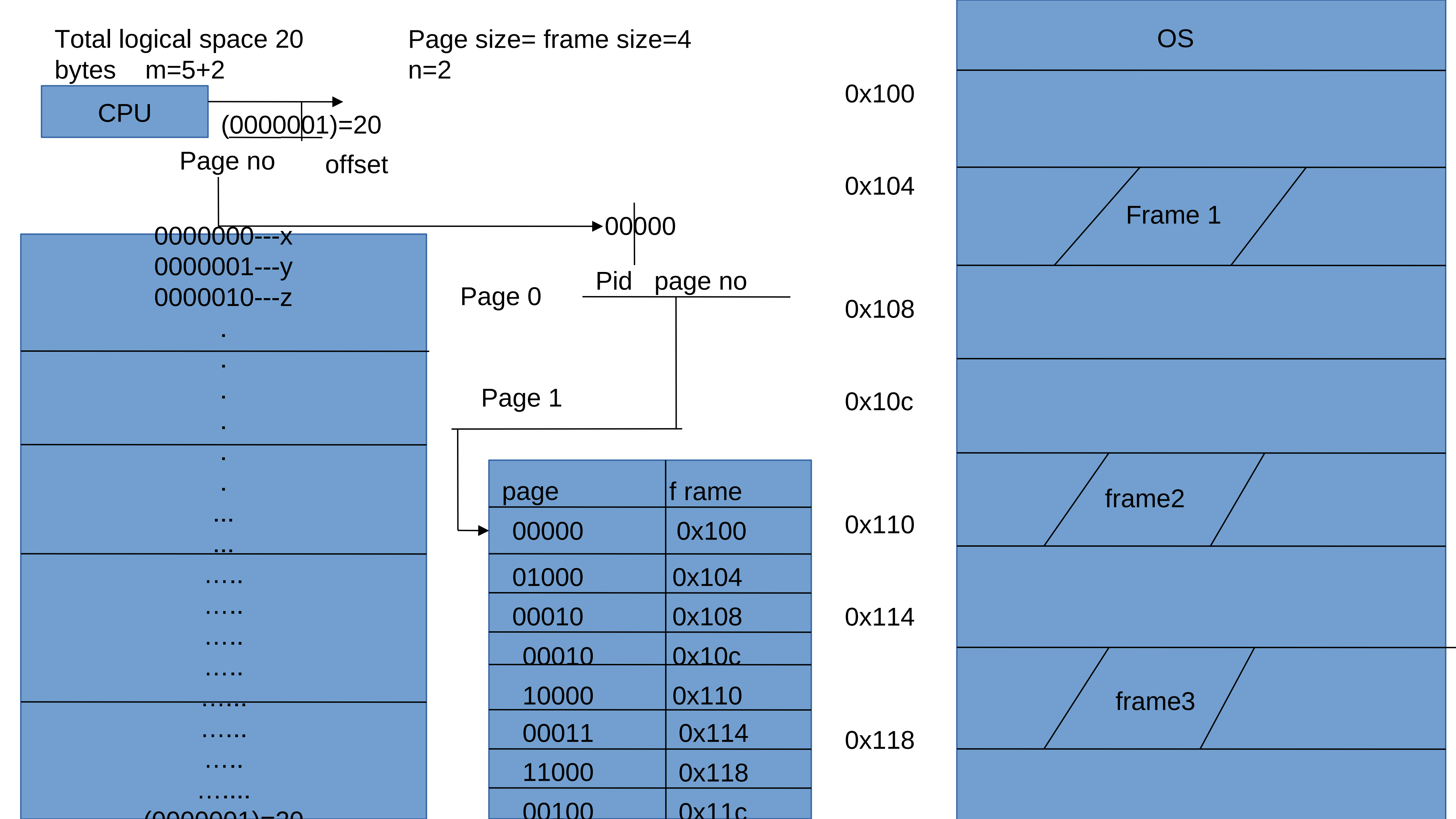
0000000---x
0000001---y
0000010---z
.
.
.
.
.
.
...
...
.....
.....
.....
.....
......
......
.....
....
(0000001)=20

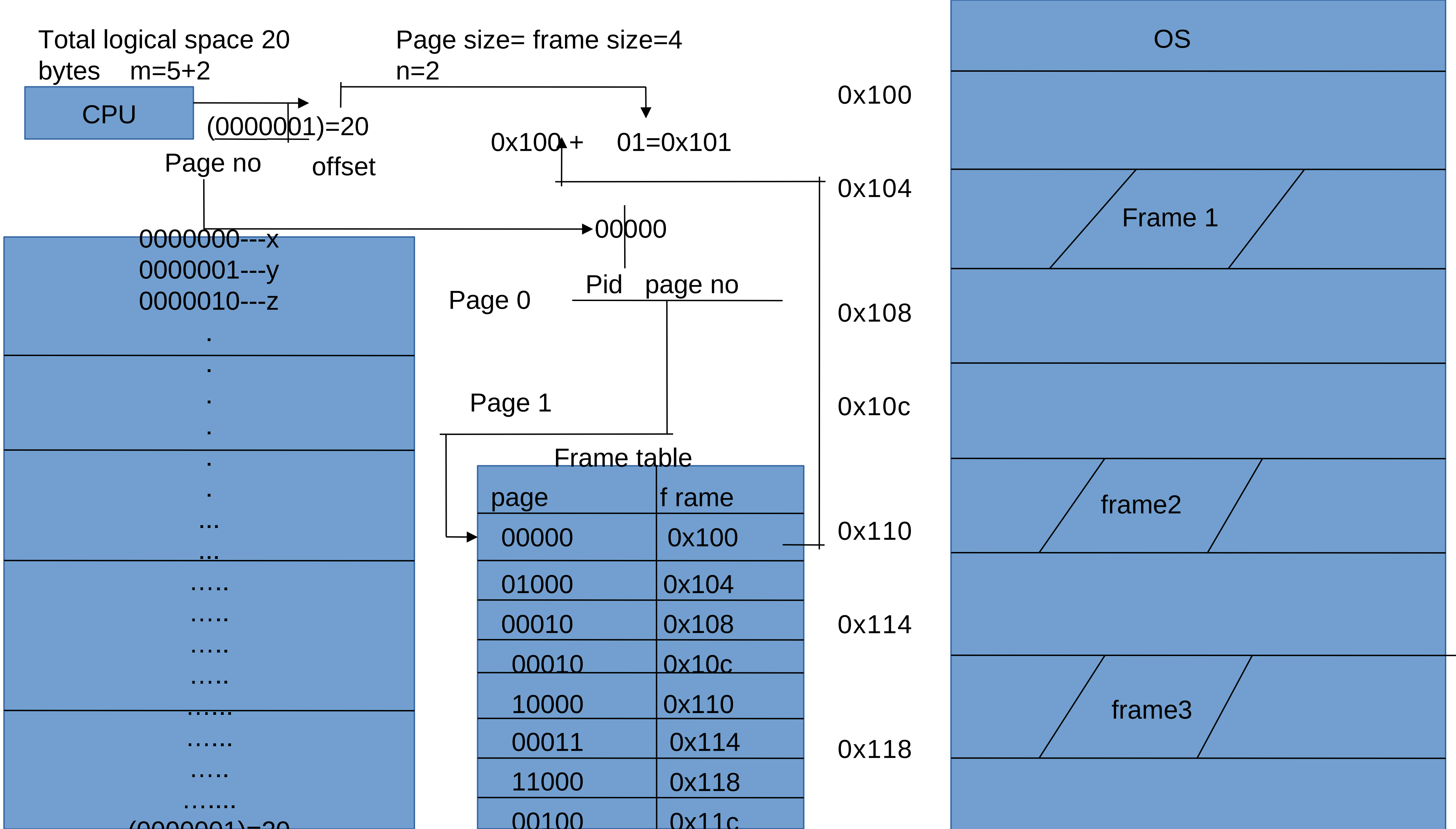
Page 0

Page 1

page	f rame
00000	0x100
01000	0x104
00001	0x108
00010	0x10c
10000	0x110
00011	0x114
11000	0x118
00100	0x11c




Total logical space 20 bytes m=5+2
Page size= frame size=4
n=2
CPU
(0000001)=20
Page no
offset
00000
Pid page no
Page 0
Page 1
0000000---x
0000001---y
0000010---z
page
f rame
00000
0x100
01000
0x104
00010
0x108
00010
0x10c
10000
0x110
00011
0x114
11000
0x118
00100
0x11c
(0000001)=20
0x100
0x104
0x108
0x10c
0x110
0x114
0x118
OS
Frame 1
frame2
frame3



page	f rame
00000	0x100
01000	0x104
00010	0x108
00010	0x10c
10000	0x110
00011	0x114
11000	0x118
00100	0x11c