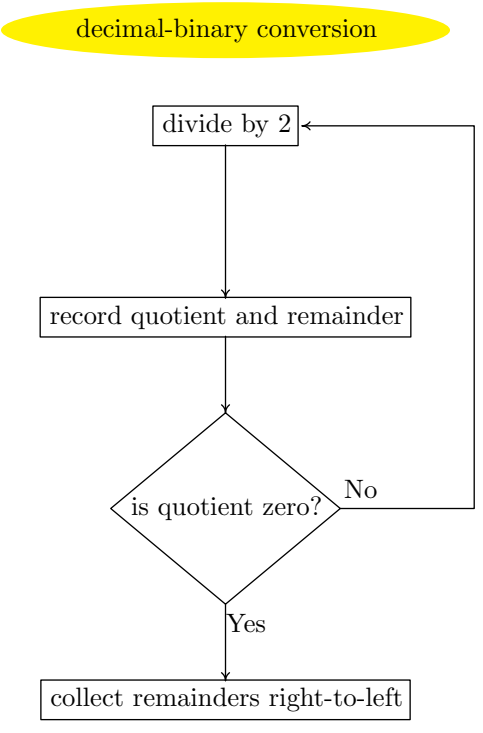
1 Algorithm



Examples

Division	Quotient	Remainder
95/2	47	1
47/2	23	1
23/2	11	1
11/2	5	1
5/2	2	1
2/2	1	0
1/2	0	1

Collect remainders from bottom to top
$\Rightarrow 95_d = 1011111_b$

Division	Quotient	Remainder
10/2	5	0
5/2	2	1
2/2	1	0
1/2	0	1

Collect remainders from bottom to top
$\Rightarrow 10_d = 1010_b$

2 SML

Implementing this algorithm in SML. If n is zero, return an empty list of digits, otherwise divide by 2 and append the remainder to the list

SML

```
PolyML.print_depth 100;
(* fun dec_to_bin 0 = []
   |dec_to_bin n = dec_to_bin(n div 2) @ [n mod 2]; *)
fun cons (x,lst) = (op ::)(x,lst);
(* fun dec_to_bin 0 = []
   |dec_to_bin n = dec_to_bin(op div)(n,2) @ cons(op mod)(n,2), []); *)
fun append (lst1, lst2) = (op @)(lst1, lst2);
fun dec_to_bin 0 = []
   |dec_to_bin n =
      append(dec_to_bin( (op div)(n,2) ), cons((op mod)(n,2), []));
fun lst_to_str (0::xs) = "0" ^ lst_to_str xs
   |lst_to_str (1::xs) = "1" ^ lst_to_str xs
   |lst_to_str _ = "";
lst_to_str(dec_to_bin 95); (* expect "1011111" *)
dec_to_bin 10;
dec_to_bin 4096;
lst_to_str(dec_to_bin 5000000000);
```

```
val it = (): unit
> # val dec_to_bin = fn: int -> int list
> # # val lst_to_str = fn: int list -> string
> val it = "1011111": string
# val it = [1, 0, 1, 0]: int list
> val it = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]: int list
> val it = "1001010100000010111110010000000000": string
>
```

3 C implementation

To implement this in C, we need to create a structure to represent each digit and a pointer to the next structure node. Every time we need to add another digit, we allocate memory for a new node and fill with the value and set the pointer at end of current list to point to this node.

C source code written to file lab4.c

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list_struct
{
  int list_head;
  struct list_struct * list_tail;
} * int_list;

int_list cons(int, int_list);
int_list cons2(int, int_list);
void print_list(int_list);
int_list append(int_list, int_list);
int_list append2(int_list, int_list);
int_list dec_to_bin(int n);
int_list dec_to_bin2(int n);
int_list EAX;
```

The main function is used to test the algorithm by using a few values such as 95, 10, and 4096.

C source code appended to file lab4.c

```
int main()
{
 /*
  int_list intLst1 = cons(4,cons(3,cons(2,cons(1,NULL))));
  int_list intLst2 = cons(8,cons(6,cons(7,cons(5,NULL))));
  print_list(append(intLst1,intLst2));

  append2(intLst1, intLst2);
  print_list(EAX);
  print_list(dec_to_bin(95));
  */
  dec_to_bin2(95); //assumes EAX global variable has result
  print_list(EAX);

  dec_to_bin2(10); //assumes EAX global variable has result
  print_list(EAX);

  dec_to_bin2(4096); //assumes EAX global variable has result
  print_list(EAX);
}
```

Convert decimal to binary using append and cons.

C source code appended to file lab4.c

```
int_list dec_to_bin(int n)
{
  if(n ==0) return NULL;//empty list
  else return append(dec_to_bin(n/2), cons(n%2,NULL));
}
```

This version of dec_to_bin does not use function composition so it will be a better model for our conversion to asm code.

C source code appended to file lab4.c

```
int_list dec_to_bin2(int n){
  int_list EBX;
  int quotient, remainder;
  if(n ==0) EAX = NULL;//empty list
  else
  { //quotient = n/2; //or shift right one bit
    quotient = n >> 1;
    dec_to_bin2(quotient);
    EBX = EAX;
    remainder = n%2; // or shift right and check carry flag
    cons2(remainder, NULL); //returns result in EAX
    append2(EBX, EAX);      //returns result in EAX
  }
}
```

Building up a list requires dynamic memory allocation. Let's not worry about freeing that memory (yet). We can create the memory using malloc and put that new element in an existing list.

C source code appended to file lab4.c

```
int_list cons(int i, int_list p)
{
  int_list lst = malloc( sizeof(struct list_struct));
  lst->list_head = i;
  lst->list_tail = p;
  return lst;
}
```

Here is a version of cons that does not return a value, but puts result in EAX

C source code appended to file lab4.c

```
int_list cons2(int i, int_list p)
{
  int_list lst = malloc (sizeof(struct list_struct));
  lst->list_head = i;
  lst->list_tail = p;
  EAX = lst;
}
```

To print out a linked list we must loop through all elements until we detect the last one (that is NULL). We can't use a "for" loop since we don't know how many are in the list.

C source code appended to file lab4.c

```
void print_list(int_list lst)
{
  if(lst != NULL)
  {
    printf("%i",lst->list_head);
    print_list(lst->list_tail);
  }
  else
    printf("\n");
}
```

To append one list to another, use cons and recursively append each element of first list to tail of second list.

C source code appended to file lab4.c

```
int_list append(int_list lst1, int_list lst2)
{
  if (lst1 != NULL)
  {
     return cons(lst1->list_head,append(lst1->list_tail, lst2));
  }
  else
    return lst2;
}
```

Here is a version of append tht is more like asm code. i.e. return is to global (register) variable, and there is no function composition, so we need to call append recursively before calling cons.

C source code appended to file lab4.c

```
int_list append2(int_list lst1, int_list lst2)
{
  if (lst1 != NULL)
  {
    append2(lst1->list_tail, lst2);
    EAX = cons(lst1->list_head,EAX);
  }
  else
    EAX = lst2;
}
```

```
debian@debian:~/labs/lab4$ ./lab4c
1011111
1010
1000000000000
```

4 ASM implementation

asm source code written to file lab4.s

```
.data
  .equ NODESIZE,8
  .lcomm int_list,NODESIZE
```

4.1 main function

This is main function where program start, it will push 2, 95, 10, 4096 to pass parameter to and call function dec_to_bin. We also need call printf to print out the results.

asm source code appended to file lab4.s

```
.text
.globl _start
_start:
```

Convert 2 decimal to binary

asm source code appended to file lab4.s

```
  push $2
  call dec_to_bin
#expect result in EAX
  add $4, %esp
  push %eax
  call print_list
  add $4, %esp
```

Convert 95 decimal to binary

asm source code appended to file lab4.s

```
  push $95
  call dec_to_bin
  add $4, %esp
  push %eax
  call print_list
  add $4, %esp
```

Convert 10 decimal to binary

asm source code appended to file lab4.s

```
push $10
  call dec_to_bin
  add $4, %esp
  push %eax
  call print_list
  add $4, %esp
```

Convert 4096 decimal to binary

asm source code appended to file lab4.s

```
push $4096
  call dec_to_bin
  add $4, %esp
  push %eax
  call print_list
  add $4, %esp
  mov $1, %eax
  mov $0, %ebx
  int $0x80
```

```
debian@debian:~/labs/lab4$ casm lab4
debian@debian:~/labs/lab4$ ./lab4asm
10
1011111
1010
1000000000000
```

4.2 cons function

Here is the cons function translated from function con2 of C code. Since we need to use the register EBX for easy copy data, we use push and pop EBX in function to save old value stored in EBX.

asm source code appended to file lab4.s

```
cons:
  push %ebp
  mov %esp, %ebp
  push $8
  call malloc
  add $4, %esp              # expect address in eax
```

call malloc function from C library to create 8-bytes size block and return eax point to address of that block. I think we can choose whatever value for size of malloc, as long as eax points to new location generated by malloc function. We choose 8 because it represents that each node have 2 parts: head and tail.

asm source code appended to file lab4.s

```
  push %ebx
  mov 12(%ebp), %ebx  #i
  mov %ebx, (%eax)    #head
```

indirect addressing mode, which means copy the value (i) from EBX to the location in memory where EAX point to

asm source code appended to file lab4.s

```
  mov 8(%ebp), %ebx   #p
  mov %ebx, 4(%eax)   #tail
  pop %ebx
  mov %ebp, %esp
  pop %ebp
  ret
```

4.3 append function

To append to 2 list together. The first node of new list is the first node of list 1 and end node of new list is the end node of list 2.

asm source code appended to file lab4.s

```
append:
 push %ebp
 mov %esp, %ebp
 push %ebx
 mov 12(%ebp), %ebx   # ebx = address of lst1
 cmp $0, %ebx         # lst1 == NULL
 jne if_append
```

If lst1 (first parameter) is equal to NULL,

asm source code appended to file lab4.s

```
 mov 8(%ebp), %ebx
 mov %ebx, %eax
 jmp end_append
```

If lst1 (first parameter) is not equal to NULL, call append function again with first parameter is the address that is 4 bytes adding to the address of lst1, and second parameter still the same (lst2).

asm source code appended to file lab4.s

```
if_append:
 push 4(%ebx)         # lst1->tail
 push 8(%ebp)         # lst2
 call append
 add $8, %esp
```

After call append, call cons to connect new list to where EAX point to, with first parameter is lst1 → head and second parameter is EAX, or actually the location where EAX points to.

asm source code appended to file lab4.s

```
 push (%ebx)
 push %eax
 call cons
 add $8, %esp
end_append:
 pop %ebx
 mov %ebp, %esp
 pop %ebp
 ret
```

4.4 dec_to_bin function

dec_to_bin function implementation. Since we need 2 local variable, quotient and remainder, we will sub 8 from ESP to create rooms for them.

asm source code appended to file lab4.s

```
dec_to_bin:
  push %ebp
  mov %esp, %ebp
  sub $8,%esp             # room for 2 local variables

  cmp $0, 8(%ebp)         # n == 0
  jne else_part
```

If n equals 0, set EAX equal to 0, which mean NULL (or empty list)

asm source code appended to file lab4.s

```
  mov $0, %eax            # empty list
  jmp end_func
```

If n is not equal to 0,we copy value n to EAX, then copy value in EAX to first local variable, which is for quotient. The reason we cannot copy value n directly to first local variable is because mov command can't take too many memory references. Then, shift right 1 bit the value stored in first local variable (quotient), which is the same as dividing it by 2. Then push quotient to stack for calling dec_to_bin.

asm source code appended to file lab4.s

```
else_part:
  mov 8(%ebp),%eax
  mov %eax,-4(%ebp)  # copying n to quotient
  shrl -4(%ebp)      # divide quotient by 2
  push -4(%ebp)
  call dec_to_bin
  add $4, %esp
  push %ebx
  mov %eax, %ebx     # EBX = EAX
```

Push and pop EBX to save old EBX value after we done calling function. The second local variable is remainder from dividing n by 2. We can shift right 1 bit and check carry flag (CF). Since we already do shr before, it is not right to do shr again to check CF. Hence, we have to replace local varibale quotient by the number before we did shr, and do shr again to check CF. Use command jnc to check whether CF is 0 or not. If CF = 0, then remainder = n

asm source code appended to file lab4.s

```
  movl $0, -8(%ebp)      # remainder =0
  mov 8(%ebp),%ecx
  mov %ecx,-4(%ebp)      # copying n to quotient
  shrl -4(%ebp)          # divide quotient by 2
  jnc call_cons
  movl $1, -8(%ebp)      # remainder = 1
```

Calling cons function with the first parameter is the remainder, second paramter is NULL

asm source code appended to file lab4.s

```
call_cons:
  push -8(%ebp)              # remainder
  push $0                    # NULL
  call cons                  # expect result in EAX
  add $8, %esp
```

call append with 2 parameter EBX and EAX to connect nodes.

asm source code appended to file lab4.s

```
  push %ebx
  push %eax
  call append
  add $8, %esp               # expect result in EAX
end_func:
  pop %ebx
  mov %ebp, %esp
  pop %ebp
  ret
```

4.5 Print_list function

Here is the implementation for print_list function. fmt string is needed for printing out head value of each node, and fmt2 string for ending print_list function calling.

asm source code appended to file lab4.s

```
.data
  fmt: .string "%i"
  fmt2: .string "\n"
.text
print_list:
  push %ebp
  mov %esp, %ebp
  push %ebx
  mov 8(%ebp), %ebx              # parameter
  cmp $0, %ebx
  je else_print
```

If the list is NULL, print nothing; or else, print out first node's head of the list.
push (%ebx) means we want to push the value store at location where EBX points to, not the value stored in EBX itself (remember that the value stored in EBX is the address of the first node of the list where EAX points to).

asm source code appended to file lab4.s

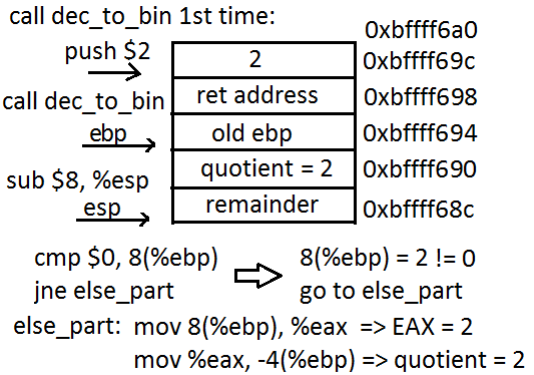
```
  push (%ebx)
  push $fmt
  call printf
  add $8, %esp
```

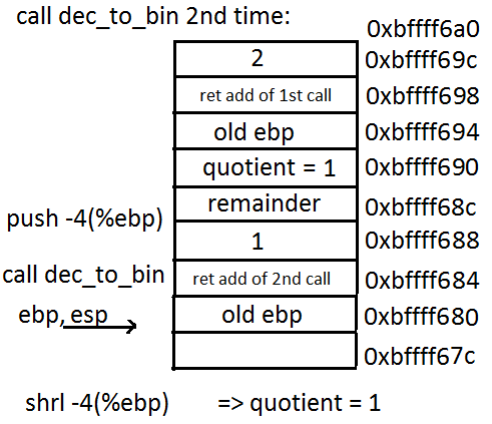
Then calling function print_list again with the parameter is the address of the head of second node (or tail of first node).

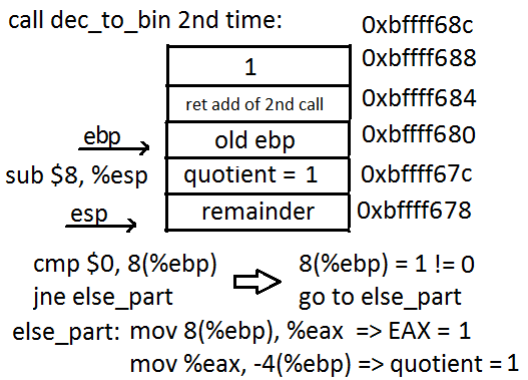
asm source code appended to file lab4.s

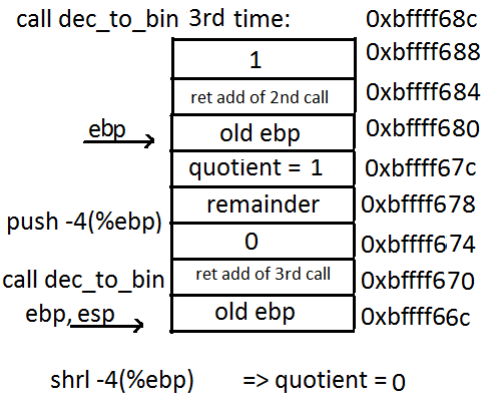
```
  push 4(%ebx)
  call print_list
  add $4, %esp
  jmp end_print
# print out "\n" if reach the end of the list
else_print:
  push $fmt2
  call printf
  add $4, %esp
end_print:
  pop %ebx
  mov %ebp, %esp
  pop %ebp
  ret
```

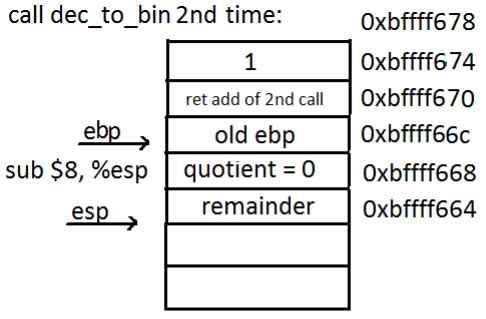
5 Stack Frame



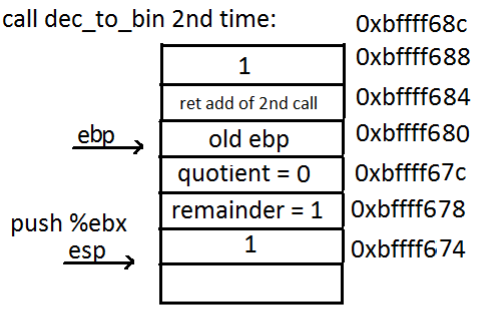




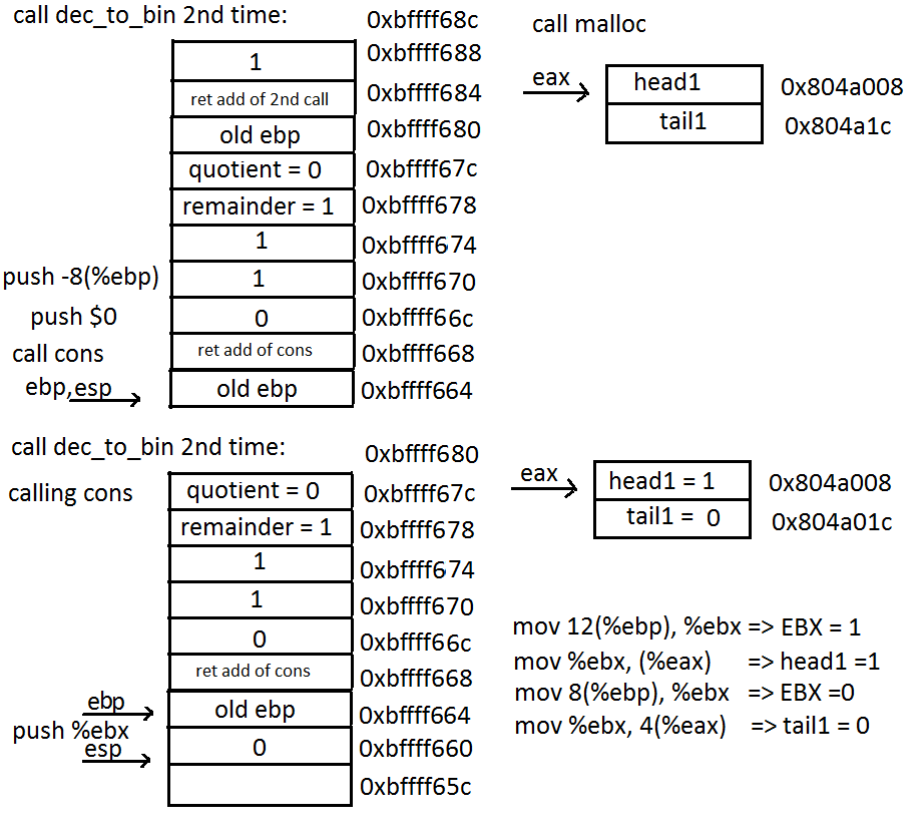


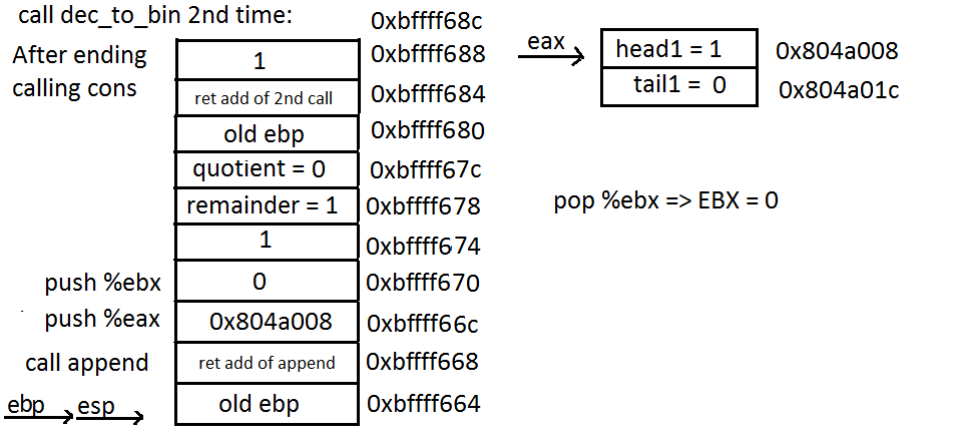


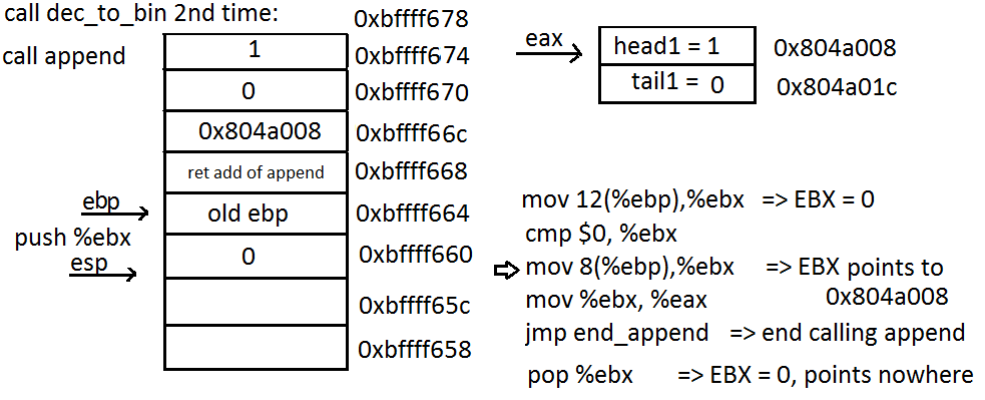

```
cmp $0, 8(%ebp)     #   8(%ebp) = 0
mov $0, %eax        => EAX = 0
jmp end_func        => end 3rd call of
                       dec_to_bin

pop %ebx            => EBX = 1
```

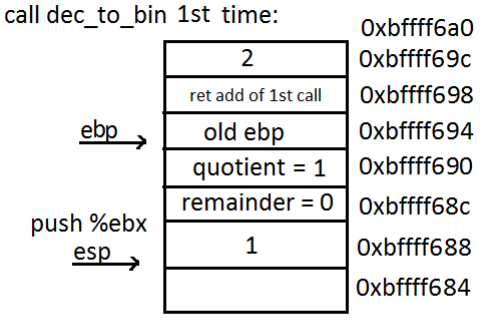


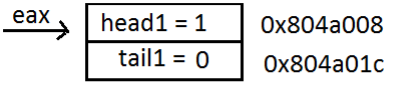

```
mov %eax, %ebx      => EBX = 0
mov $0, -8(ebp)     => remainder = 0
mov 8(%ebp), %ecx   => ECX = 1
mov %ecx, -4(%ebp)  => quotient = 1

shrl -4(%ebp)       => quotient = 0, CF =1
mov $1, -8(%ebp)    => remainder = 1
```


call dec_to_bin 2nd time:
0xbffff68c
1
0xbffff688
ret add of 2nd call
0xbffff684
old ebp
0xbffff680
quotient = 0
0xbffff67c
remainder = 1
0xbffff678
1
0xbffff674
push -8(%ebp)
1
0xbffff670
push $0
0
0xbffff66c
call cons
ret add of cons
0xbffff668
ebp,esp
old ebp
0xbffff664
call malloc
eax
head1
0x804a008
tail1
0x804a1c
call dec_to_bin 2nd time:
0xbffff680
calling cons
quotient = 0
0xbffff67c
remainder = 1
0xbffff678
1
0xbffff674
1
0xbffff670
0
0xbffff66c
ret add of cons
0xbffff668
ebp
old ebp
0xbffff664
push %ebx
esp
0
0xbffff660
0xbffff65c
eax
head1 = 1
0x804a008
tail1 = 0
0x804a01c
mov 12(%ebp), %ebx => EBX = 1
mov %ebx, (%eax) => head1 =1
mov 8(%ebp), %ebx => EBX =0
mov %ebx, 4(%eax) => tail1 = 0


call dec_to_bin 2nd time:
0xbffff68c
After ending calling cons
1
0xbffff688
ret add of 2nd call
0xbffff684
old ebp
0xbffff680
quotient = 0
0xbffff67c
remainder = 1
0xbffff678
1
0xbffff674
push %ebx
0
0xbffff670
push %eax
0x804a008
0xbffff66c
call append
ret add of append
0xbffff668
ebp
esp
old ebp
0xbffff664
eax
head1 = 1
0x804a008
tail1 = 0
0x804a01c
pop %ebx => EBX = 0


call dec_to_bin 2nd time:
0xbffff678
call append
1
0xbffff674
0
0xbffff670
0x804a008
0xbffff66c
ret add of append
0xbffff668
ebp
old ebp
0xbffff664
push %ebx
esp
0
0xbffff660
0xbffff65c
0xbffff658
eax
head1 = 1
0x804a008
tail1 = 0
0x804a01c
mov 12(%ebp),%ebx => EBX = 0
cmp $0, %ebx
mov 8(%ebp),%ebx => EBX points to
mov %ebx, %eax 0x804a008
jmp end_append => end calling append
pop %ebx => EBX = 0, points nowhere





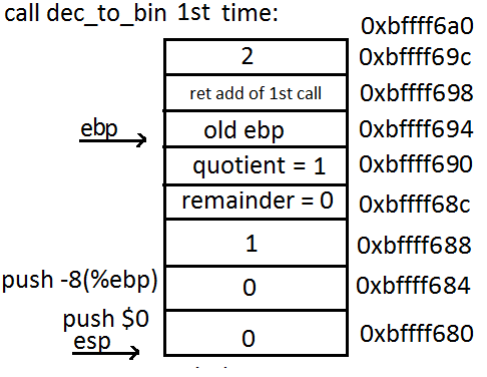
After end calling append, eip also reaches the end of 2nd call dec_to_bin.

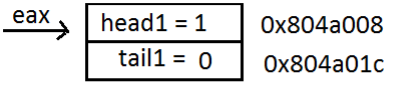
pop %ebx => EBX = 1

Continue 1st call dec_to_bin:

mov %eax, %ebx => EBX points to 0x804a008

mov $0, 8(%ebp) => remainder = 0



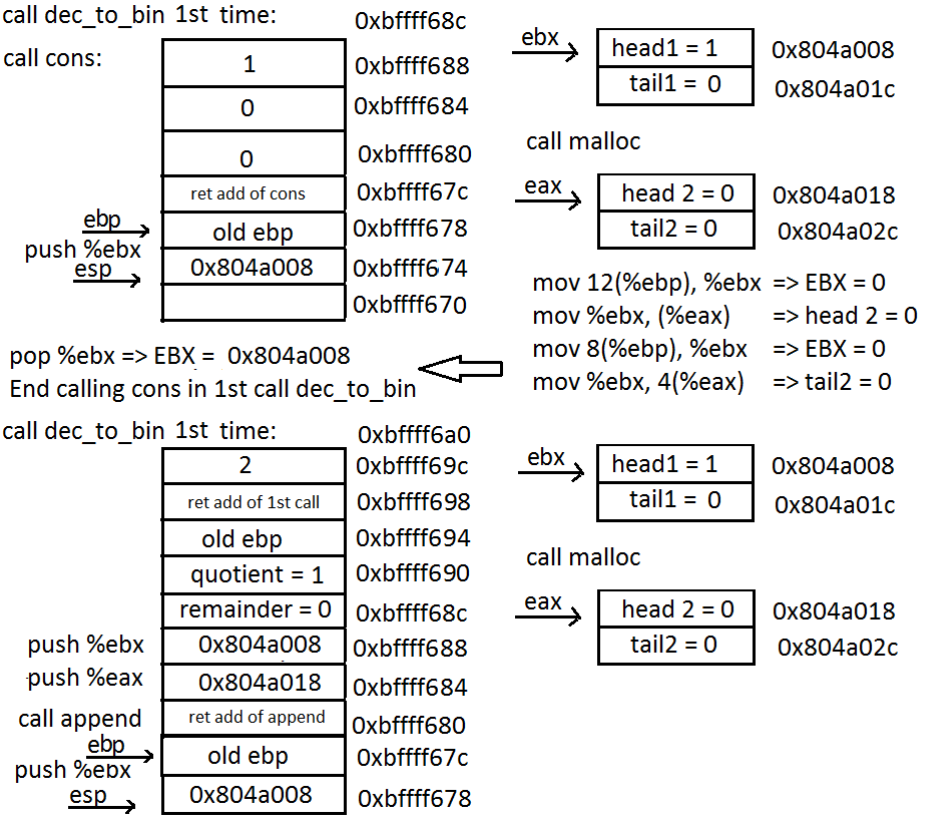


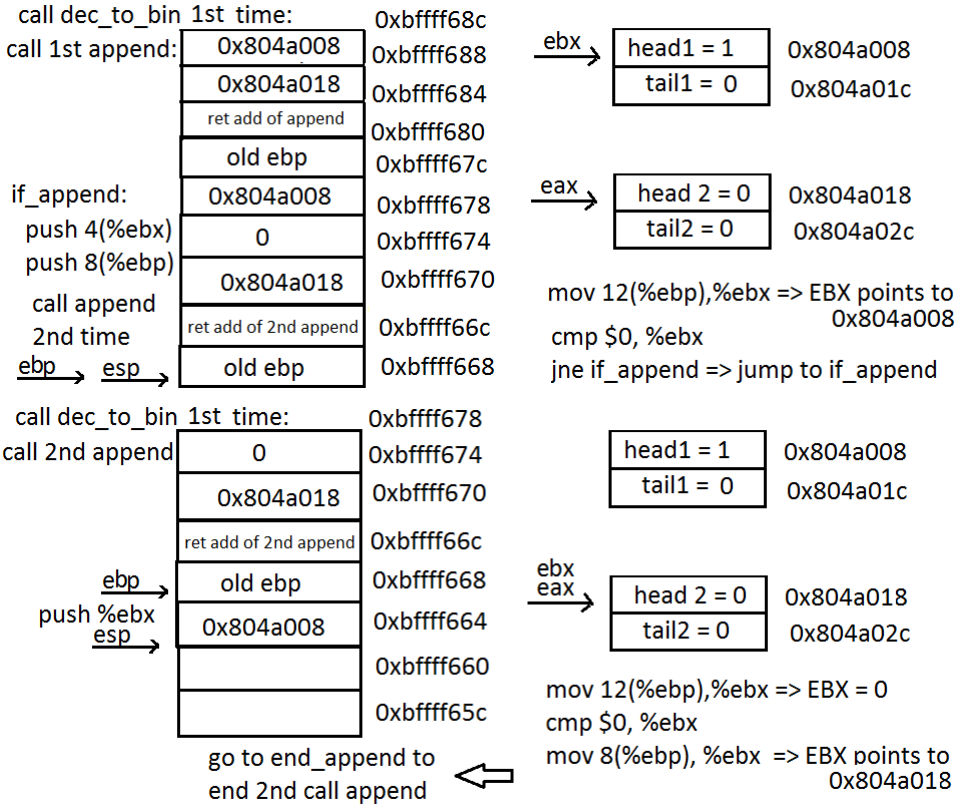
mov 8(%ebp), %ecx => ECX = 2

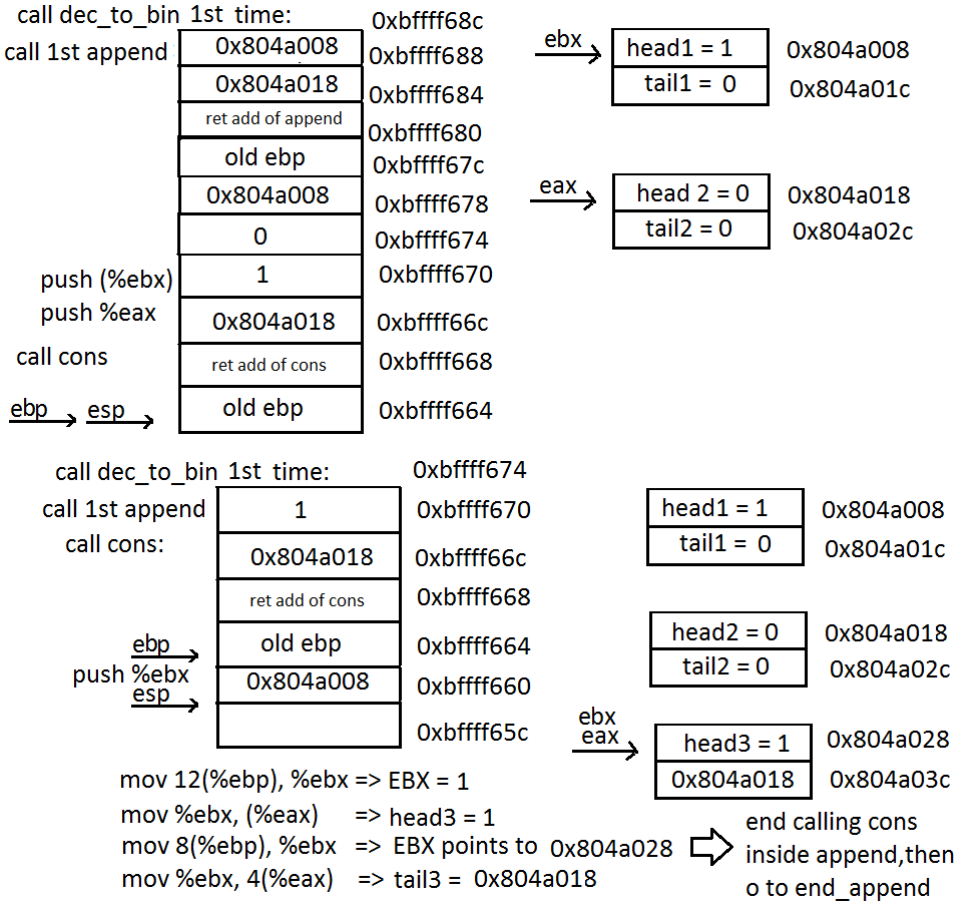
mov %ecx, -4(%ebp) => quotient = 2

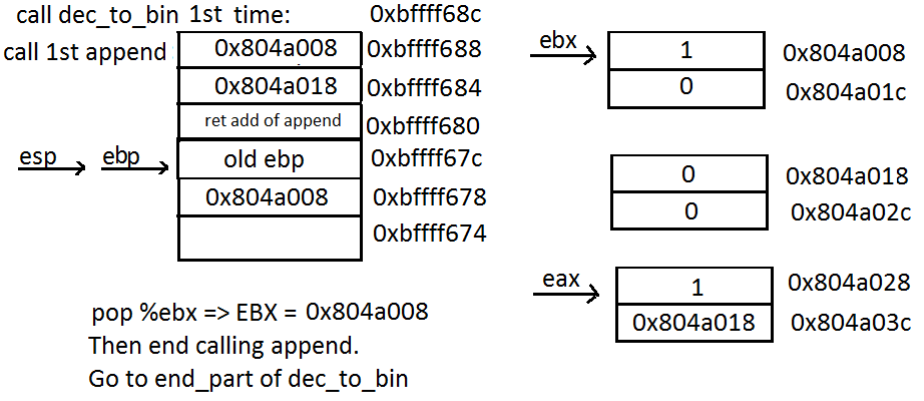
shrl -4(%ebp) => quotient =1, CF = 0

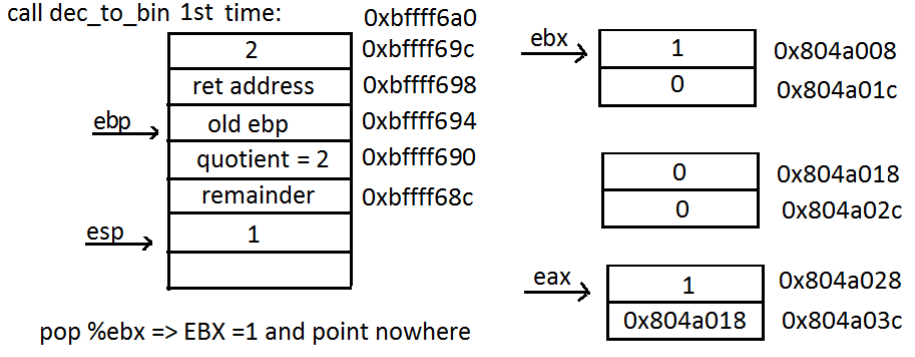
jnc call_cons => jump to call_cons


call dec_to_bin 1st time:
0xbffff68c
call cons:
1
0xbffff688
0
0xbffff684
0
0xbffff680
ret add of cons
0xbffff67c
ebp
old ebp
0xbffff678
push %ebx
esp
0x804a008
0xbffff674
0xbffff670
ebx
head1 = 1
tail1 = 0
0x804a008
0x804a01c
call malloc
eax
head 2 = 0
tail2 = 0
0x804a018
0x804a02c
mov 12(%ebp), %ebx => EBX = 0
mov %ebx, (%eax) => head 2 = 0
mov 8(%ebp), %ebx => EBX = 0
mov %ebx, 4(%eax) => tail2 = 0
pop %ebx => EBX = 0x804a008
End calling cons in 1st call dec_to_bin
call dec_to_bin 1st time:
0xbffff6a0
2
0xbffff69c
ret add of 1st call
0xbffff698
old ebp
0xbffff694
quotient = 1
0xbffff690
remainder = 0
0xbffff68c
push %ebx
0x804a008
0xbffff688
push %eax
0x804a018
0xbffff684
call append
ret add of append
0xbffff680
ebp
old ebp
0xbffff67c
push %ebx
esp
0x804a008
0xbffff678
ebx
head1 = 1
tail1 = 0
0x804a008
0x804a01c
call malloc
eax
head 2 = 0
tail2 = 0
0x804a018
0x804a02c


call dec_to_bin 1st time:
call 1st append:
0x804a008
0x804a018
ret add of append
old ebp
if_append:
0x804a008
push 4(%ebx)
0
push 8(%ebp)
0x804a018
call append
2nd time
ret add of 2nd append
ebp
esp
old ebp
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
ebx
head1 = 1
tail1 = 0
0x804a008
0x804a01c
eax
head 2 = 0
tail2 = 0
0x804a018
0x804a02c
mov 12(%ebp),%ebx => EBX points to 0x804a008
cmp $0, %ebx
jne if_append => jump to if_append
call dec_to_bin 1st time:
call 2nd append
0
0x804a018
ret add of 2nd append
ebp
old ebp
push %ebx
esp
0x804a008
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
0xbffff660
0xbffff65c
head1 = 1
tail1 = 0
0x804a008
0x804a01c
ebx
eax
head 2 = 0
tail2 = 0
0x804a018
0x804a02c
mov 12(%ebp),%ebx => EBX = 0
cmp $0, %ebx
mov 8(%ebp), %ebx => EBX points to 0x804a018
go to end_append to
end 2nd call append


call dec_to_bin 1st time:
call 1st append
0x804a008
0x804a018
ret add of append
old ebp
0x804a008
0
1
push (%ebx)
push %eax
0x804a018
call cons
ret add of cons
ebp
esp
old ebp
0xbffff68c
0xbffff688
0xbffff684
0xbffff680
0xbffff67c
0xbffff678
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
ebx
head1 = 1
tail1 = 0
0x804a008
0x804a01c
eax
head 2 = 0
tail2 = 0
0x804a018
0x804a02c
call dec_to_bin 1st time:
call 1st append
call cons:
1
0x804a018
ret add of cons
ebp
old ebp
push %ebx
esp
0x804a008
0xbffff674
0xbffff670
0xbffff66c
0xbffff668
0xbffff664
0xbffff660
0xbffff65c
head1 = 1
tail1 = 0
0x804a008
0x804a01c
head2 = 0
tail2 = 0
0x804a018
0x804a02c
ebx
eax
head3 = 1
0x804a018
0x804a028
0x804a03c
mov 12(%ebp), %ebx => EBX = 1
mov %ebx, (%eax) => head3 = 1
mov 8(%ebp), %ebx => EBX points to 0x804a028
mov %ebx, 4(%eax) => tail3 = 0x804a018
end calling cons
inside append,then
o to end_append





The head of the list is where eax points to. We actually create 3 nodes but just 2 last nodes are connected to form a linked list.