

ELEC 377 – Operating Systems

Week 6 – Class 1

Last Class

- Deadlock
- Bankers Algorithm

Admin

- Lab today
- Quiz 2 Tomorrow

Today

- Memory Management
 - ◇ Binding
 - ◇ memory management
 - ◇ dynamic memory
 - ◇ overlays
 - ◇ Swapping
 - ◇ Contiguous Allocation
 - ◇ Paging

Binding Instructions and Data

- Entities in the original program must be bound to a location in memory
 - ◇ `int count;`
- Programs may reside in different parts of memory
- Three different stages
 - ◇ Compile (and linkage) time (MS-DOS .COM)
 - ◇ Load Time – When loader loads program into memory, addresses are resolved
 - ◇ Execution Time – programs move in memory
 - hardware support required

Binding Instructions and Data

- Compile Time (or Assembly Time)
 - ◇ use absolute addressing
 - ◇ code can only be loaded at a particular location

.text

.org 0x734

.start 0x734

mov AL,L

$AL \leftarrow L$

add AL,P

$AL \leftarrow AL + P$

...

.data

.org 9af

L: .byte 12

P: .byte 42

Binding Instructions and Data

- Load Time
 - ◇ header of load file contains locations of referneces
 - ◇ loader (Part of OS) resolves them at the time they are read into memory

.text

start:

mov AL,L *<= header gives this location in memory*

add AL,P *<= and this location in memory*

...

.data

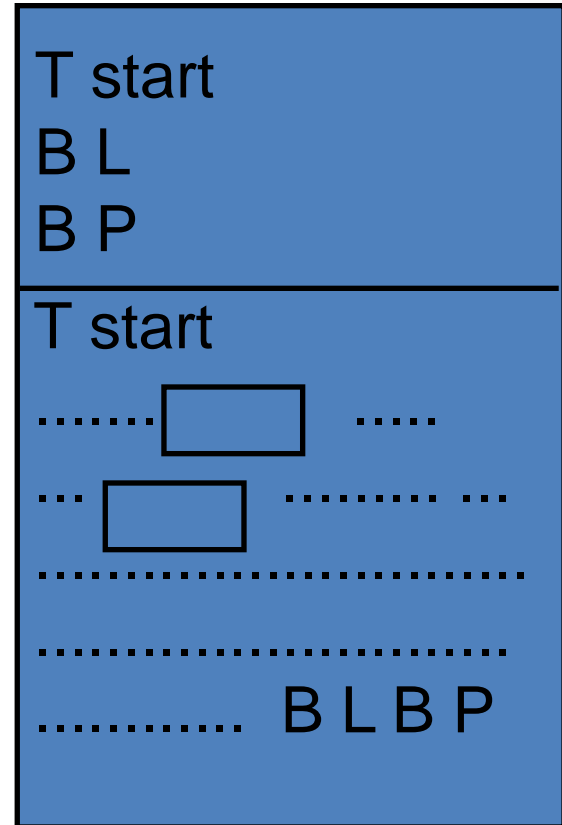
L: .byte 12

P: .byte 42

Load Time Module

```
.text
start:
mov AL,L
add AL,P
...
.data
L: .byte 12
P: .byte 42
```

bar.c

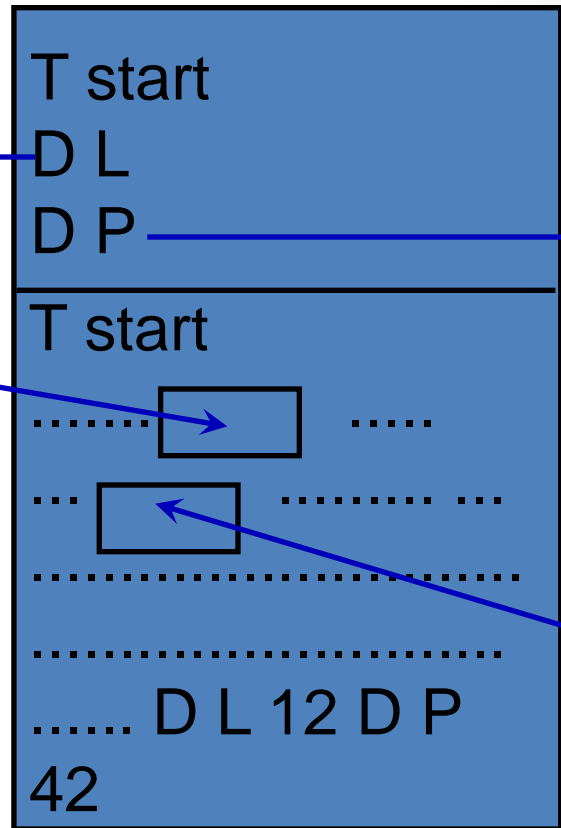


bar.o

Load Time Module

```
.text
start:
mov AL,L
add AL,P
...
.data
L: .byte 12
P: .byte 42
```

bar.c



bar.o

Load Time (Loader)

- In load time binding, the loader resolves the address

```
int load_program(void * startAddress, char * prgName){
    ProgHeader header;
    Symbol sym;
    ... open file prgName for read ...
    ... read the header into header...
    ... read the rest of the file (header.size) to startAddress
    for each sym in header
        ... adjust each reference to the symbol by offset
        from startAddress ...
}
```

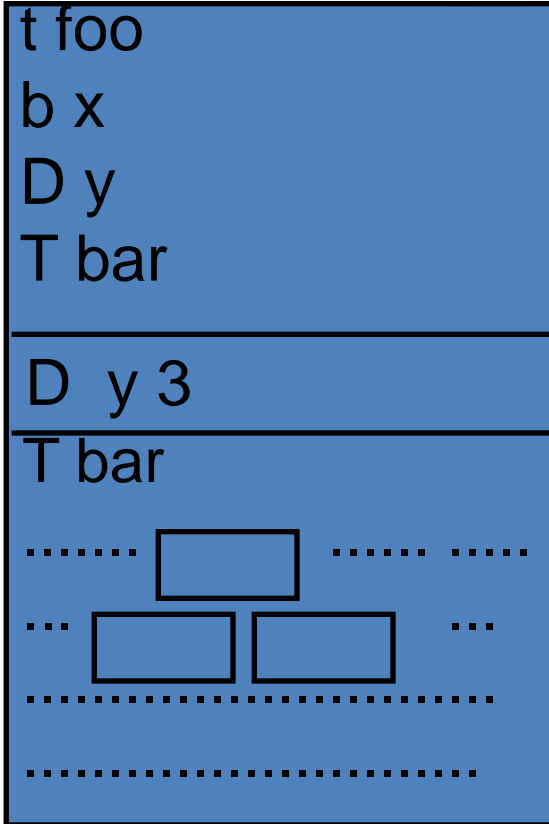
Binding Instructions and Data

- Execution Time
 - ◇ Similar to Compile Time, but hardware looks after translation

Compile

```
extern int x;  
extern foo();  
  
int y = 3;  
int bar(int z){  
...  
    x = y+z;  
    x += foo();  
}
```

bar.c

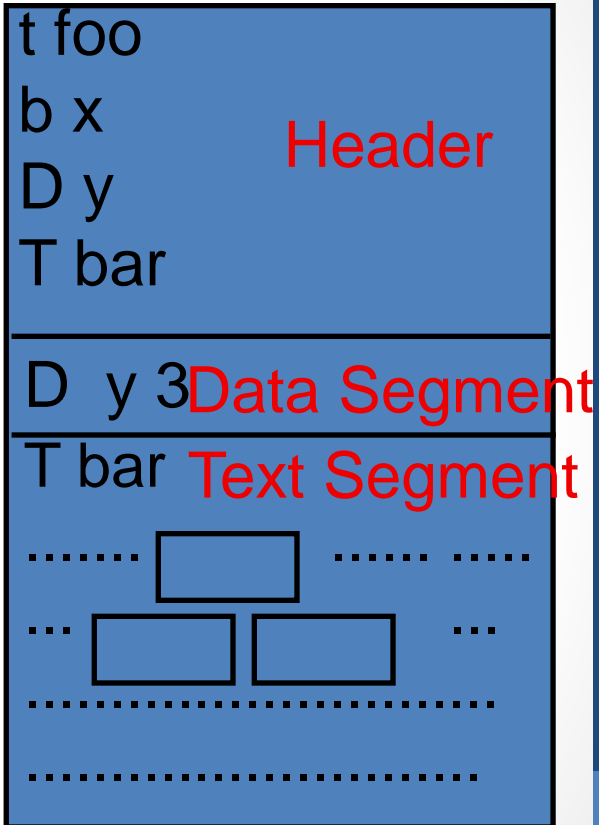


bar.o

Compile

```
extern int x;  
extern foo();  
  
int y = 3;  
int bar(int z){  
...  
    x = y+z;  
    x += foo();  
}
```

bar.c

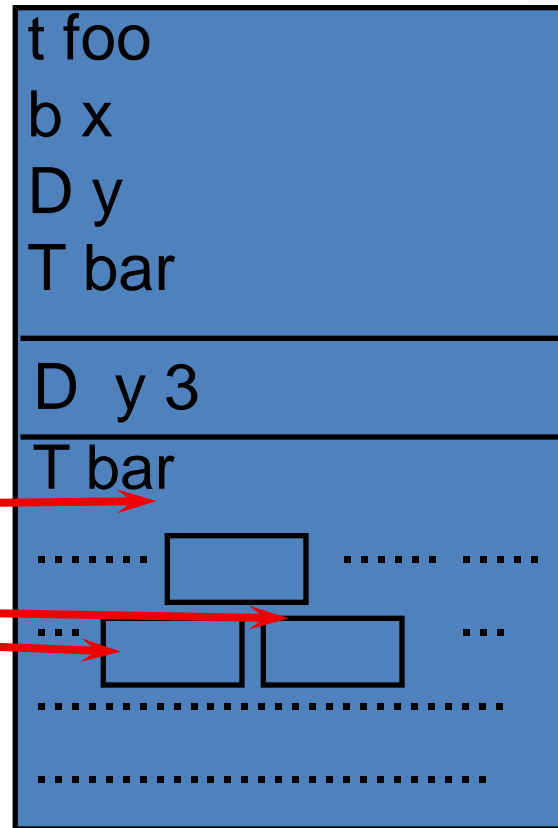


bar.o

Compile

```
extern int x;  
extern foo();  
  
int y = 3;  
int bar(int z){  
...  
    x = y+z;  
    x += foo();  
}
```

bar.c

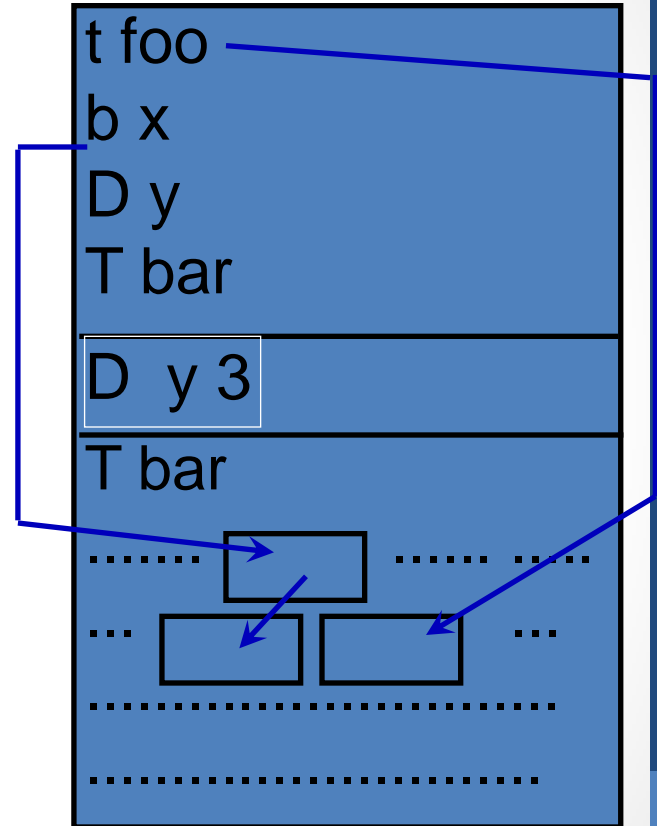


bar.o

Compile

```
extern int x;  
extern foo();  
  
int y = 3;  
int bar(int z){  
...  
    x = y+z;  
    x += foo();  
}
```

bar.c

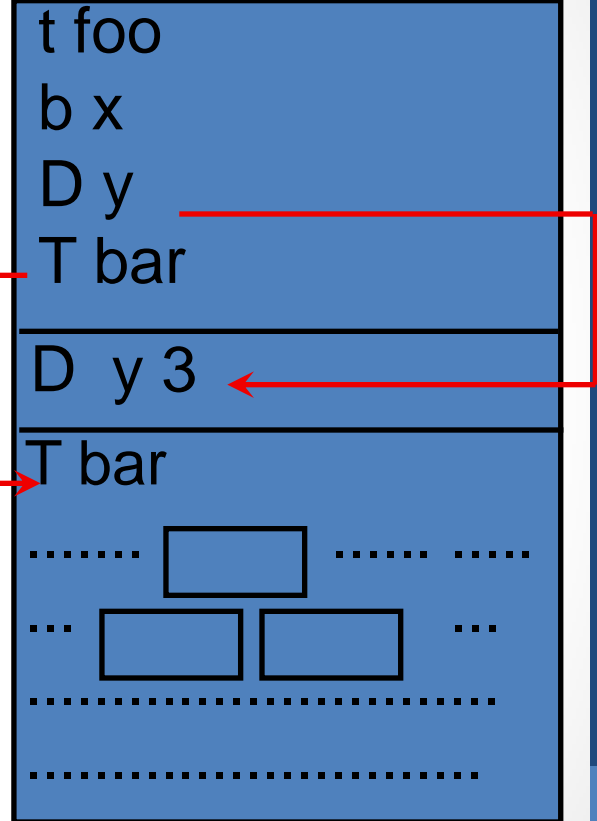


bar.o

Compile

```
extern int x;  
extern foo();  
  
int y = 3;  
int bar(int z){  
...  
    x = y+z;  
    x += foo();  
}
```

bar.c



bar.o

Linking

...

```
int x;
```

```
extern int y;
```

```
FILE * f;
```

```
int main(int argc, char * argv[]){
```

```
    f = fopen("/proc/lab3", "r");
```

```
    fprintf(stderr, "foobar %d", x + bar(y));
```

...

```
}
```

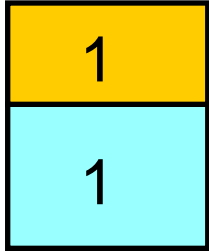
```
int foo(){...}
```

Linking

```
...
int x;
extern int y;
FILE * f;
int main(int argc, char * argv[]){
    f= fopen("/proc/lab3", "r");
    fprintf(stderr, "foobar %d", x+bar(y));
    ...
}
int foo(){...}
```

■ Defined
■ Undefined bar.o
■ Undefined library

Linking—combine to single executable

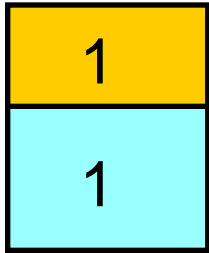


foo.o

def **main**,x

undef **fopen**,bar

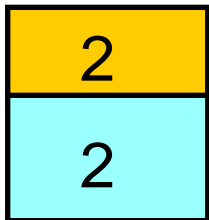
Linking—combine to single executable



foo.o

def **main**,x

undef **fopen**,bar

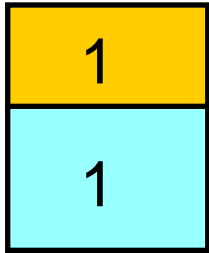


bar.o

def **bar**,y

undef **foo**,x

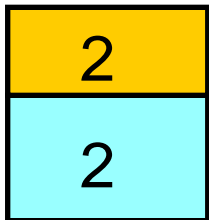
Linking—combine to single executable



foo.o

def **main**,x

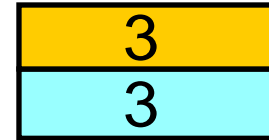
undef **fopen**,bar



bar.o

def **bar**,y

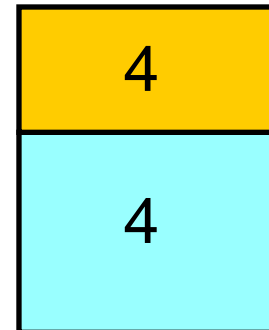
undef **foo**,x



crt.o

- defines **entry**

- undef **main**

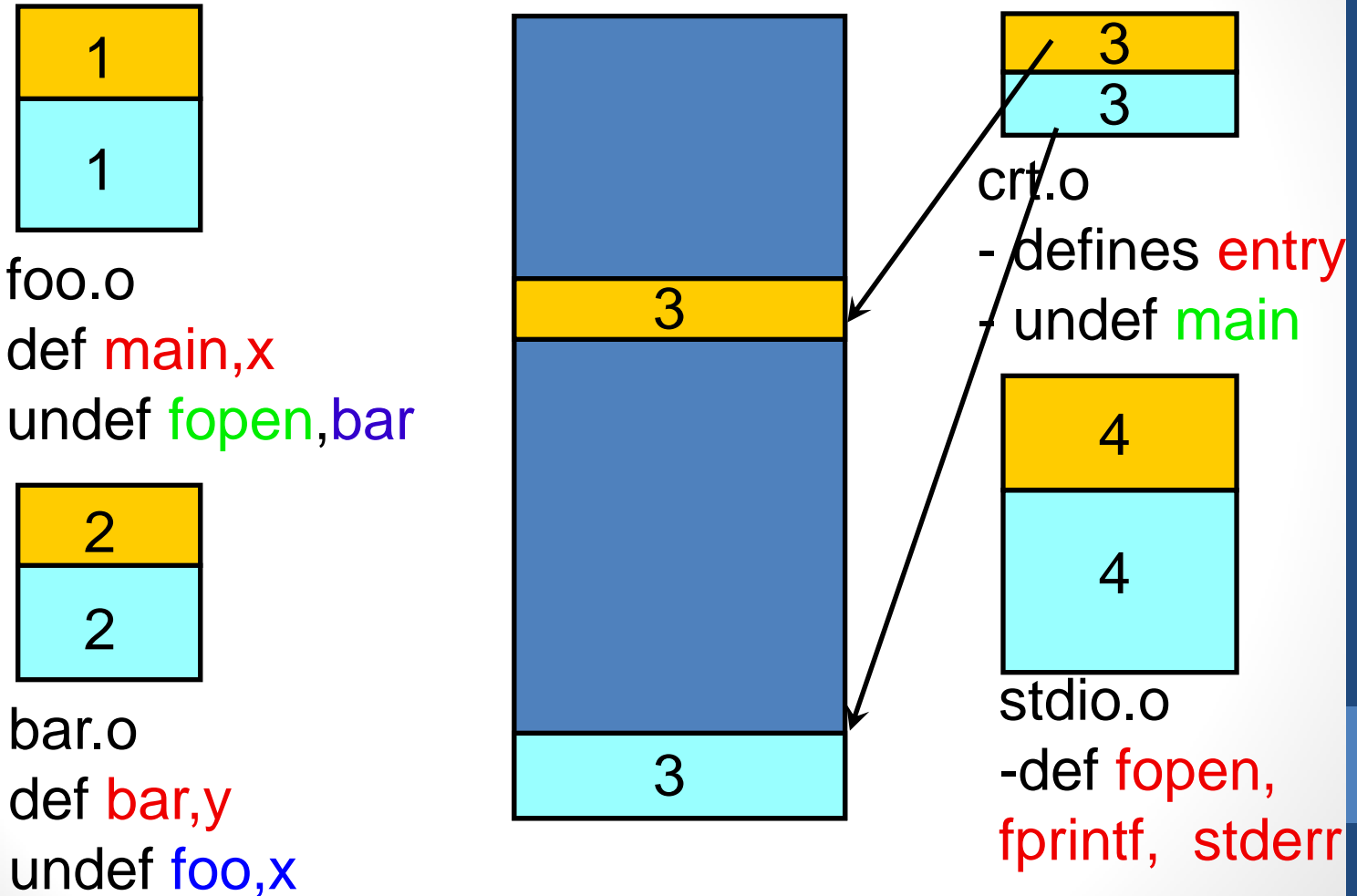


stdio.o

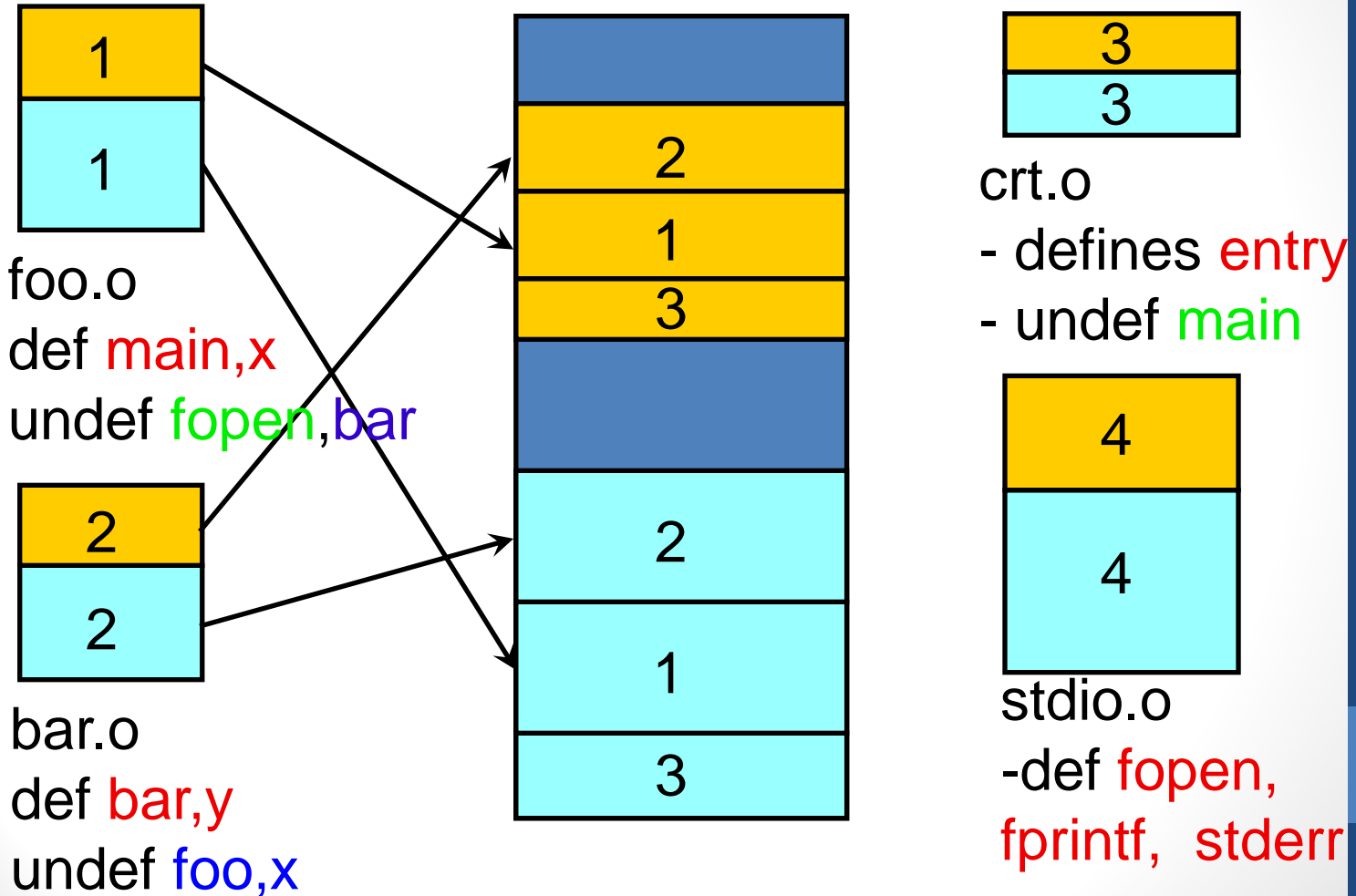
-def **fopen**,

fprintf, **stderr**

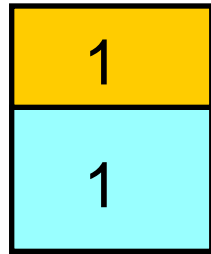
Linking—combine to single executable



Linking—combine to single executable



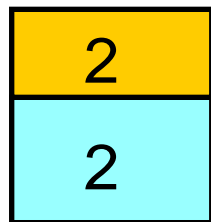
Linking—combine to single executable



foo.o

def **main**,x

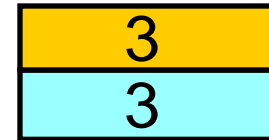
undef **fopen**,bar



bar.o

def **bar**,y

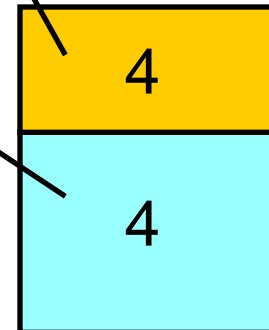
undef **foo**,x



crt.o

- defines **entry**

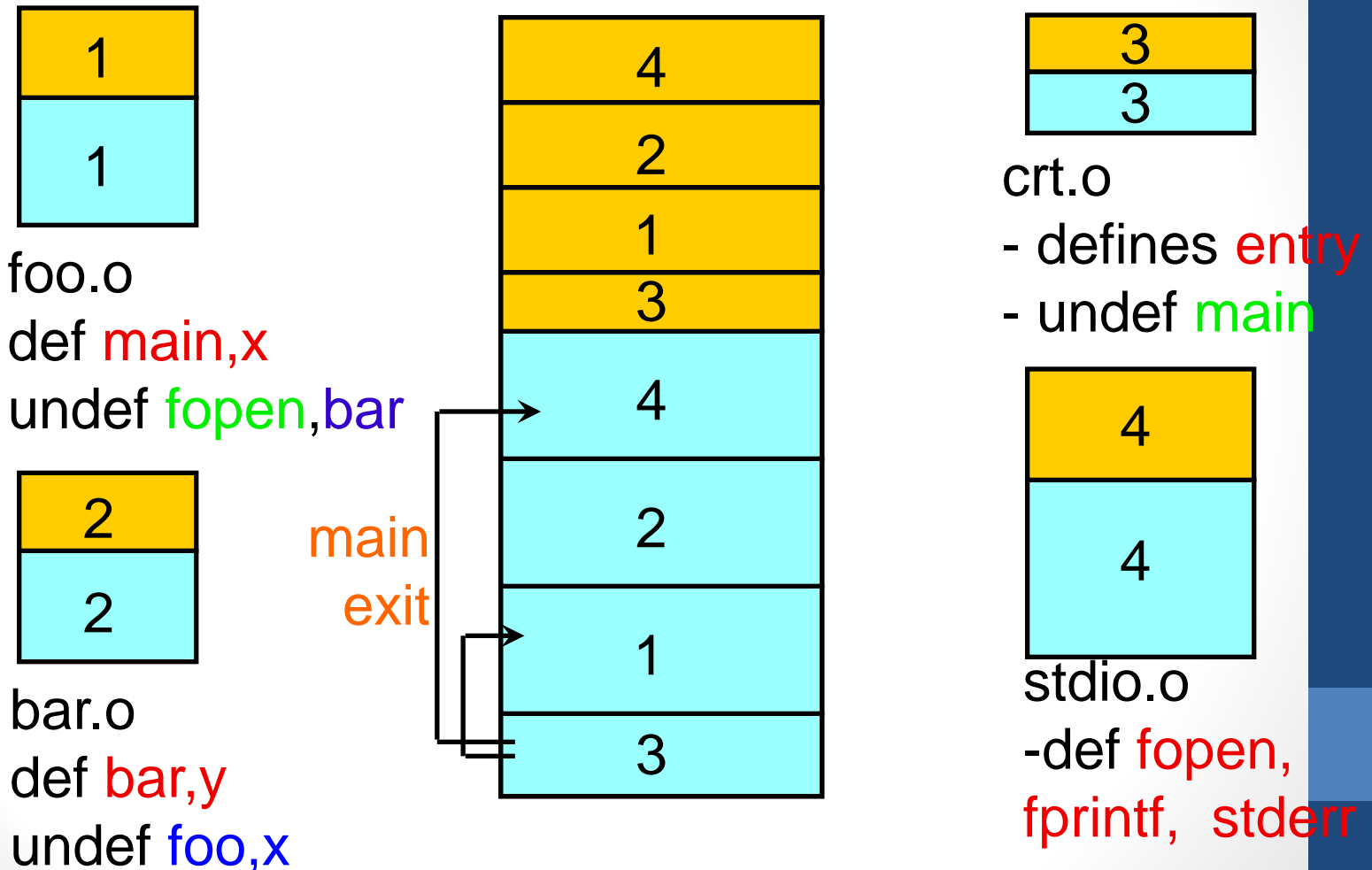
- undef **main**



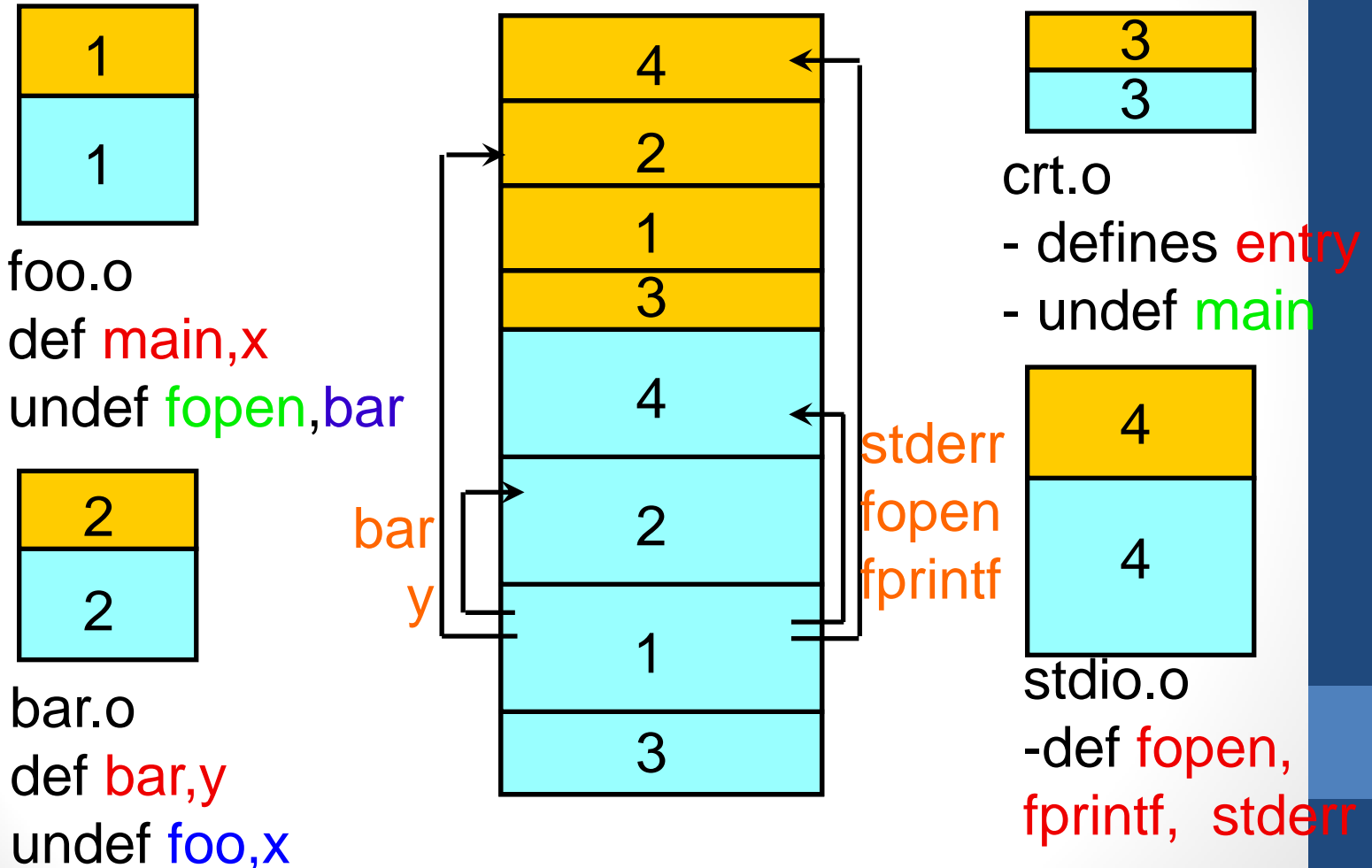
stdio.o

-def **fopen**,
fprintf, **stderr**

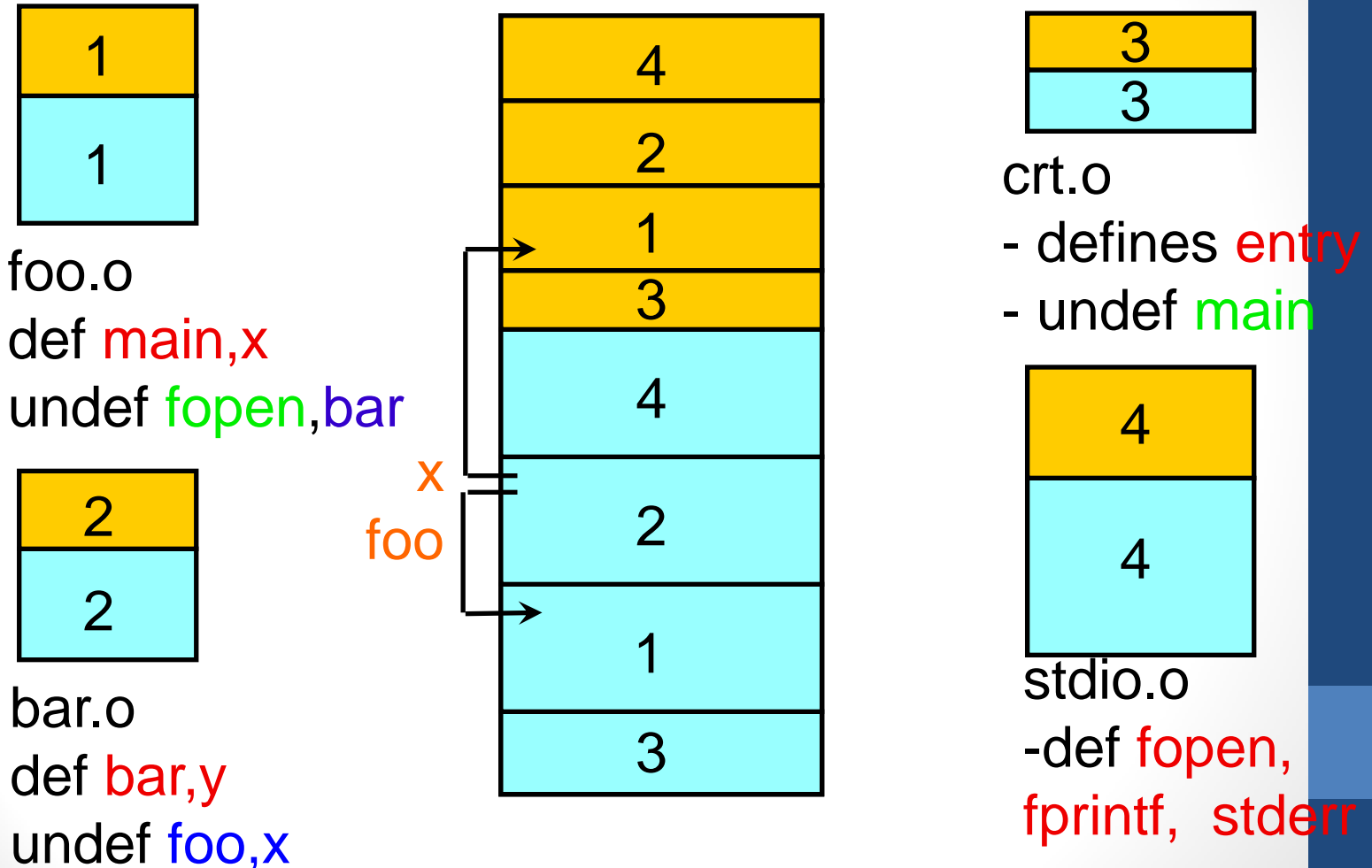
Linking—combine to single executable



Linking—combine to single executable



Linking—combine to single executable



Logical vs Physical Address Space

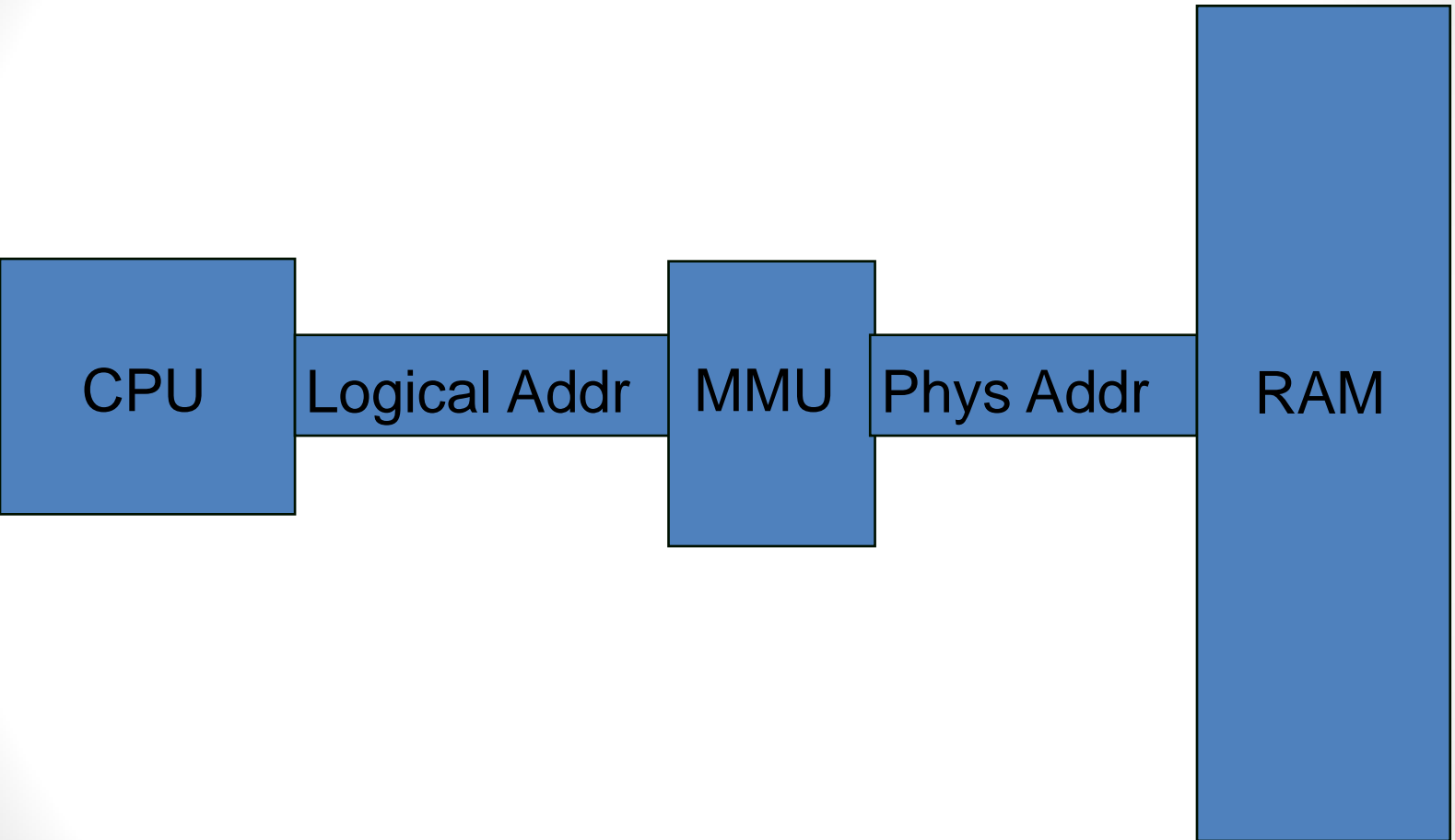
Central Concept to Memory Management

- Logical Address
 - ◇ address generated by CPU
 - ◇ also known as virtual address
- Physical Address
 - ◇ location in physical memory
- Logical and Physical address are the same in compile and load time address binding. They differ in execution time binding
- User program only deals with logical addresses. It never sees the physical address

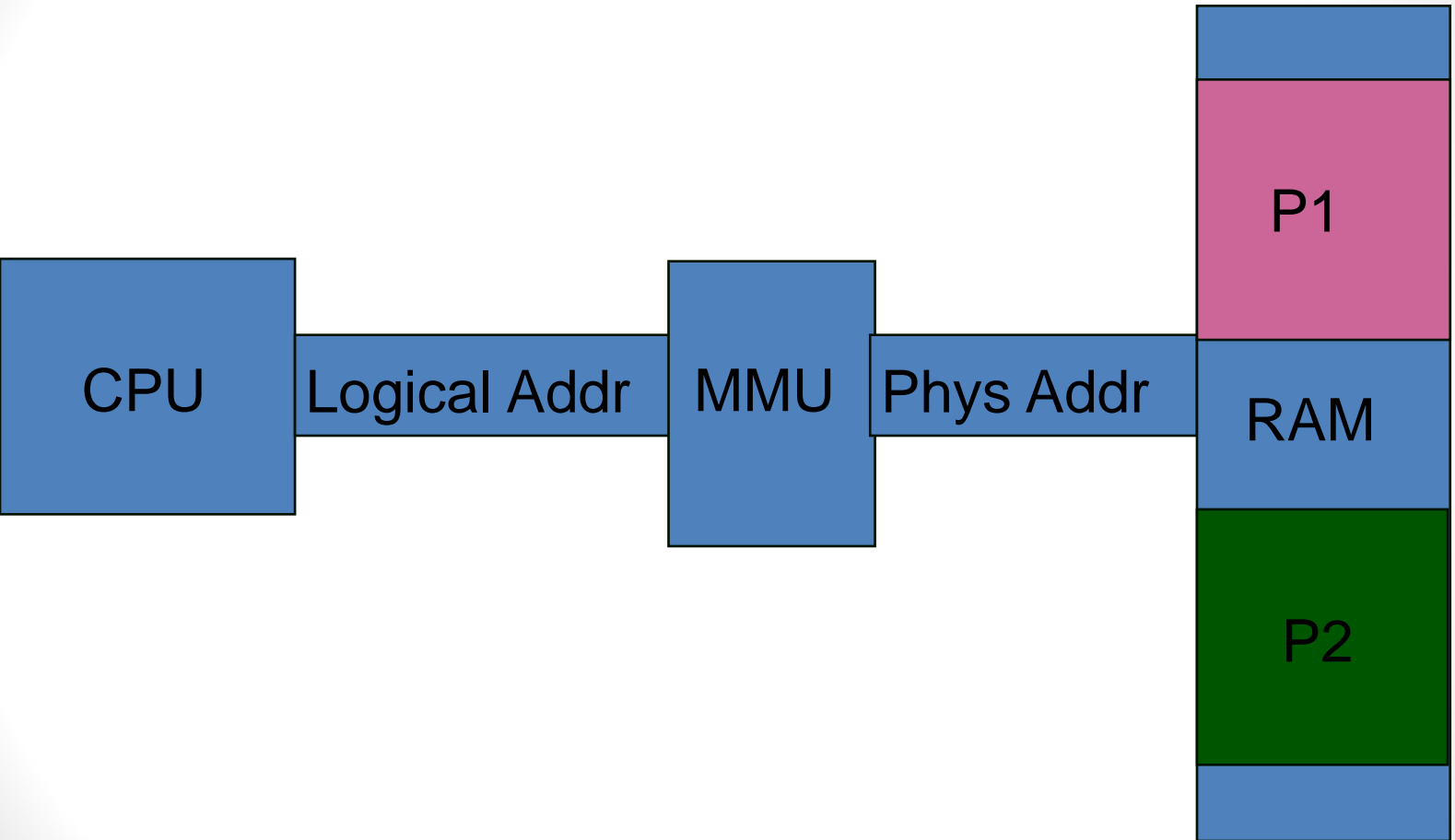
Memory Management Unit (MMU)

- Hardware that maps virtual to physical address
 - ◇ many different approaches
- One simple approach is to have a single register that is added to every virtual address
 - ◇ Similar to the original memory protection scheme talked about in Week 1.
 - ◇ Limit register is now size of memory space
 - ◇ base register is called the **relocation** register
 - ◇ Used by MSDOS on 386, PDP-11
- Logical addresses ($0 \dots max$)
- Physical address ($R \dots R+max$)
 - ◇ R is the value of the relocation register

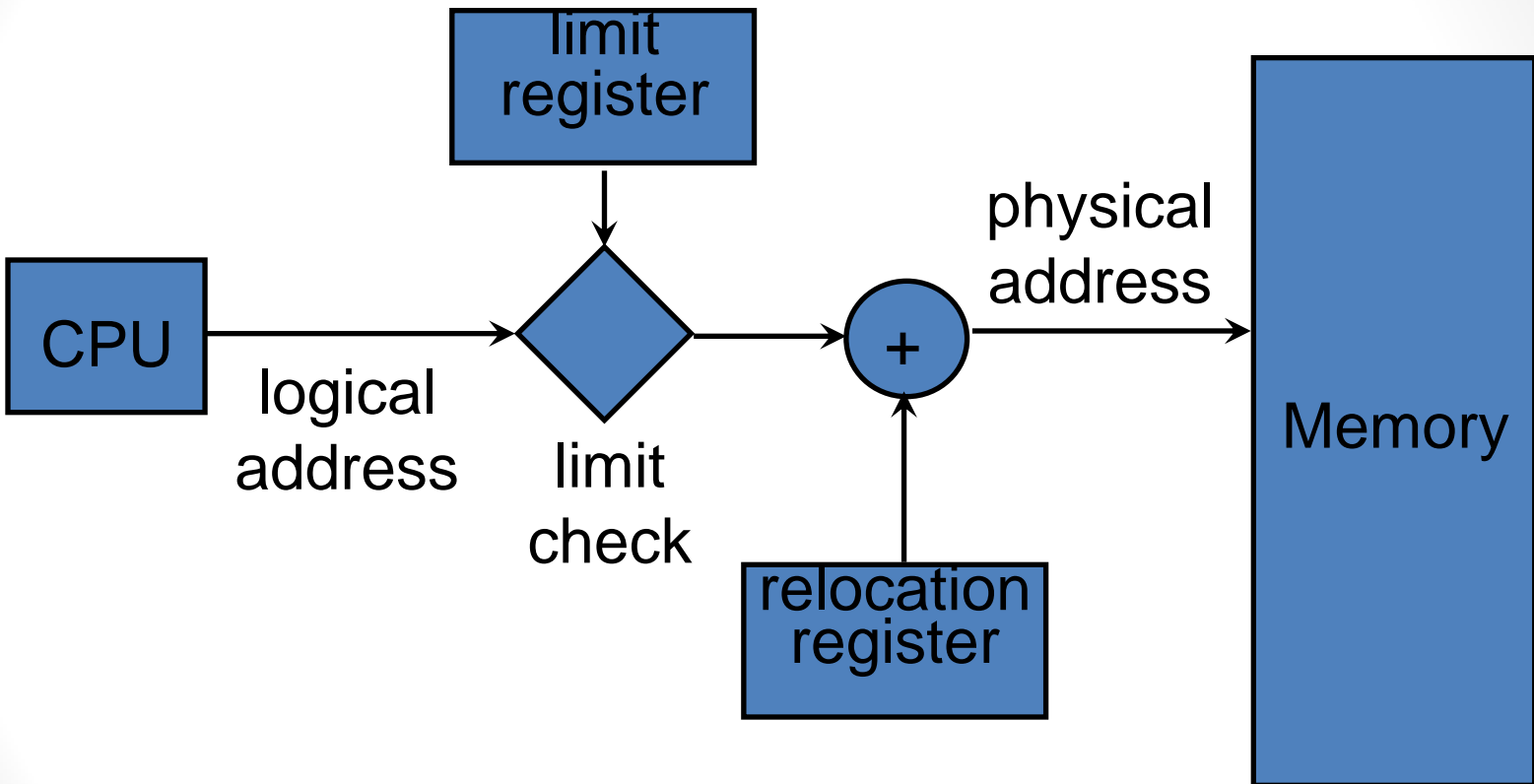
Memory Management Unit (MMU)



Memory Management Unit (MMU)



Simple MMU



Contiguous Allocation

- main memory is divided into two parts
 - ◇ operating system (usually in same part as interrupt vector)
 - ◇ User memory (divided among processes)
- single partition allocation
 - ◇ simple allocation, each process gets a single chunk of main memory to live in
 - ◇ hardware relocation register and limit register provides relocation and memory protection.

Contiguous Allocation

- When system first starts, allocation is simple
 - ◇ One block of memory, and as each process starts, allocate the memory to the process

Contiguous Allocation

- When system first starts, allocation is simple
 - ◇ One block of memory, and as each process starts, allocate the memory to the process



A horizontal bar representing memory allocation. The bar is divided into two sections: a smaller blue section on the left labeled 'P1' and a larger red section on the right. The entire bar has a black border.

P1

Contiguous Allocation

- When system first starts, allocation is simple
 - ◇ One block of memory, and as each process starts, allocate the memory to the process



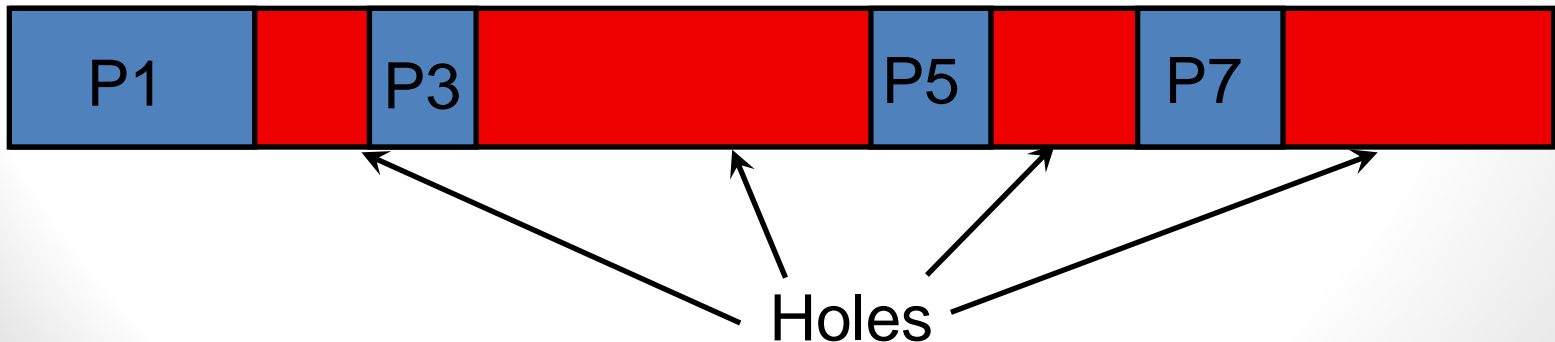
Contiguous Allocation

- When system first starts, allocation is simple
 - ◇ One block of memory, and as each process starts, allocate the memory to the process



Contiguous Allocation

- When system first starts, allocation is simple
 - ◇ One block of memory, and as each process starts, allocate the memory to the process
- Some processes run long, some quit soon after they start
 - ◇ Not related to size. A large program can run short or long...



Contiguous Allocation

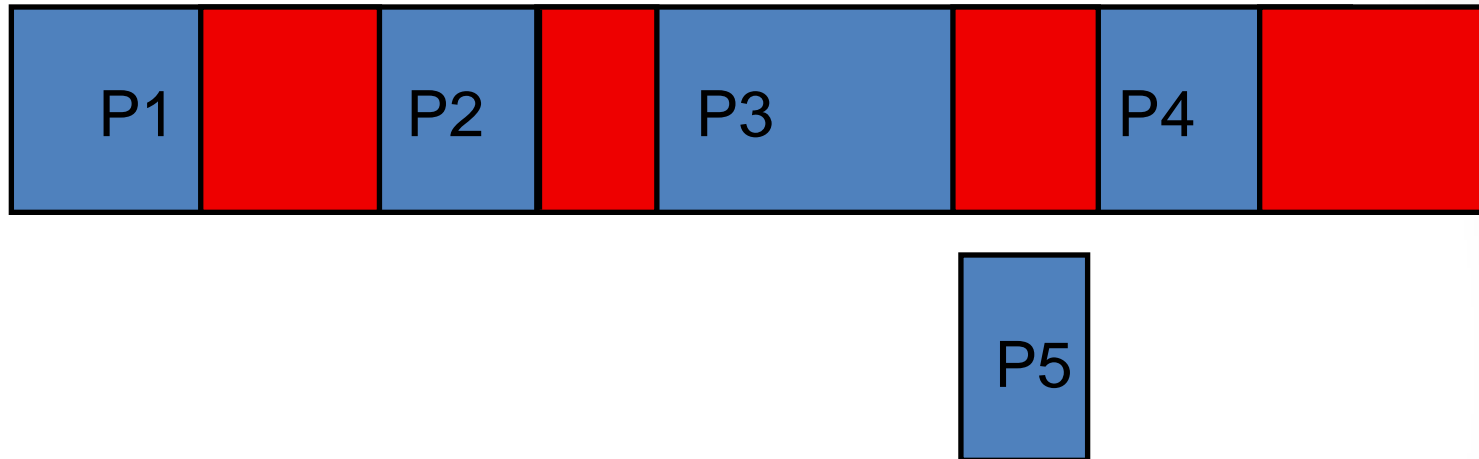
- User memory must be allocated to processes
 - ◇ fixed size segments – IBM MFT – obsolete
 - ◇ variable size segments
- OS keeps list of *holes*
 - ◇ Memory not allocated to a process
 - ◇ When a process is started, find a hole big enough to hold it
 - ◇ When a process ends, add the memory to the free list
- General memory allocation problem
 - ◇ merge adjacent holes
 - on allocation or on free?

Storage Allocation

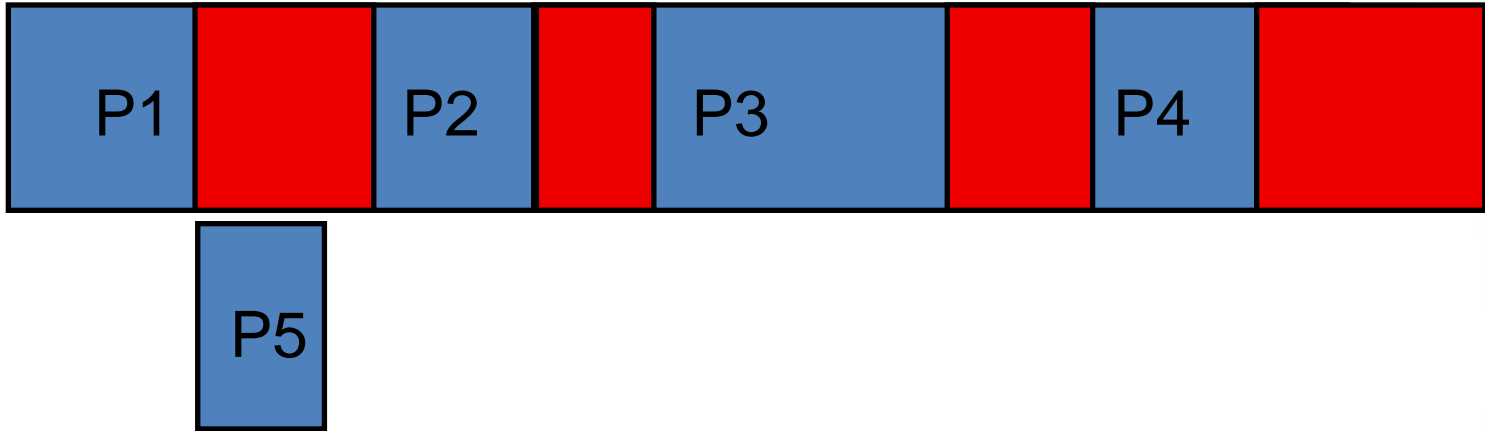
Three general approaches

- First Fit
 - ◇ use the first hole on the free list that is big enough
 - ◇ always search from beginning?
 - ◇ search from previous location
 - ◇ only look at part of list
- Best Fit
 - ◇ smallest block that is large enough
 - ◇ search entire list
- Worst Fit
 - ◇ largest block (largest remainder)
 - ◇ worst algorithm

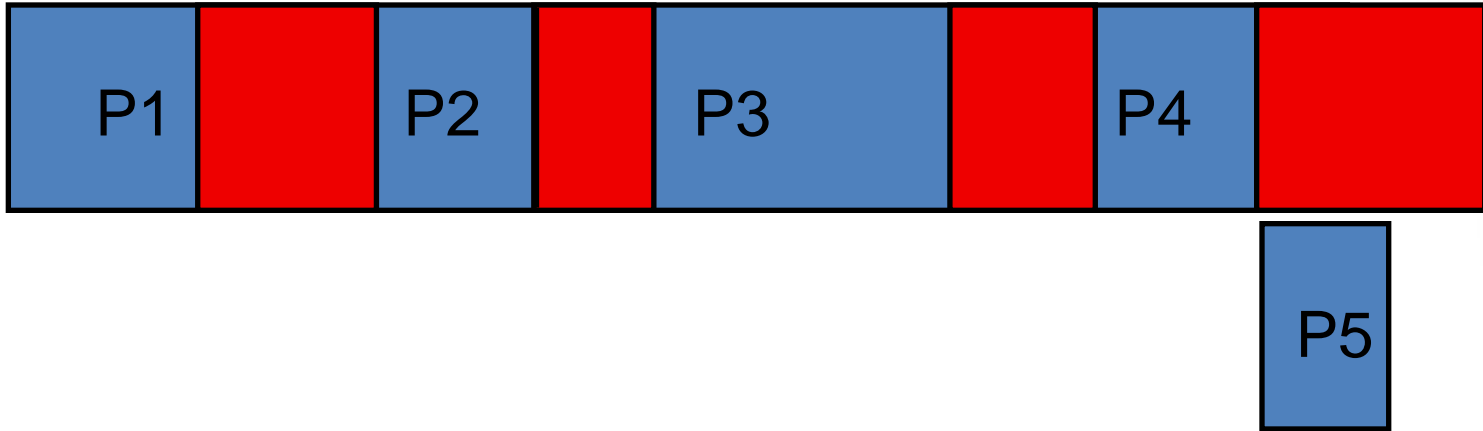
Best Fit



First Fit



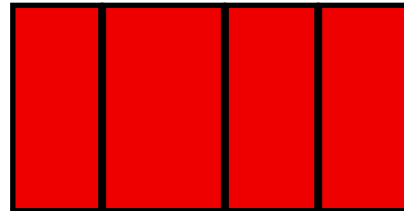
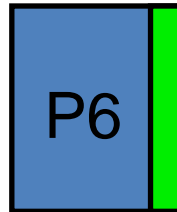
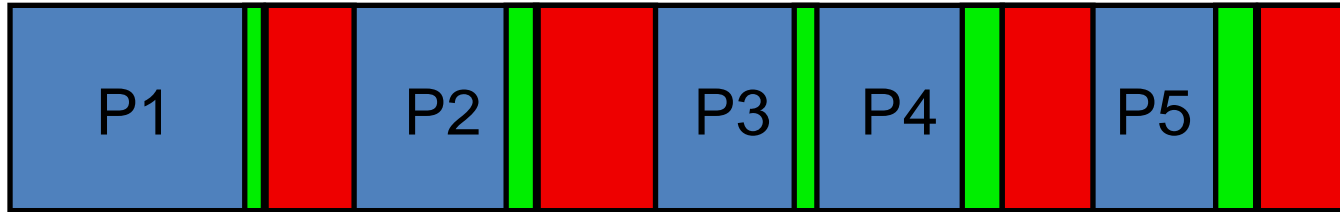
Worst Fit




Fragmentation

- Internal Fragmentation
 - ◇ if we allocate memory in units larger than a single byte (say 1K)
 - ◇ last block is only partially used
- External Fragmentation
 - ◇ lots of small holes spread throughout memory, none big enough to satisfy a request
 - ◇ worst fit tries to reduce this
 - ◇ compaction - move blocks (requires execution-time binding)
 - **50 percent rule** - N allocated blocks, $0.5 N$ lost to fragmentation (1/3 of memory unusable)

Fragmentation



 External
Fragmentation

 Internal
Fragmentation

Dynamic Loading

- Memory is always in short supply
- Not all routines are loaded when the program is loaded
 - ◇ only loaded when needed
 - ◇ some routines are rarely if ever used
 - ◇ does not require any special support from operating system
- Some execution environments do support dynamic loading (IBM mainframe, Java VM)
 - ◇ external programs are called by name, OS provides binding

Dynamic Linking

- *Static* linking is when all of the modules, including system libraries, are linked together at compile time.
- *Dynamic* linking provides stubs for each routine.
 - ◇ when the routine is called the first time, the routine is loaded
 - ◇ primarily used for shared libraries
 - libraries commonly used by many programs
e.g. strcpy, fopen, fclose.
 - allows updates and bug fixes without relinking
 - ◇ if libraries are to be shared between processes, then operating system must provide support (memory protection changes)

Overlays

- common on older systems (MS-DOS)
- no OS support required (although OS can get in the way)
- program is broken into multiple parts
- one common part of program always in memory
- other parts of program are replaced as needed
- common in early games for MS-DOS
 - different levels of the game might have different code parts, as each level is loaded, the code overlays the previous code
- also common in tools like compilers and assemblers
- complex details in overlays, not common today

Swapping

- Processes can be temporarily stored (*swapped*) from memory to a *backing store*
 - ◇ very fast hard drive - continuous store
- If memory binding is not execution time, then process must be swapped back into same place in memory
 - ◇ PDP-11 Unix used swapping to relocate and resize processes
- Make room for higher priority processes
- Major time is transfer time - amount of memory swapped.
- Used with some modifications on many systems