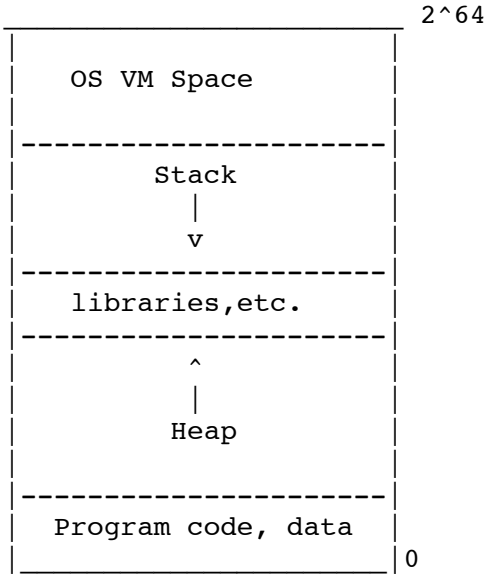
Dynamic Memory Allocation with malloc()

The Virtual Address Space

Virtual Memory is a feature provided by the operating system that allows each individual process (each program) to run with the illusion that is has exclusive access to all of the main memory on the system. When a program is launched, the operating system creates a new *virtual address space* for the process.

The details vary depending on the operating system, but in general the virtual address space looks something like this:



Two areas of interest are the *stack* and the *heap*. Both of these areas grow and shrink dynamically as the code executes, unlike the program code/data area which is of a fixed sized and is loaded once when the program execution begins. The stack is where function calls are created and processed. When a function call is made, the values of variables, etc are pushed onto the stack. When a function returns, the values are "popped" off the stack and becomes available again. This is how state can be maintained across function calls. Consider the following *recursive* function that searches an array for a given value:

```
int find(int *searcharray,int pos,int length, int target) {
    if (pos >= length) return -1;
    if (searcharray[pos] == target) return pos;
    find(searcharray, pos+1, length, target);
}
```

The *heap* grows from the "bottom" (lower memory addresses) up and gives us the ability to give the program access to more memory on the fly. We call this **dynamic memory allocation**. Consider a program that reads and stores integers from stdin. If we know input size ahead of time, then we can just create a fixed-size integer array. In this case, all of the memory space is created and stored in the program data section of the virtual address space. It is of fixed size and we cannot alter the size once program execution begins. If, on the other hand, we do not know ahead of time what the size will be then we may need to ask the operating system for more memory at run time. We do this by using the C function called *malloc()*.Calls to malloc() will

"grow" the heap upward and give your program access to the memory at the location.

malloc()

malloc is a C function that allocates memory for a dsignated number of bytes and returns a pointer to newly allocated memory block. The prototype for malloc() is:

```
void * malloc(size_t size);
```

void * is the generic pointer data type. You can use the generic pointer type to *hold* memory address but you cannot *dereference* a generic pointer. This means that we need to use type casting in order to use memory returned by a call to malloc().

size_t is a special data type that designates the number of bytes in another data type. It is really just another type of integer and you can treat it as such when it comes to arithmetic expressions.

So if we want to allocate memory for an integer at run time, we call malloc() and ask for a chunk of memory that is the size of an integer. We use the helper function sizeof to make sure we ask for the correct amount of memory:

```
int *p = (int *)malloc(sizeof(int)); // allocates memory to hold one integer
                                     //  and assigns the memory address to
                                     //  to the pointer *p
```

The following code allocates memory for an integer, sets a value and then prints the memory address and the value stored there to stdout.

```
#include
#include

int main(void) {

  int *ip=(int *)malloc(sizeof(int));   // allocate memory
  *ip=5;                                // store 5 at memory location
  printf("%p = %d\n", ip, *ip);
}
```

the output would look something like this:

```
0x100100080 = 5
```

using malloc() to generate arrays

Recall that arrays in C are really just pointers in disgusie. We can dynamically generate an array by inreasing the number of bytes we request from our malloc call. If we wish to dynamically allocate an integer array of size 200, then we would use a call to malloc like this:

```
int *array=(int *)malloc(200*sizeof(int));   // allocate array of 200 integers
```

We can either use array notation of pointer arithmetic when accessing the elements of our array:

```
#include
#include

int main(void) {
```

```
  int i;
  int *array=(int *)malloc(200*sizeof(int));   // allocate array of 200 integers

  // initialize the array to zero
  for(i=0;i<200;i++)
    array[i]=0;

  // store value using array notation
  array[14]=2;

  //store value using pointer arithmetic
  *(array+15) = 15;

  // print to demonstrate both methods of access
  printf("%d %d %d %d\n", array[14], *(array+14), array[15], *(array+15));

}
```

This code would output something like this:

```
2 2 15 15
```

Question: What is the difference between...

```
  int *p;
  int number;

  scanf("%d", &number);
  p = (int *)malloc(number * sizeof(int));
```

... and ...

```
  int *p;
  int i,number;

  scanf("%d", &number);
  for(i=0;i< number; i++)
    p = (int *)malloc(sizeof(int));
```

free()

Once we have finished with memory requested through dynamic allocation, we must return that memory to heap. We do so by using the *free()* function:

```
#include
#include

int main(void) {

  int *p;
  int number;

  scanf("%d", &number);
  p = (int *)malloc(number * sizeof(int));

  do_something(p);
```

```
  free(p);
  p=NULL;

  return 0;
}
```

free() returns the space to which it points to the heap **AND** the pointer is then undefined. We set p equal to NULL in order to designate that the pointer variable no longer points to anything.

Question: What is wrong with the following code?

```
int main(void) {
  int *p;
  int i,number;

  scanf("%d", &number);
  for(i=0;i
```