ELEC 377 – Operating Systems

Week 5 – Class 3

Today

- Deadlock
- ◊ Bankers Algorithm

System Model

- Resource Types $R_1, R_2, \ldots, R_n$
- ◊ Each resource has a number of instances (W_i)
- ◊ Resource instances are indistinguishable – doesn't matter which one you get.

- Process resource protocol
- ◊ request
- ◊ use
- ◊ release

Deadlock Conditions

- four conditions necessary for deadlock:

◊ **mutual exclusion**: only a limited number (usually one) process at a time can use a resource

◊ **hold and wait**: a process has (at least) one resource and is waiting for another

◊ **no preemption**: we can't take a resource away from a process

◊ **circular wait**: P_0 waits for a resource held by P_1, which waits for a resource held by P_2, … P_n, which waits for a resource held by P_0

Deadlock Basics

- No cycle –> no deadlock

- Cycle

◊ one instance per resource type –> deadlock

◊ more than one instance per resource type?
  - might be a deadlock
  - also might not be a deadlock!!

What do we do??

- Prevention
  - Ensure one of the 4 conditions never happens
- Avoidance:
  - Extra information before allocating an available resource
- Recovery:
  - Enter deadlock state and recover
- Ignore
  - Hope it never happens
  - Handle it manually
  - Most interactive operating systems use this approach

Prevention

- Mutual Exclusion
  - difficult, most resources are not shareable
  - spooler
- Hold and Wait
  - allocate all resources at once
  - inefficient
- Preemption
  - take resources away from other processes
  - rollback
- Circular Wait
  - always allocate resources in the same order
  – inefficient

Avoidance

- Information up front

◊ processes declare maximum resources needed

- Dynamically check current resource allocation to make sure cannot be a circular-wait condition

- *Resource allocation state*

◊ number of available and allocated resources and the *a priori* known maximum resources

Safe State

- system is safe if there is some order we can allocate the resources and not produce a deadlock
  - ◊ might not be the order that the processes actually request the resources
  - ◊ safe order means that someone may have to wait
- <P1, P2, …, Pn> is safe if Pi can satisfy the maximum resources with available (free) and the resources owned by previous processes.
  - ◊ P1 max must be satisfied only with free resources
  - ◊ P2 max must be satisfied with free + P1
  - ◊ P3 max gets available + P1 + P2
  - ◊ If not, wait until a previous process finishes.

Safe State

- Safe state - no deadlock
- unsafe - possibility of deadlock
- Stay in safe state

◊ easier to calculate than deadlock

Safe State – Examples

Process	Current	Max
P0	5	10
P1	2	4
P2	2	9

Total = 12, Free = 3

< P1 , P0 , P2 >

P1 (2) + 3 = 5 >= P1Max(4)

5 + P0(5) = 10 >= P0Max(10)

5 + P2(2) = 7 not >= P2Max(9)

10 + P2(2) = 12 >= P2Max(9)

Safe State

- As described, only works on one resource type
- Have to define the order for multiple resource types simultaneously.
- Find order of processes so that cascading sum holds in parallel for each resource type
  ◊ Bankers Algorithm
  - based on algorithm designed for banks to compute cash on hand

Bankers Algorithm

M types of resources
N processes

Available[M] = number of available resources
Max[N][M] = the max resources for each process
Allocation[N][M] = the currently allocated resources
Need[N][M] = the max resources that might be needed (Need[i][j] = Max [i][j] - Allocation[i][j]
Finished[N] = boolean flags to signal termination (initially all false)
Work[M] = working copy of Available

Bankers Algorithm

```
while (∃ i ∍ Finished[i] == false & Need[i] ≤ Work)
        // i is the next process in the safe sequence
        // add i's resources to pool
        Work = Work + Allocation[i]
        // i is in the safe sequence
        Finished [i] = true
        if (∀i, Finished[i] == true)!
        //all processes can complete?
                return safe
        else
                return unsafe
```

Bankers Algorithm

	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail	3 3 2
#Res	10 5 7

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	f
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	3 3 2
#Res	10 5 7
Work	3 3 2

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	f
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	3 3 2
#Res	10 5 7
Work	3 3 2

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	5 3 2	<P1

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			
Avail	3 3 2			
#Res	10 5 7			
Work	5 3 2	<P1		

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	7 4 3	<P1, P3

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	7 4 3	<P1, P3

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	t
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	7 4 5	<P1, P3, P4

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	t
Total Allocated	7 2 5			
Avail	3 3 2			
#Res	10 5 7			
Work	7 4 5	<P1, P3, P4		

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	t
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	t
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	10 4 7	<P1, P3, P4, P2

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	t
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	t
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	10 4 7	<P1, P3, P4, P2

Bankers Algorithm - Safety Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	t
P1	2 0 0	3 2 2	1 2 2	t
P2	3 0 2	9 0 2	6 0 0	t
P3	2 1 1	2 2 2	0 1 1	t
P4	0 0 2	4 3 3	4 3 1	t
Total Allocated	7 2 5			

Avail	3 3 2	
#Res	10 5 7	
Work	10 5 7	<P1, P3, P4, P2, P0> SAFE

Bankers Algorithm

- Allocation Algorithm

1 - compare request to available
  - not available, cannot allocate (sleep)

2 - compare request to max
  - violates max request, terminate process

3 - create temporary new state as if resource were allocated
  - do not allocate resources, just pretend to

4 - run safety algorithm on new state.
  - If not safe, put process to sleep
    until another process releases resources
  - if safe, allocate resources

Bankers Algorithm

	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail 3 3 2

#Res 10 5 7

P1 request = 1 0 2

Bankers Algorithm

	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	2 0 0	3 2 2	1 2 2
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail 3 3 2

#Res 10 5 7

P2 request = 1 0 2

enough resources?

Bankers Algorithm

	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail	2 3 0
#Res	10 5 7

P1 request = 1 0 2

Bankers Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	3 0 2	3 2 2	0 2 0	f
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail	2 3 0
#Res	10 5 7
Work	2 3 0

Bankers Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	3 0 2	3 2 2	0 2 0	f
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail 2 3 0

#Res 10 5 7

Work 2 3 0 <P1

Bankers Algorithm

	Allocation	Max	Need	Finished
P0	0 1 0	7 5 3	7 4 3	f
P1	3 0 2	3 2 2	0 2 0	f
P2	3 0 2	9 0 2	6 0 0	f
P3	2 1 1	2 2 2	0 1 1	f
P4	0 0 2	4 3 3	4 3 1	f
Total Allocated	7 2 5			

Avail 2 3 0

#Res 10 5 7

Work 10 5 7 <P1, P3, P4, P2, P0>

SAFE

Bankers Algorithm

	Allocation	Max	Need
P0	0 1 0	7 5 3	7 4 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail	2 3 0
#Res	10 5 7

P0 request = 0 2 0

Bankers Algorithm

	Allocation	Max	Need
P0	0 3 0	7 5 3	7 2 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail	2 1 0
#Res	10 5 7

P0 request = 0 2 0

Bankers Algorithm

	Allocation	Max	Need
P0	0 3 0	7 5 3	7 2 3
P1	3 0 2	3 2 2	0 2 0
P2	3 0 2	9 0 2	6 0 0
P3	2 1 1	2 2 2	0 1 1
P4	0 0 2	4 3 3	4 3 1
Total Allocated	7 2 5		

Avail	2 1 0	
#Res	10 5 7	
Work	2 1 0	- no row in Need matches
		* UNSAFE*

Deadlock Detection

- allow system to deadlock

- run a detection algorithm Occasionally
  - ◊ maintain “waitfor” graph (already have it)
  - ◊ look for cycles
  - ◊ expensive - O(n x m x m)

- recovery scheme

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 <- L
add AL,P        AL <- AL + P
…
.data
.org 9af
L:  .byte 12
P:  .byte 42
```

Binding Instructions and Data

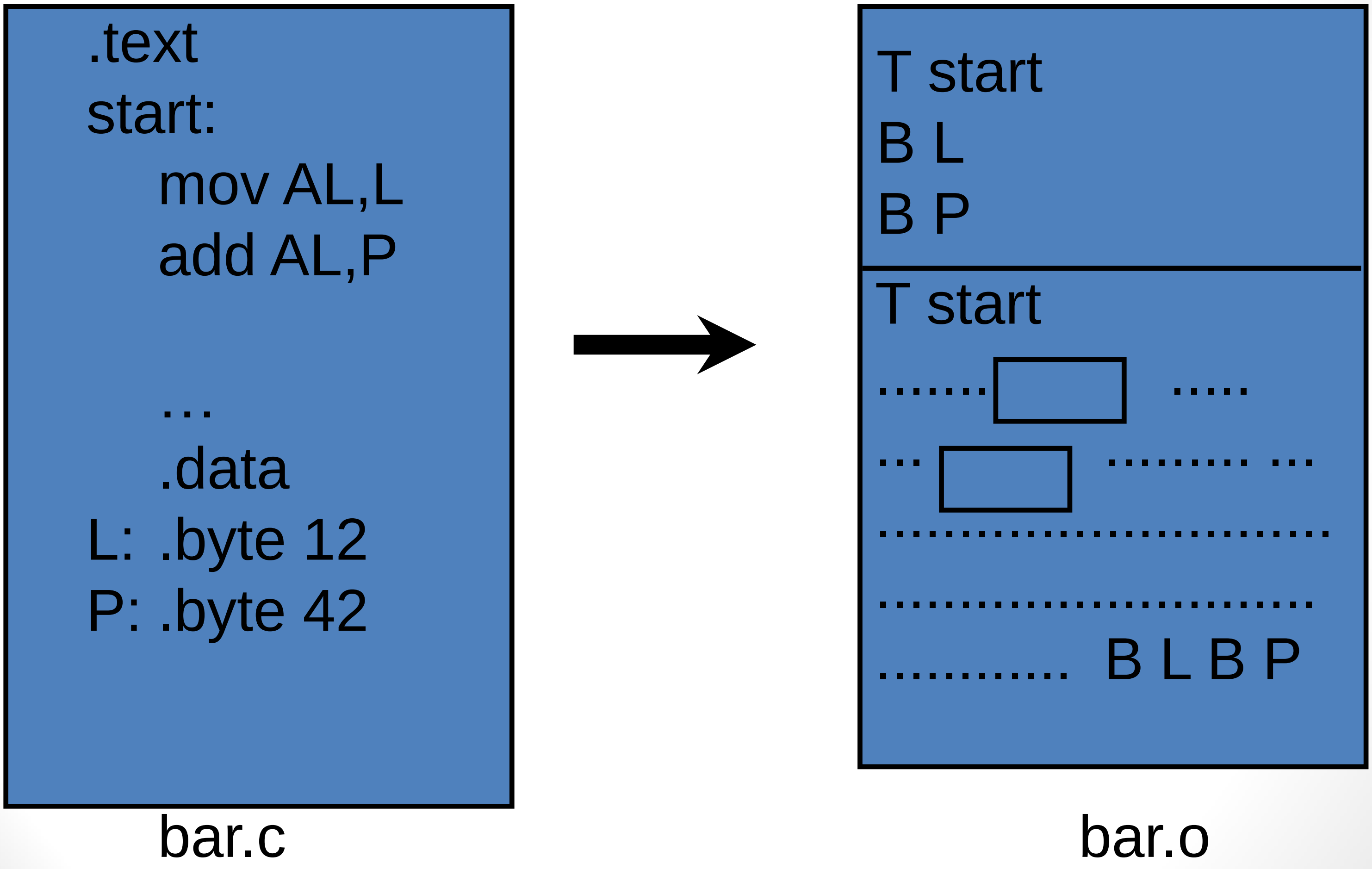
- Load Time

◊ header of load file contains locations of referneces

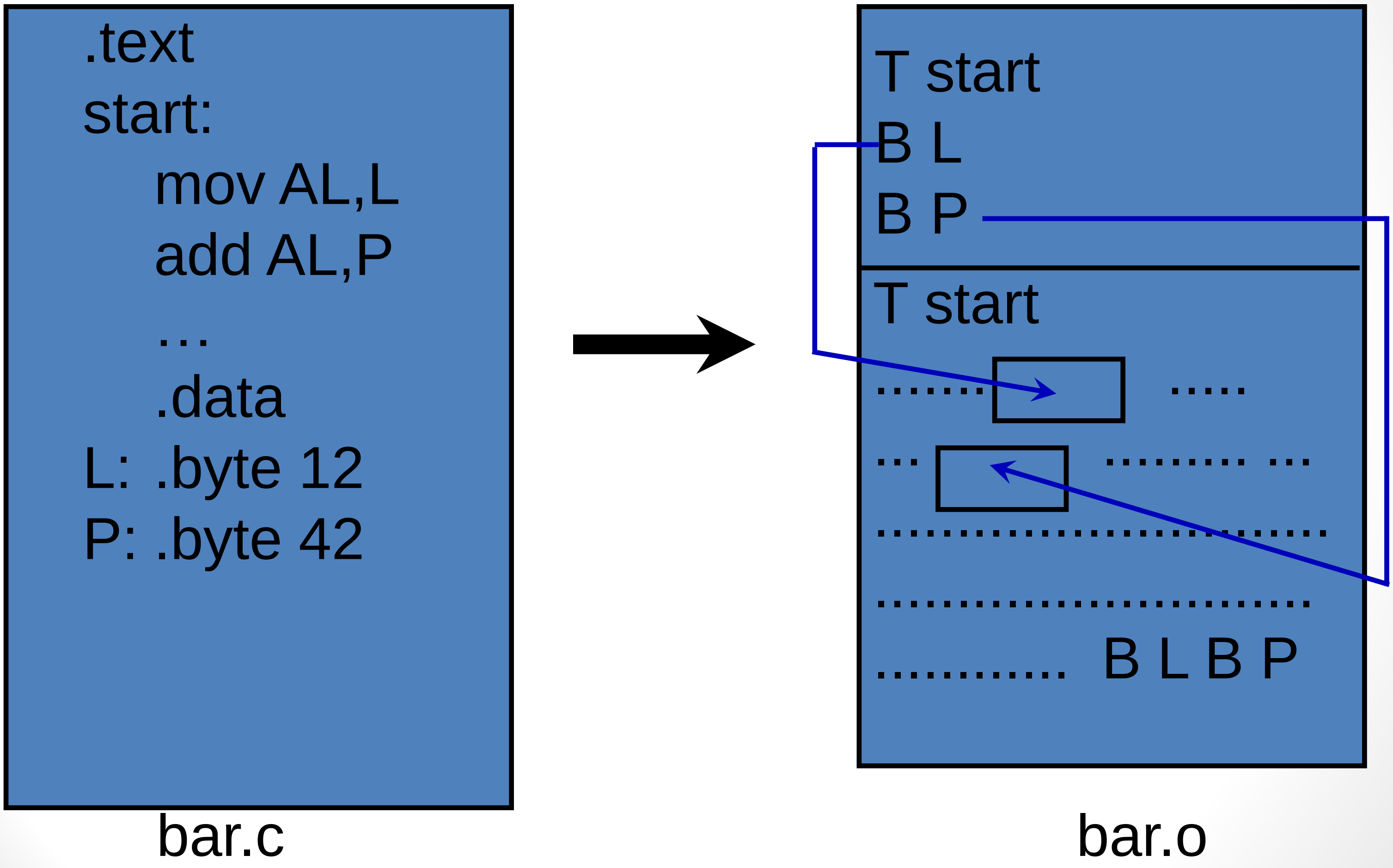
◊ loader (Part of OS) resolves them at the time they are read into memory

```
    .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



Load Time Module



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

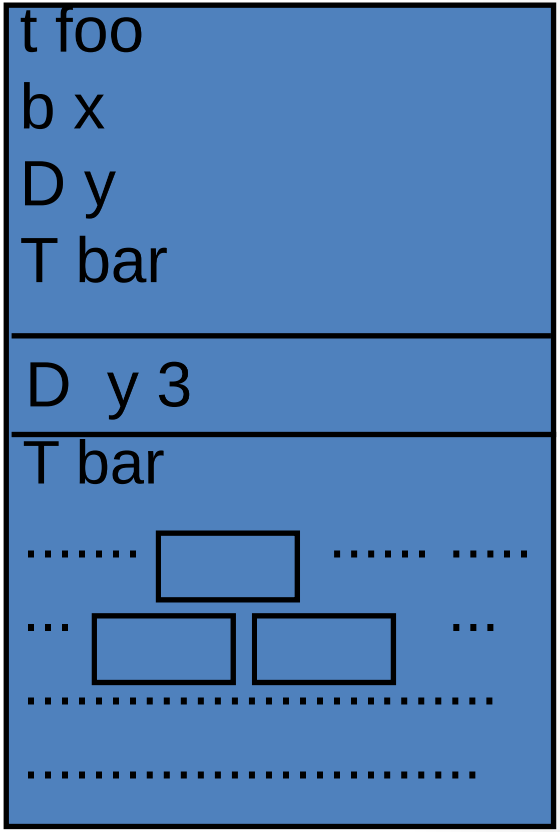
- Execution Time

◊ Similar to Compile Time, but hardware looks after translation

Compile

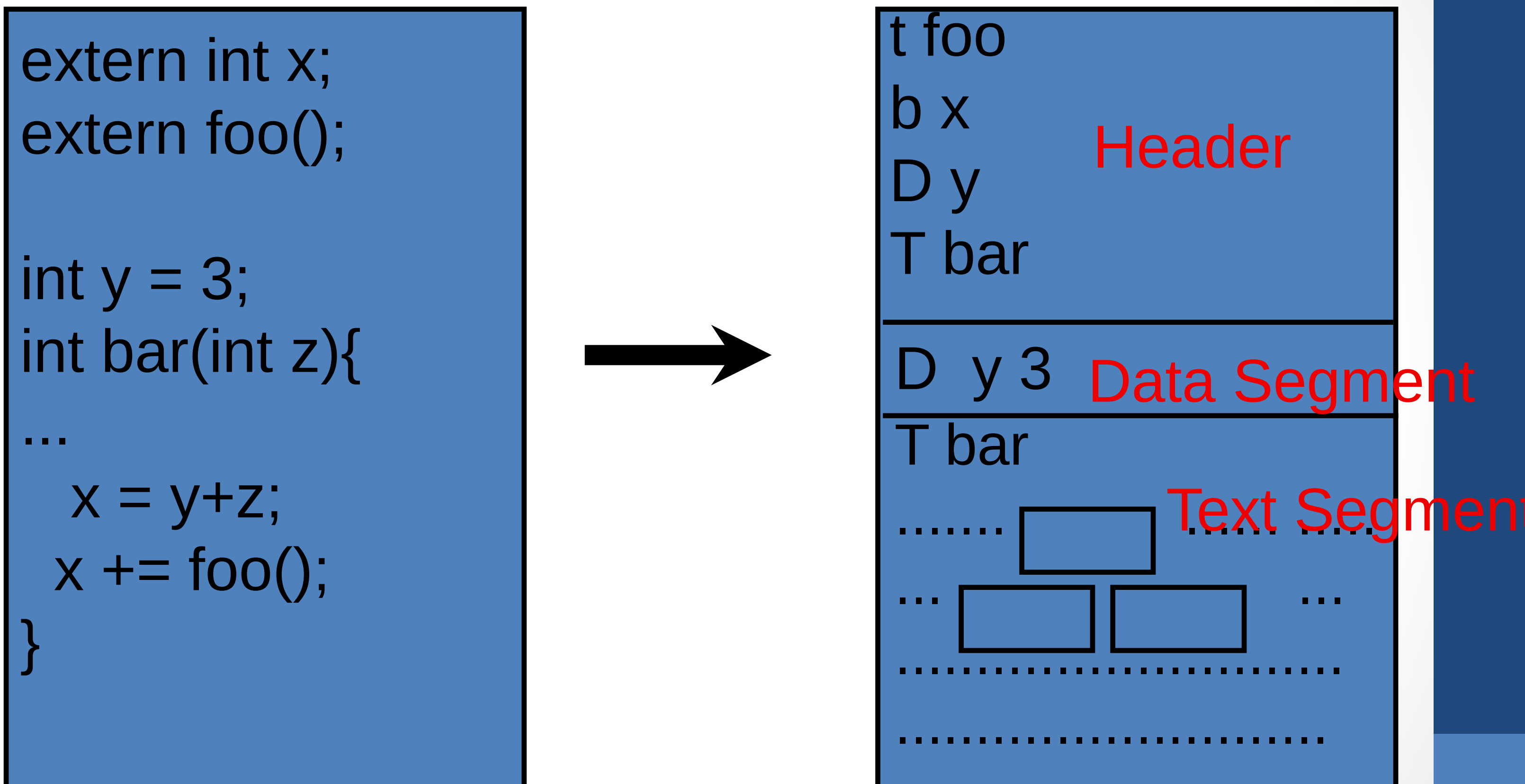
```
extern int x;
extern foo();

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

bar.c



bar.o

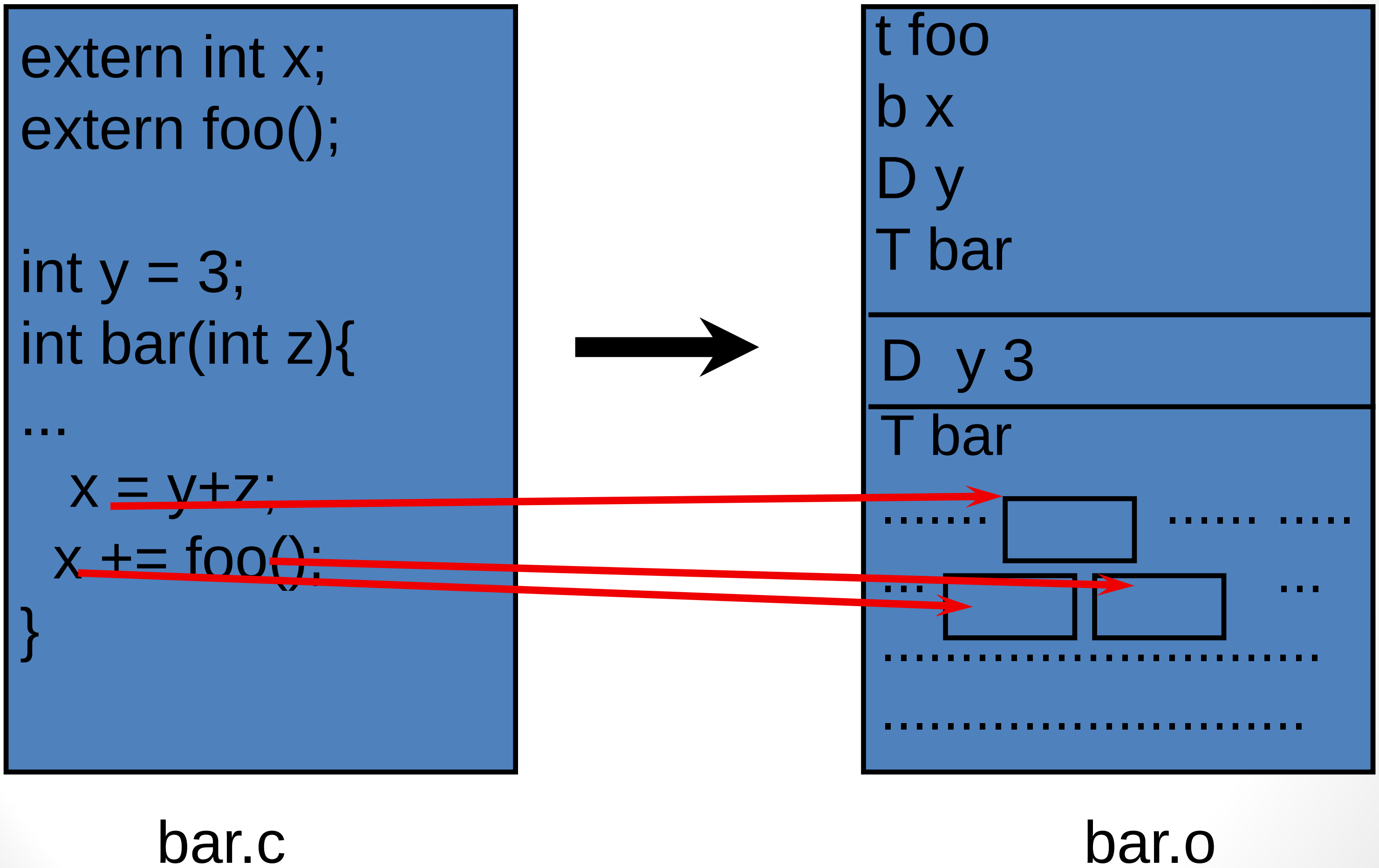
Compile



bar.c

bar.o

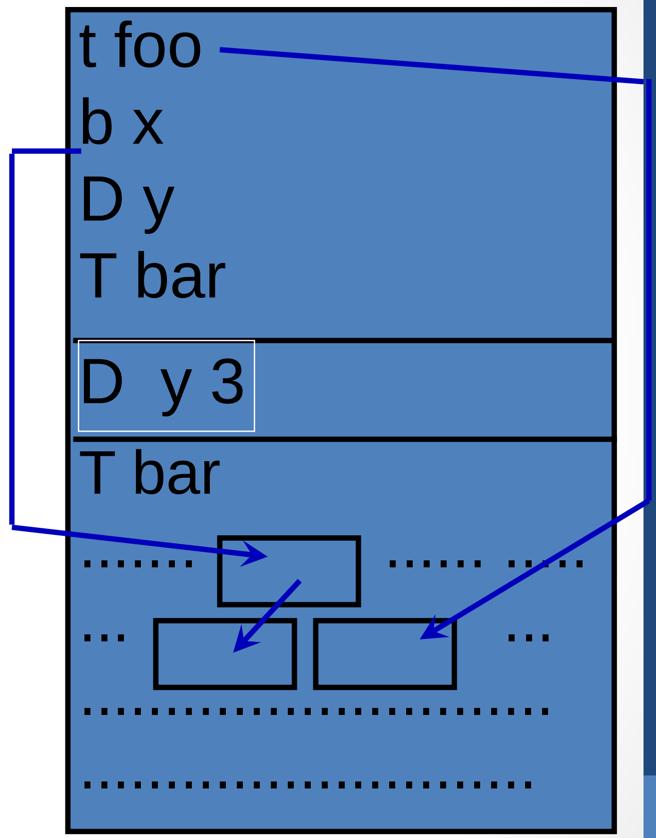
Compile


extern int x;
extern foo();
int y = 3;
int bar(int z){
...
x = y+z;
x += foo();
}
t foo
b x
D y
T bar
D y 3
T bar
...
bar.c
bar.o

Compile

```
extern int x;
extern foo();

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

bar.c

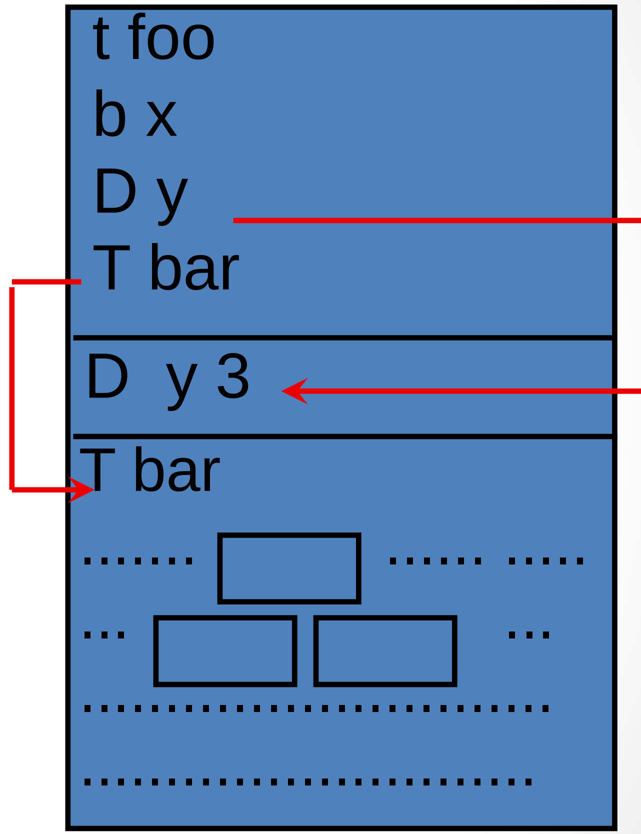


bar.o

Compile

```
extern int x;
extern foo();

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

bar.c



bar.o

Linking

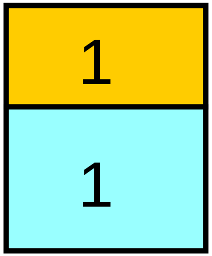
```
...
int x;
extern int y;
FILE * f;
int main(int argc, char * argv[]){
   f= fopen(“/proc/lab3”,”r”);
   fprintf(stderr,”foobar %d”,x+bar(y));

   ...

}
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(){...}
```

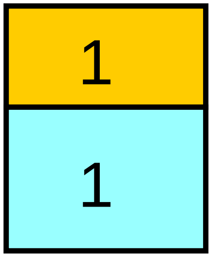
- Defined
- Undefined bar.o
- Undefined library

Linking–combine to single executable

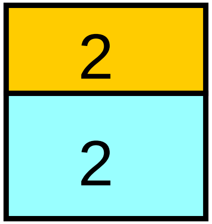


foo.o
def main,x
undef fopen,bar

Linking–combine to single executable

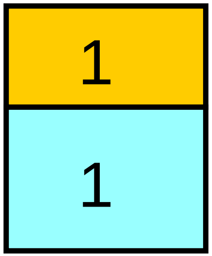


foo.o
def main,x
undef fopen,bar

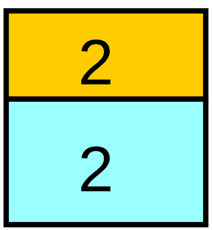


bar.o
def bar,y
undef foo,x

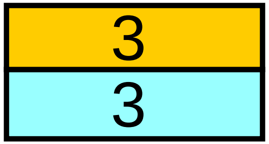
Linking–combine to single executable



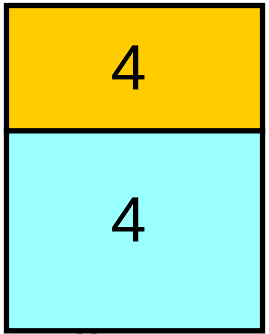
foo.o
def main,x
undef fopen,bar



bar.o
def bar,y
undef foo,x

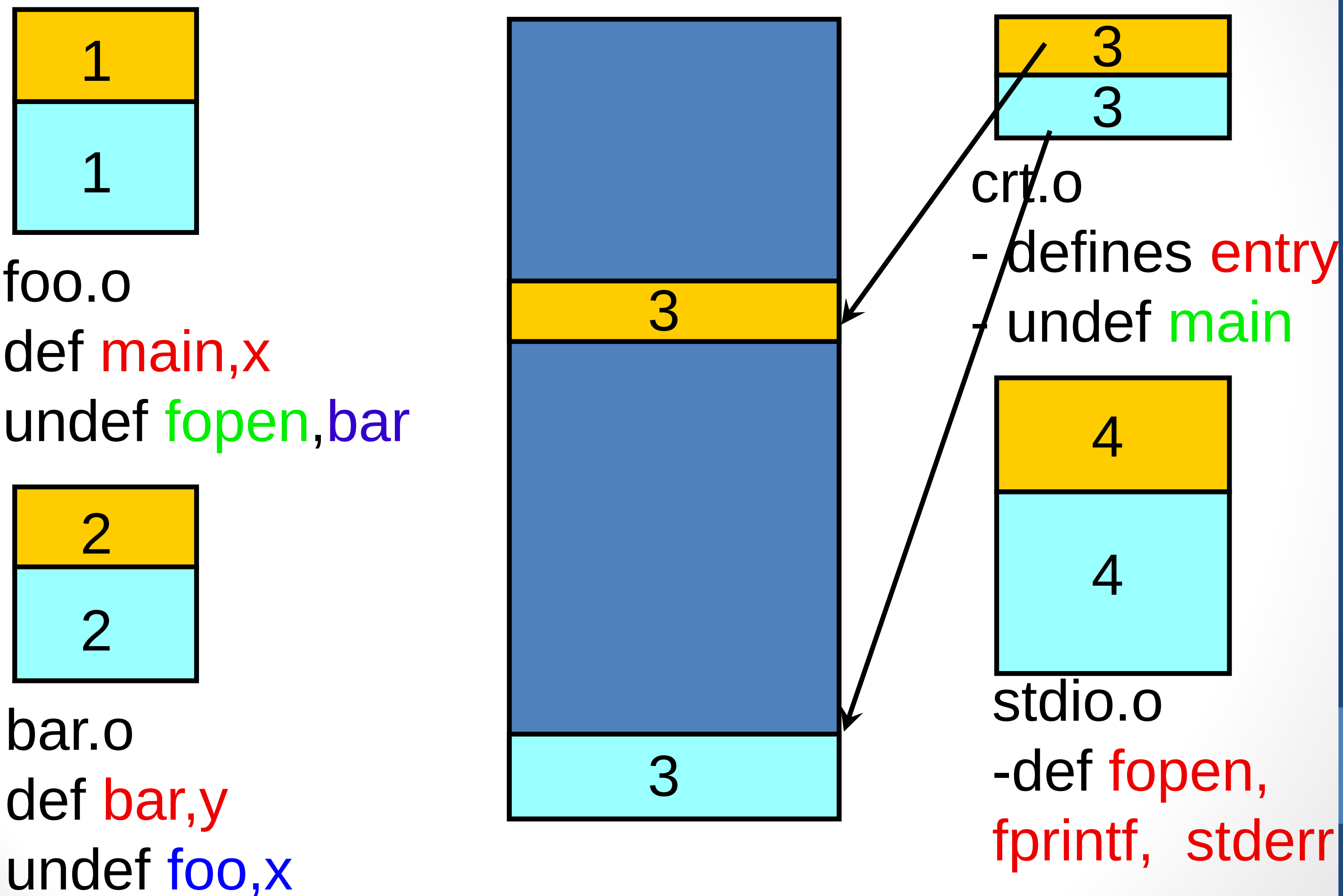


crt.o
- defines entry
- undef main

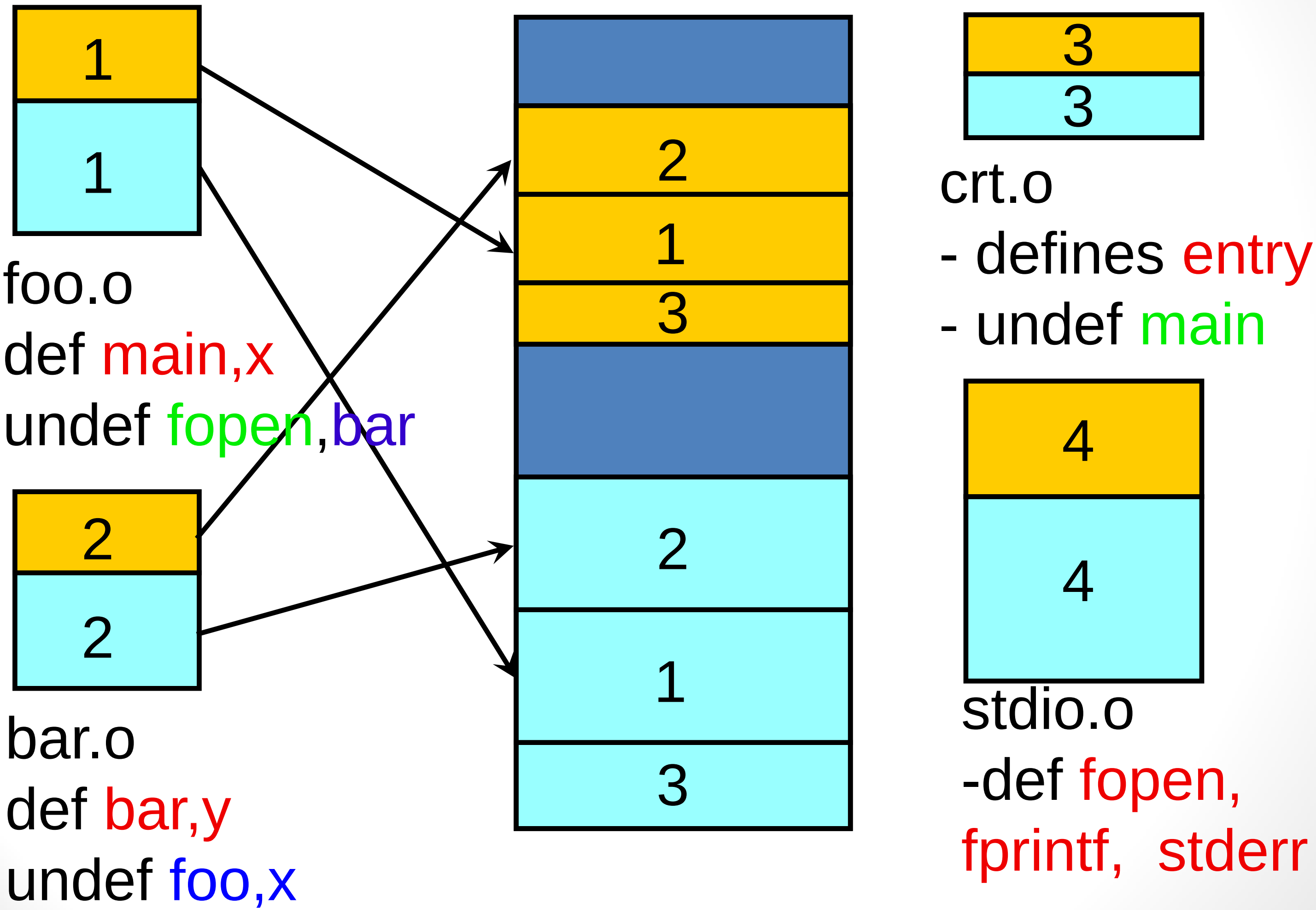


stdio.o
-def fopen,
fprintf, stderr

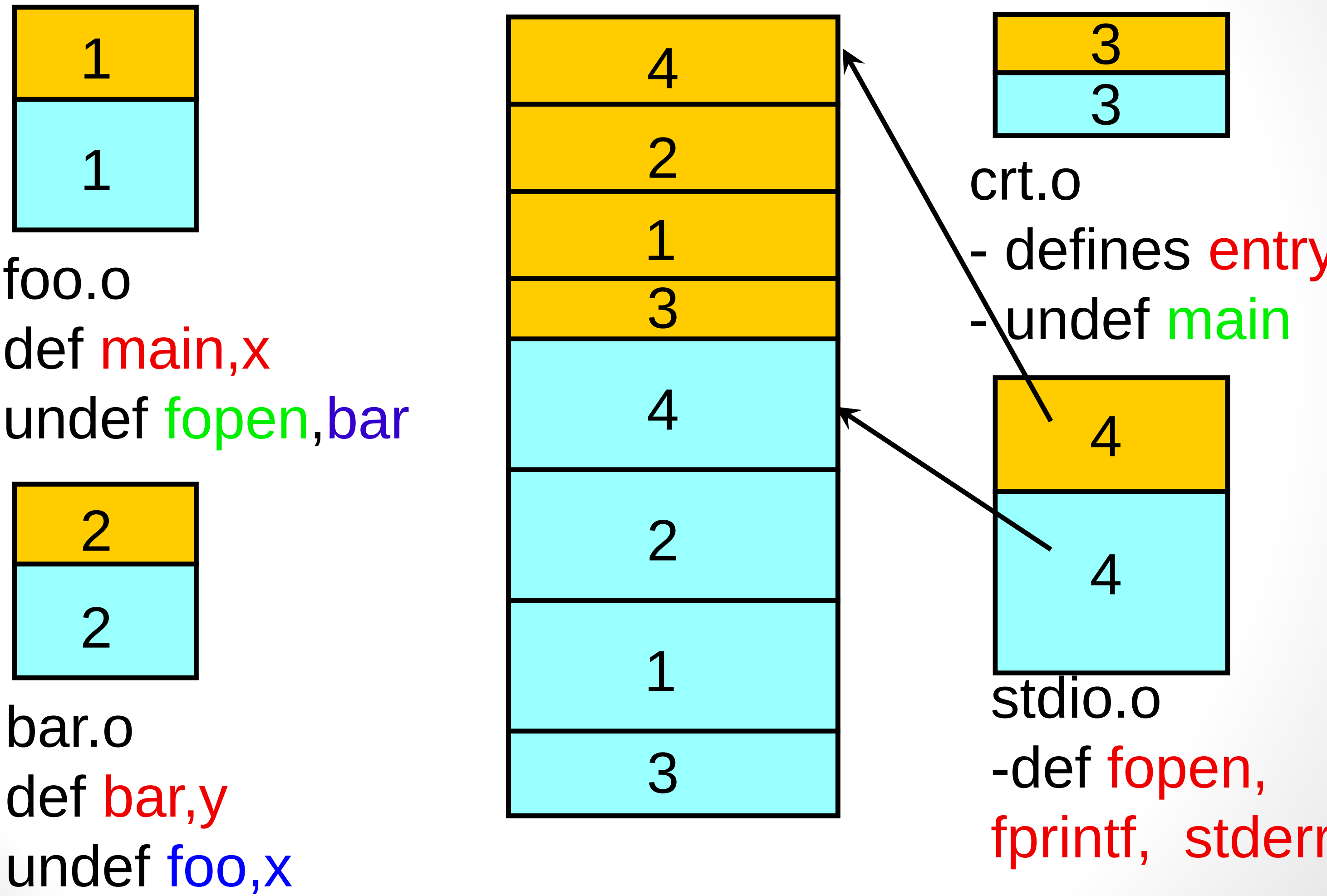
Linking–combine to single executable



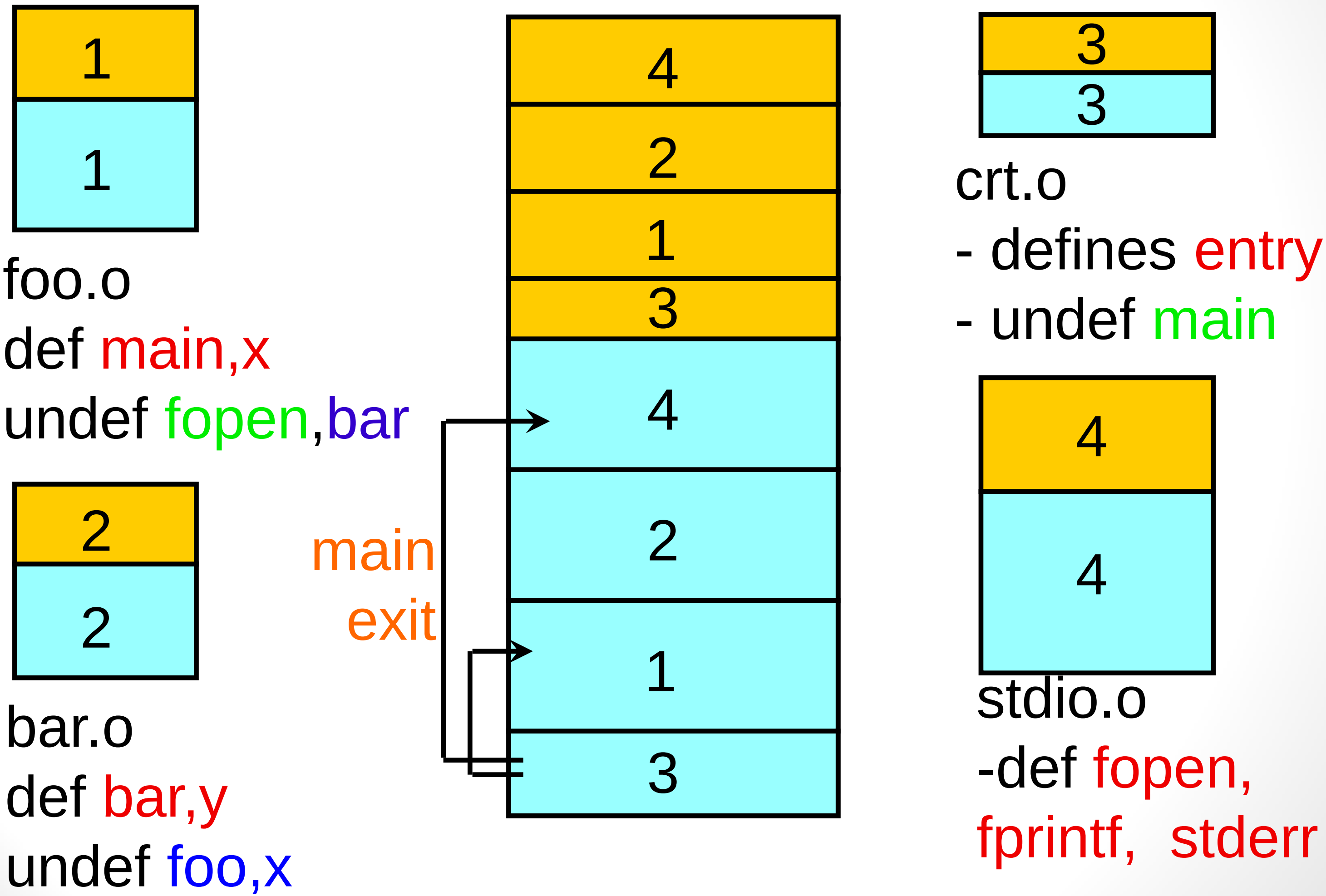
Linking–combine to single executable



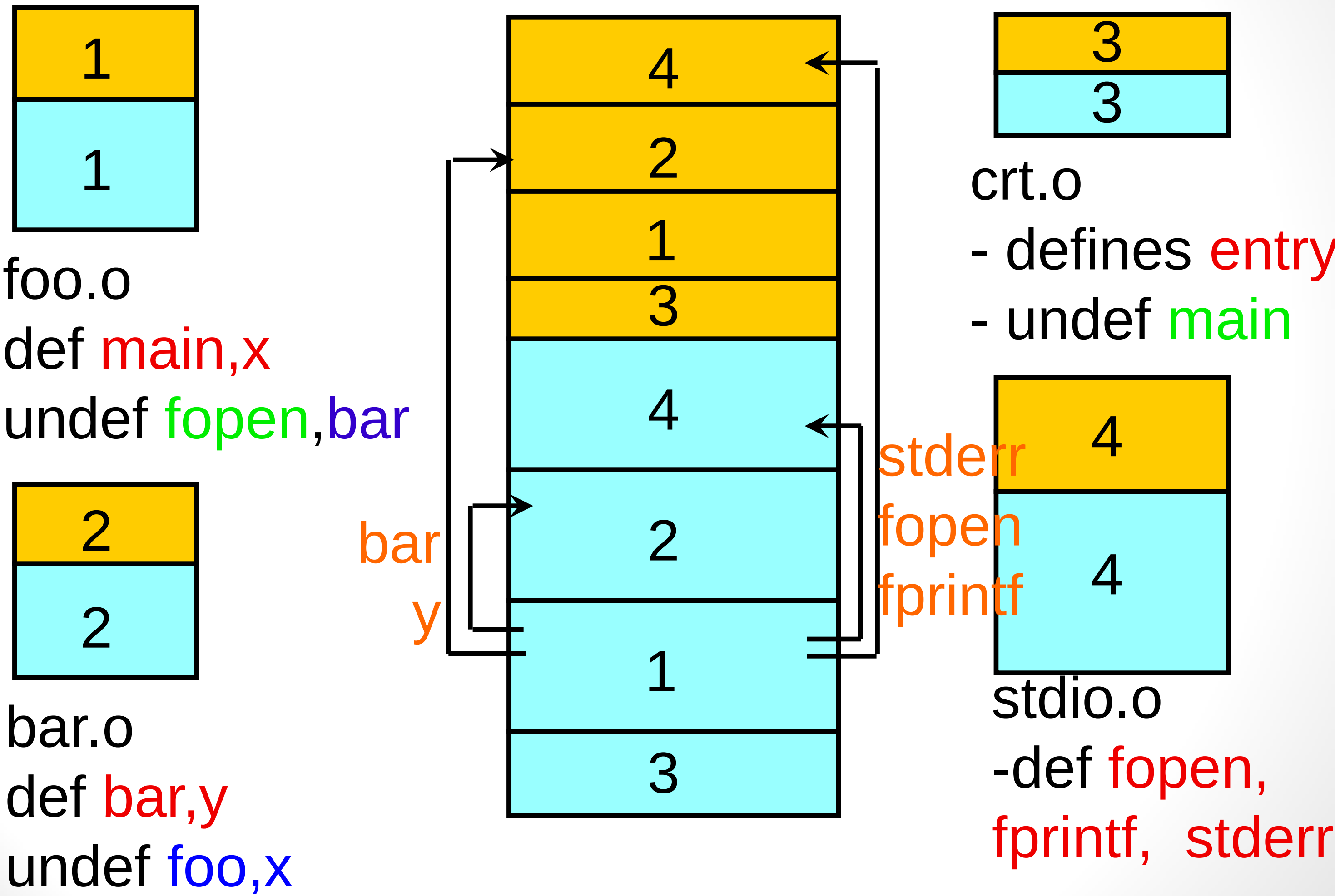
Linking–combine to single executable



Linking–combine to single executable



Linking–combine to single executable



Linking–combine to single executable