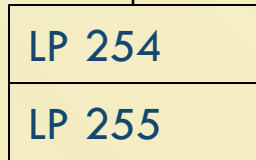
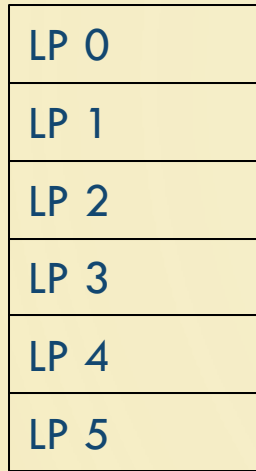
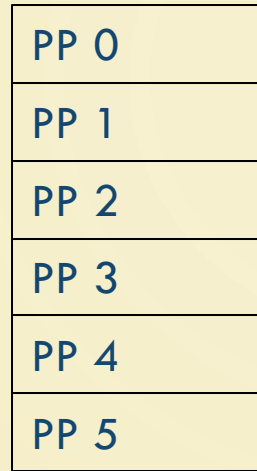


HW4: SIMPLE VIRTUAL MEMORY SYSTEM

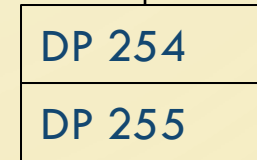
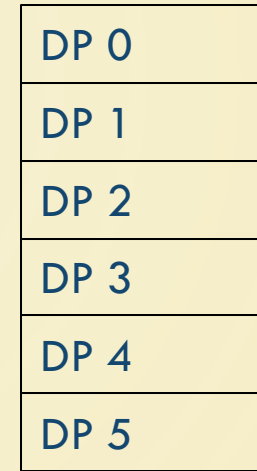
- The physical system has 64 page frames of 256 bytes = $2^6 * 2^8 = 2^{14}$ bytes
- The logical address space is 256 logical pages each of which is 256 bytes 2^{16} bytes.
- Addresses are 16 bits, treated as 8-bit logical page and 8-bit offset
- The 'disk' is setup as 256 pages of 256 bytes.
 - Implemented in program as a binary file.



Logical address space of process.



RAM



Backing Store (Disk)

PP	
PP	
PP	
PP	
PP	
PP	

PP	
PP	

Physical
page

Valid bit
(V or I)

Page Table: Logical/Physical
address translation.

If Valid bit is set the page is in
memory and can be accessed

If valid bit not set it is a page
fault.

	Occupied	Logical page
0	0	
1	1	78
2	0	
3	1	12
⋮	⋮	⋮
63	0	

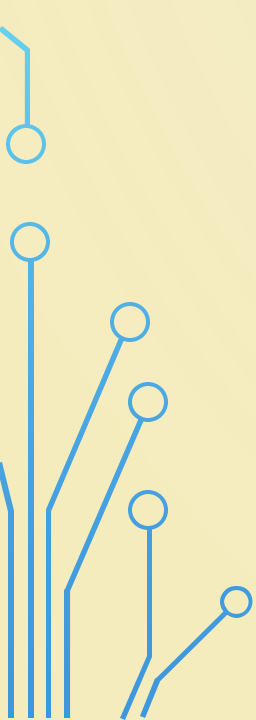
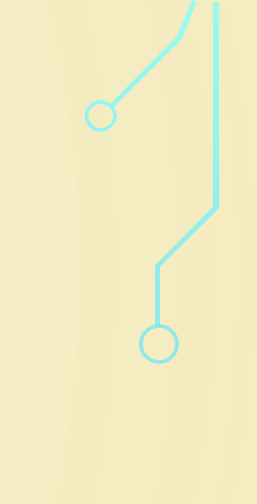
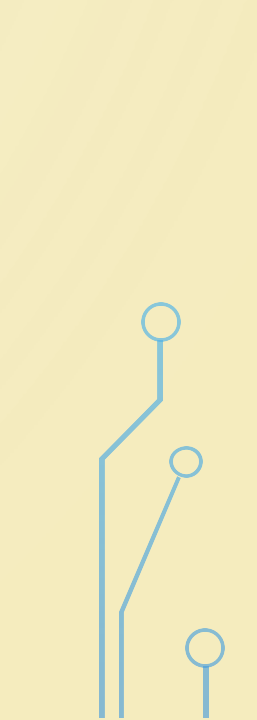
Physical page 1 is occupied by logical page 78

Physical page 3 is occupied by logical page 12



The CPU generates logical addresses between 0 – 16535

Your program:

- Divides logical address into logical page and offset within the page
 - Uses logical page number as offset into page table
 - If valid bit is set:
 - Fetches physical page frame number from PT
 - Computes physical address
 - Prints out contents at that address
- 
- 
- 

The CPU generates logical addresses between 0 – 16535

Your program:

- If valid bit not set it is a page fault because page not in memory
 - Scan Ram_Tracker to find free Physical Page Frame
 - If Found:
 - Read page in from disk and store in found page frame
 - Update RAM_Tracker to show physical page occupied and current occupant (logical page).
 - Update page table with new mapping
 - Update valid bit to valid
 - Print contents of address

The CPU generates logical addresses between 0 – 16535

Your program:

- If valid bit not set it is a page fault because page not in memory
 - Scan Ram_Tracker to find free Physical Page Frame
 - If NO free page frame Found:
 - Use some algorithm (TBD) to pick a physical page frame to evict
 - Read page in from disk and store in victim page frame
 - Update RAM_T to show new page occupant
 - Update page table entry of Evicted page to show no longer valid
 - Update page table with new mapping
 - Update valid bit to valid
 - Print contents of address

```
unsigned char buf [256] ;           //contents of a page
unsigned short int Addresses[tbd] ; //reference string
unsigned short int Current ;        //current address
srand(9) ;                          //all generate same address string
```

```
for(i = 0; i < tbd; i++) //generate program addresses
{ Addresses[i] = rand() % 65535 ;
  printf("Address is %u \n", Addresses[i]) ;
}
```

//64 physical page frames

```
unsigned char RAM [64][256] = { [0 ... 63] = { [0 ... 255] = -1 } };
```



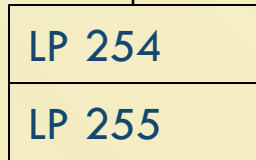
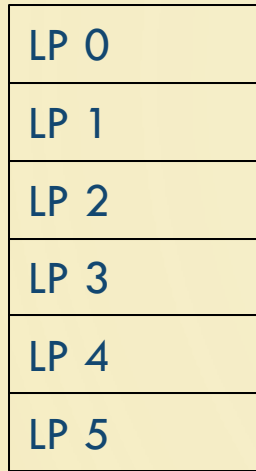
```
unsigned char buf [256] ;           //contents of a page
unsigned short int Addresses[tbd] ; //reference string
unsigned short int Current ;        //current address
srand(9) ;                          //all generate same address string
```

```
// generating pages for backend storage ('disk')
```

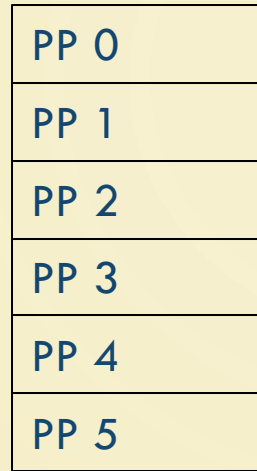
```
for (i = 0 ; i < 256 ; i++)
    buf[i] = (unsigned char) i ;
```

```
fp = fopen("Back.bin", "wb") ; //implemented as binary file
```

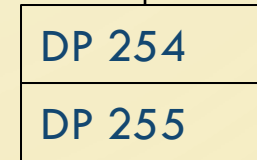
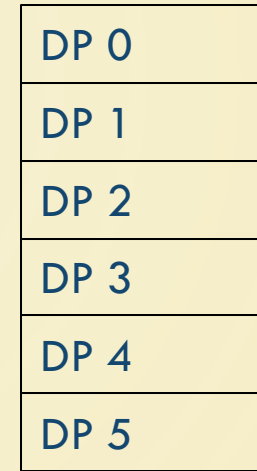
```
for (i = 0; i < 256 ; i++)
    fwrite(buf,256, 1, fp) ; //writing 256 pages to 'disk' to serve as
    //complete process image
```




Logical address space of process.



RAM



Backing Store (Disk)



```
for (I = 0; I < num_addresses ; i++)  
  {Current = Addresses[i] ;  
   logical Page = MS 16 bits ;  
   Offset = LS 16 bits ;
```

Go to page table to determine if in memory

Yes:

compute address and print out contents

NO:

Find Free Frame in RAM and bring page from disk into memory

update data structures

print out contents at that address

NO: And no free Frame

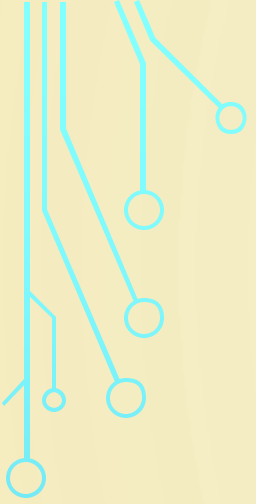
Pick victim page frame

Bring page in from disk and load into victim page frame

update data structures

print out contents of address






```
unsigned char buf [256] ; //why??
```

Ease of use. Unsigned chars hold values from 0 .. 255

Initialized such that offset n within any given page will have a value of n.

```
unsigned short int Addresses[tbd] ; //reference string
                                   //(except complete address,
                                   // you have to pick out page/offset)
```

Why unsigned short int?

- 1) don't want negative addresses!
 - 2) 2-bytes. //nice match with 8-bit page and 8-bit offset
 - 3) Easy to separate page/offset using binary operations.
- 

Recall bitwise operators:

Bitwise AND '&'

compares each bit of 2 numbers

the resulting number has a bit value of 1 only if each of the corresponding bits have a value of 1. otherwise it is 0.

example: **Unsigned char X = A & B ;**

A	001110011
B	<u>110111001</u>
X	000110001

Bitwise shift:

$A = B \gg \text{positions}$

$A = B \ll \text{positions}$

Assume $B = 1110\ 0011$.

shift 1 position: $0111\ 0001$: shift bits to the right one position,
fill in leftmost shifted bit with 0
shift rightmost bit to the bit-bucket. i.e., dropped

shift 2 positions: $0011\ 1000$: shift bits to the right one more position,
fill in leftmost shifted bit with 0
shift rightmost bit to the bit-bucket. i.e., dropped

Bitwise shift:

$A = B \gg \text{positions}$

$A = B \ll \text{positions}$

Assume $B = 1110\ 0011$.

$A = B \gg 4: \quad 0000\ 1110$

$1110011 \rightarrow \underline{0000}\ 1110\ \underline{011}$
shifted in. shifted out

Recall bitwise operators:

Bitwise AND:

compares each bit of 2 numbers

the resulting number has a bit value of 1 only if each of the corresponding bits have a value of 1. otherwise it is 0.

example:

```
001110011
110111001
-----
000110001
```

0001 0001 1101 0010

So what??

Assume Logical address Current = 4562 =

0001 0001 1101 0010
page offset

Current & 0xFF provides the offset within the page: **offset = Current & 0xFF** (Recall 0x = hex, FF = 1111 1111)

4562 = 0001 0001 1101 0010

0xFF = 0000 0000 1111 1111

Off = 0000 0000 1101 0010

So what??

Assume Logical address Current = 4562 =

0001 0001 1101 0010
page offset

Need to get page number. To get a number between 0 – 255 must shift page number into least significant digits.


0000 0000 0001 0001

How to do that?

4562 = 0001 0001 1101 0010

How to shift high order 8 bits into low order 8 bits?

For safety should also perform bitwise & with shifted result.

- 
- The background is a solid light yellow color. In the four corners, there are decorative elements consisting of thin blue lines that resemble circuit traces or a stylized tree structure. These lines end in small open circles. The lines are more dense in the bottom-left and top-left corners and more sparse in the top-right and bottom-right corners.
- I will post deliverables and page replacement requirements.

APPROACH

- Step 1:
 - Create basic program architecture
 - Initialize data structures (PT, RAM_Tracker, Backend store)
 - Go through address loop and translate each address into page number and offset
 - Check and make sure this is done correctly.
 - Use page number as entry into page table.
- Step 2 (Demand Paging)
 - Since no pages are in memory initially there will be many page faults
 - Bring pages into memory as faults occur, update page table and RAM-Tracker
 - Print out contents
 - Check to make sure the contents are correct: value should be equal to offset
 - If no free frames ignore

APPROACH

- Step 3:
 - Come up with page replacement algorithm
 - Use it to pick victim page
 - Bring in page and update data structures

TO BE ADDED

- Deliverables
- Page Replacement Algorithm