```
Chapter 13 Memory Virtualization

multiprocessing, timesharing placed demands
   + expensive to copy/restore memory footprint of a process
   + protection to avoid rogue process from corrupting memory used by OS or other processes


Fig. 13.2 example of early systems
OS code, user code in single memory space

Fig. 13.3
heap - dynamic allocation, grows one way (down)
stack - local variables, arguments, grows other way (up)

13.4 Goals for memory virtualization
  transparency - process should not know how address translation works
  efficient
  protected

Example Code Listing p. 6

Every Address you see is virual
Each process memory address is reported in terms of
a flat virtual address space

As we saw before, many processes can each have a
datum having the same virtual memory address.

Types of memory

14.1 stack allocation

void func() {
  int x;
}

push space on stack
pop when function exits
The stack makes recursion possible

heap allocated memory
  dynamic size
  longer-term, memory needed even when function exits

#include <stdlib.h>

void func() {
   int* x = (int*)malloc(sizeof(int));

   PCB* list = (PCB*)malloc(sizeof(PCB));

   char** words = (char**)malloc(sizeof(char*)*NUM_WORDS);

   // allow extra byte for EOS symbol
   char* clone = (char*)malloc(sizeof(strlen(myString)+1));

malloc receives an integer that gives amount of
memory needed in bytes and returns a pointer (address) if
successful; else, NULL.

Releasing memory from example above
free(x);
free(list);
free(words);
free(clone);
```

```
Common errors with dynamic memory
a) forget to alloc memory
b) not alloc enough
c) forget to initialize alloc'd memory
d) forget to free alloc'd memory

Aside - lifetime of memory leak


15.3 Hardware-based address translation
Each process has an assigned base and limit
that demarks its assigned location in physical RAM
base register
limit register

given virtual address V

if (V < limit)
  physical address = base + V
else
  trap - memory out of bounds error
```