Memory Management

- Primary role of memory manager:
  - Allocates primary memory to processes
  - Maps process address space to primary memory
  - Minimizes access time using cost effective memory configuration

- Memory management approaches range from primitive bare-machine approach to sophisticated paging and segmentation strategies for implementing virtual memory.
Relocating Executables

- Compile, Link, and Load phases.
- Source program, relocatable object modules, absolute program.
- Dynamic address relocation using relocation registers.
- Memory protection using limit registers. (violating the limit generates an hardware interrupt, often called \textit{segment violation}, that results in a fatal execution error.)
Building the address space

[Diagram showing the process of building an address space, including the interactions between Source Code, Compiler, Relocatable Object Code, Static Library Code, Linker, Loader, and Absolute Program (executable).]
Process Address Space model

- **Text**: 0x00000000
  - Program Binary
- **Initialized Data**: Global/Static variables
- ** uninitialized Data**: Dynamically allocated variables
- **Heap**: Local variables, function/method arguments
- **Stack**: Return values
  - 0xFFFFFFFF

Diagram:

```
  Text
    ├── Program Binary
    │    └── 0x00000000

Uninitialized Data
  └── Global/Static variables

Initialized Data
  └── Dynamically allocated variables

Heap
  └── Local variables, function/method arguments

Stack
  └── Return values
      └── 0xFFFFFFFF
```
Dynamic Memory Allocation in Processes

- Using `malloc` in C or `new` in C/C++/Java and other languages causes memory to be dynamically allocated. Does `malloc` or `new` call the Operating system to get more memory?

- The system creates heap and stack segment for processes at the time of creation. So `new/malloc` already has some memory to work with without having to call the operating system for every memory request from the program.

- If more memory is needed (either due to `malloc/new` or due to stack growing), then a system call (`sbrk()` in Linux/Unix) is made to add more space to the area between the heap and the stack in the process address space.

See example: `malloc-and-OS.c, sbrk-test.c` in lab folder `memory-management`.
Using `sbrk` to create a memory pool

```c
/* memory-management/sbrk-test.c */
/* appropriate header files */
/* memory pool size is 2^22 or 4MB */
#define MAX_MEM_SIZE 4194304

int main(int argc, char **argv) {
    int i, value, *slot;
    void *pool = sbrk(MAX_MEM_SIZE);
    if (pool < 0) {
        perror("Could not allocate memory pool!");
        exit(1);
    }
    /* Now pool is an array of MAX_MEM_SIZE bytes */
    printf("starting address for memory pool = %p\n", pool);
    for (i=0; i<1000000; i++) {
        slot = pool+4*i;
        *slot = i;
    }

    for (i=0; i<10; i++) {
        memcpy(&value, pool+4*i, sizeof(int));
        printf("int stored at slot[%d]=%d\n", i, value);
        /*printf("int stored at slot[%d]=%d\n", i, *(int *)(pool+4*i));*/
        /*int * array = (int *) pool; array[i]*/
    }
    exit(0);
}
```
Strategies for memory management by the operating system.

- **Fixed Partitions**: Simple but inflexible. Leads to internal fragmentation.
- **Variable partitions**: Flexible but more complex. Can lead to external fragmentation (which can be solved using memory compaction).
Free List Management

- Maintain separate lists for free blocks and reserved blocks to speed up allocation but that would make the merging of free blocks more complicated.
- Keep the free blocks sorted by size.
- Instead of a separate data structure, the first word in a free block could be its size and the second word could be the pointer to the next free block.
- Strategies for finding the memory chunk requested:
  - First Fit.
  - Best Fit.
  - Next Fit.
  - Worst Fit.
An example where first fit does better than best fit.

Two segments: 1300, 1200.
Requests: 1000, 1100, 250.

First fit does OK but best fit gets stuck.

Similarly, we can come up with examples where best fit does better than first fit. Similarly for worst fit and next fit.
How does dynamic memory allocation work in a program

The memory model is a one-dimensional array. That implies that multi-dimensional arrays, pointers and objects all have to map to one-dimensional array that represents the memory.

References:

1. *Introduction to Algorithms* by Cormen, Lesiserson, Rivest and Stein. Section 10.4 (Implementing Pointers and Objects)

2. *The C Programming Language* by Kernighan and Ritchie. Section 5.4 (Address Arithmetic) and Section 8.7 (A Storage Allocator)
Let’s design a simple allocator for a double-linked list that has a *prev*, *data* and *next* fields.

We will represent the three fields with three separate one-dimensional arrays.

Pointers will simply be indices into the arrays. We will use -1 as the null pointer.
Simple Memory Allocator example
Simple Allocator Code

```c
int prev[MAX], data[MAX], next[MAX];
int free;

void init_allocator() {
    for (int i=0; i < MAX - 1 ; i++)
        next[i] = i + 1;
    next[MAX - 1] = -1;
    free = 0;
}

int allocate_object() {
    int ptr = -1;
    if (free != -1) {
        ptr = free;
        free = next[free];
    }
    return ptr;
}

void free_object(int ptr) {
    next[ptr] = free;
    free = ptr;
}
```
1. Why don't we need to set or reset `prev` fields of the objects in the `allocate_object` or `free_object` methods.

2. How would compact the memory in-place? (that is, you can only use a constant number of extra variables). By compaction, we mean move the nodes in use to the start of the arrays and collect the free nodes at the end. (Hint: Use a permutation).

3. Generalize to use only one one-dimensional array.

4. Write pseudocode for a homogeneous collection of objects implemented by a single array representation.
Malloc: Storage Allocator

Memory layout for malloc

A block returned by malloc
Malloc Algorithms

- The free storage is kept as a list of free blocks. Each block contains a pointer to the next free block, a size, and the space itself. The blocks are kept in increasing order of addresses and the last block points to the first.
- To allocate memory, the free list is scanned until a big enough block is found ("first-fit" algorithm). One optimization done is to leave the free list pointer where the last block was found ("next-fit" algorithm). Then one of the following three steps is taken.
  - If the block is exactly the right size, it is unlinked from the list and returned to the user.
  - If the block is too big, it is split, and the proper amount is returned to the user while the residue remains on the free list.
  - If no big-enough chunk is found, then another large chunk is obtained from the operating system and linked into the free list. Then a part of this new free block is split and returned to the user.
- To free an allocated block, we search the free list to find the proper place to insert it. If the block being freed is adjacent to a free block on either side, they are coalesced into a single bigger block, so the storage does not become fragmented.
Malloc Block Layout and Allocation

```c
typedef double Align; /* for alignment to double boundary */
union header { /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size; /* size of this block */
    } s;
    Align x; /* force alignment of blocks */
};
typedef union header Header;
```

- The requested size in bytes is rounded up to proper number of header-sized units; the block that will be allocated contains one more unit, for the header itself, and this is the value recorded in the size field.
- The pointer returned by `malloc` points at the free space, not at the header itself.
Simple Malloc Implementation

- See examples (based directly from the C book, Section 8.7):
  - memory-management/simple-malloc/simple-malloc.h
  - memory-management/simple-malloc/simple-malloc.c
Simple Malloc Implementation

#include <unistd.h>
#include <stdlib.h>
typedef double Align; /* for alignment to long boundary */
union header { /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;/* size of this block */
    } s;
    Align x;
};
typedef union header Header;

void *simple_malloc(unsigned nbytes);
void simple_free(void *ap);
void print_free_list();
#endif
Simple Malloc Implementation (contd.)

```c
static Header *morecore(unsigned);
static Header base; /* empty list to get started */
static Header *freep = NULL; /* start of free list */

/* simple_malloc: general purpose storage allocator */
void *simple_malloc(unsigned nbytes)
{
    Header *p, *prevp;
    unsigned nunits;
    nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
    if ((prevp = freep) == NULL) { /* no free list yet */
        /* this becomes a marker node with size 0 */
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) { /* big enough */
            if (p->s.size == nunits) /* exactly */
                prevp->s.ptr = p->s.ptr;
            else { /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
        freep = prevp; /* next-fit */
        return (void *) (p+1);
    }
    if (p == freep) /* wrapped around free list */
        if ((p = morecore(nunits)) == NULL)
            return NULL; /* none left */
}
```
Simple Malloc Implementation (contd.)

```c
#define NALLOC 1024 /* minimum #units to request */

/* morecore: ask system for more memory */
static Header *morecore(unsigned nu)
{
    void *cp;
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (void *) -1) /* no memory left */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    simple_free((void *)(up+1));
    return freep;
}
```
/* put block ap in the free list */

void simple_free(void *ap) {
    Header *bp, *p;

    bp = (Header *) ap - 1; /* point to block header */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr) {
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break; /* freed block at start or end of arena */
    }

    if (bp + bp->s.size == p->s.ptr) { /* join to upper neighbor */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else {
        bp->s.ptr = p->s.ptr;
    }

    if (p + p->s.size == bp) { /* join to lower neighbor */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else {
        p->s.ptr = bp;
        freep = p;
    }
}
/ * print out the free list for demonstration purposes * /
void print_free_list()
{
    Header *next;
    int i;

    if (!freep) {
        printf("free list is empty\n");
        return;
    }

    i=0;
    printf("\n");
    next = freep;
    do {
        printf("node %d: size = %u bytes\n", i,
               next->s.size * (unsigned) sizeof(Header));
        next = next->s.ptr;
        i++;
    } while (next != freep);
    printf("\n");
}
Getting the source for the Standard C library

- Download the source rpm for gcc C library named glibc:
  `yumdownloader --source glibc`

- Install the source package:
  `rpm -ivh glibc-*src.rpm`

- Finally prep the source code. This step may require you to install additional packages.
  ```
  cd ~/rpmbuild/SPECS/
rpmbuild -bp glibc.spec
  ```

- Now you can look at the source for the glibc (replace 2.21 with the version that you downloaded)
  ```
  cd /rpmbuild/BUILD/glibc-2.21
  ```
Buddy System Memory Management

The Walrus and the Coder

“The time has come,” the Walrus said,
“To talk of many things:
Of shells–and system calls–and sealing-stacks–
Of threads–and pointers–
And why the C code is pointing over–
And whether buddies have wings.”

Modified from the original “The Walrus and the Carpenter” by by Lewis Carroll in “Through the Looking-Glass and What Alice Found There, 1872”

“The time has come,” the Walrus said,
“To talk of many things:
Of shoes–and ships–and sealing-wax–
Of cabbages–and kings–
And why the sea is boiling hot–
And whether pigs have wings.”
Buddy System

- Assume that the memory pool is of size $2^m$, with addresses 0 through $2^m - 1$.

- Block sizes are of powers of two, $2^k$, $0 \leq k \leq m$. There are $m + 1$ different lists: $avail[0], avail[1], \ldots, avail[m]$. The $i$th list keeps track of blocks of size $2^i$. At the beginning there is one block of size $2^m$.

- All memory allocations are always done in sizes that are powers of two. Each block has a tag field which denotes if the block is free or reserved. Each block also has the usual links next, prev to maintain each list as a doubly-linked list. Finally each block also has a $kval$ field that stores the size of the block (the value $k$ is stored for a block of size $2^k$.)
Buddy System (contd.)

- Address of a block of size $2^k$ is a multiple of $2^k$ (that is, at least $k$ zeroes on the right). For example, a block of size 32 has an address of the form $xxx \ldots xx00000$. After splitting the addresses of the two buddy blocks of size 16 are $xxx \ldots xx00000$ and $xxx \ldots xx10000$.

- Whenever a block is split into two halves, the two new blocks are called *buddies*. If we know the address of a block and its size, then we also know the address of its buddy.

\[
\text{buddy}_k(x) = \begin{cases} 
  x + 2^k & \text{if } x \mod 2^{k+1} = 0 \\
  x - 2^k & \text{if } x \mod 2^{k+1} = 2^k
\end{cases}
\]

- The address of the buddy can be computed using an exclusive-or operation. In Java and C, the exclusive-or operator is `^`. So the buddy calculation can be written as follows:

\[
x \hat{(}1 \ll k)\]
Buddy System Allocation

- **Step A1**: Find block. To allocate a block of size $2^k$, search the $k$th list and return the first free block. If the $k$th list is empty, then search the next higher list and so on until we find a free block. If no such list is found, then the allocation was unsuccessful and returns a null value. Otherwise, let the block be found in the $j$th list.
- **Step A2**: Remove from list. Remove first block from the $j$th list.
- **Step A3**: Split required? If $j$ equals $k$, we have found a block of the right size. Return the appropriate address and exit.
- **Step A4**: Split. Split the first block in the $j$th list and add the unused half to the $(j - 1)$th list. Set $j \leftarrow j - 1$. Go back to **Step A3**.
Buddy System Free

Free a block of size $2^k$ at address L.

- **Step F1:** Is buddy available? Check for a buddy of size $2^k$ for block at address L. Go to Step F3 if the buddy isn’t available.

- **Step F2:** Merge with buddy. Merge freed block with buddy in $k$th list. Set $k \leftarrow k + 1$. Go back to Step F1.

- **Step F3:** Put on list. Add freed block to the front of the $k$th list.
Memory pool size $n = 2^m = 2^{20} = 1$MB.

- The first example shows the initial lists.
- The second example shows what happens when we allocate 1 byte. Note that the minimum block size is 32 in the system so that’s why the allocation stops at $2^5$.
- The third example shows that the blocks merge back up when we free the memory allocated.
Buddy System—Example

Buddy system initialized.
Buddy system lists after initialization.

List 0: head = 0x7f344e6500b8 --> <null>
List 1: head = 0x7f344e6500d0 --> <null>
List 2: head = 0x7f344e6500e8 --> <null>
List 3: head = 0x7f344e650100 --> <null>
List 4: head = 0x7f344e650118 --> <null>
List 5: head = 0x7f344e650130 --> <null>
List 6: head = 0x7f344e650148 --> <null>
List 7: head = 0x7f344e650160 --> <null>
List 8: head = 0x7f344e650178 --> <null>
List 9: head = 0x7f344e650190 --> <null>
List 10: head = 0x7f344e6501a8 --> [tag=1,kval=20,addr=0xf98000] --> <null>
List 11: head = 0x7f344e6501c0 --> <null>
List 12: head = 0x7f344e6501d8 --> <null>
List 13: head = 0x7f344e6501f0 --> <null>
List 14: head = 0x7f344e650208 --> <null>
List 15: head = 0x7f344e650220 --> <null>
List 16: head = 0x7f344e650238 --> <null>
List 17: head = 0x7f344e650250 --> <null>
List 18: head = 0x7f344e650268 --> <null>
List 19: head = 0x7f344e650280 --> <null>
List 20: head = 0x7f344e650298 --> [tag=1,kval=20,addr=0xf98000] --> <null>

Number of available blocks = 1
Buddy System—Example

Buddy system succeeding in allocating 1 byte.
Buddy system lists after malloc’ing 1 byte.

List 0: head = 0x7f344e6500b8 --> <null>
List 1: head = 0x7f344e6500d0 --> <null>
List 2: head = 0x7f344e6500e8 --> <null>
List 3: head = 0x7f344e650100 --> <null>
List 4: head = 0x7f344e650118 --> <null>
List 5: head = 0x7f344e650130 --> [tag=1,kval=5,addr=0xf98020] --> <null>
List 6: head = 0x7f344e650148 --> [tag=1,kval=6,addr=0xf98040] --> <null>
List 7: head = 0x7f344e650160 --> [tag=1,kval=7,addr=0xf98080] --> <null>
List 8: head = 0x7f344e650178 --> [tag=1,kval=8,addr=0xf98100] --> <null>
List 9: head = 0x7f344e650190 --> [tag=1,kval=9,addr=0xf98200] --> <null>
List 10: head = 0x7f344e6501a8 --> [tag=1,kval=10,addr=0xf98400] --> <null>
List 11: head = 0x7f344e6501c0 --> [tag=1,kval=11,addr=0xf98800] --> <null>
List 12: head = 0x7f344e6501d8 --> [tag=1,kval=12,addr=0xf99000] --> <null>
List 13: head = 0x7f344e6501f0 --> [tag=1,kval=13,addr=0xf9a000] --> <null>
List 14: head = 0x7f344e650208 --> [tag=1,kval=14,addr=0xf9c000] --> <null>
List 15: head = 0x7f344e650220 --> [tag=1,kval=15,addr=0xfa0000] --> <null>
List 16: head = 0x7f344e650238 --> [tag=1,kval=16,addr=0xfa8000] --> <null>
List 17: head = 0x7f344e650250 --> [tag=1,kval=17,addr=0xfb8000] --> <null>
List 18: head = 0x7f344e650268 --> [tag=1,kval=18,addr=0xfd8000] --> <null>
List 19: head = 0x7f344e650280 --> [tag=1,kval=19,addr=0x1018000] --> <null>
List 20: head = 0x7f344e650298 --> <null>

Number of available blocks = 15
Buddy System—Example

Buddy system succeeding in free’ing 1 byte.
Buddy system lists after free’ing the block.

List 0: head = 0x7f344e6500b8 --> <null>
List 1: head = 0x7f344e6500d0 --> <null>
List 2: head = 0x7f344e6500e8 --> <null>
List 3: head = 0x7f344e650100 --> <null>
List 4: head = 0x7f344e650118 --> <null>
List 5: head = 0x7f344e650130 --> <null>
List 6: head = 0x7f344e650148 --> <null>
List 7: head = 0x7f344e650160 --> <null>
List 8: head = 0x7f344e650178 --> <null>
List 9: head = 0x7f344e650190 --> <null>
List 10: head = 0x7f344e6501a8 --> [tag=1,kval=20,addr=0xf98000] --> <null>
List 11: head = 0x7f344e6501c0 --> <null>
List 12: head = 0x7f344e6501d8 --> <null>
List 13: head = 0x7f344e6501f0 --> <null>
List 14: head = 0x7f344e650208 --> <null>
List 15: head = 0x7f344e650220 --> <null>
List 16: head = 0x7f344e650238 --> <null>
List 17: head = 0x7f344e650250 --> <null>
List 18: head = 0x7f344e650268 --> <null>
List 19: head = 0x7f344e650280 --> <null>
List 20: head = 0x7f344e650298 --> [tag=1,kval=20,addr=0xf98000] --> <null>

Number of available blocks = 1
Memory pool size $n = 2^m = 2^{20} = 1$MB.

Work out what happens with the following example.

**Requests:** 70KB (Process A), 35KB (Process B), 80KB (Process C), free A, 60KB (Process D), free B, free D, free C.

Each request is rounded up to the next power of two.
Buddy System—Example 2

Buddy system initialized.
Buddy system lists after initialization.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c5332d0 --> <null>
List 10: head = 0x7f272c5332e8 --> <null>
List 11: head = 0x7f272c533300 --> <null>
List 12: head = 0x7f272c533318 --> <null>
List 13: head = 0x7f272c533330 --> <null>
List 14: head = 0x7f272c533348 --> <null>
List 15: head = 0x7f272c533360 --> <null>
List 16: head = 0x7f272c533378 --> <null>
List 17: head = 0x7f272c533390 --> <null>
List 18: head = 0x7f272c5333a8 --> <null>
List 19: head = 0x7f272c5333c0 --> <null>
List 20: head = 0x7f272c5333d8 --> [tag=1,kval=20,addr=0x1123000] --> <null>

Number of available blocks = 1
Buddy System—Example 2

Buddy system lists after malloc’ing 70KB.

- List 0: head = 0x7f272c5331f8 --> <null>
- List 1: head = 0x7f272c533210 --> <null>
- List 2: head = 0x7f272c533228 --> <null>
- List 3: head = 0x7f272c533240 --> <null>
- List 4: head = 0x7f272c533258 --> <null>
- List 5: head = 0x7f272c533270 --> <null>
- List 6: head = 0x7f272c533288 --> <null>
- List 7: head = 0x7f272c5332a0 --> <null>
- List 8: head = 0x7f272c5332b8 --> <null>
- List 9: head = 0x7f272c533300 --> <null>
- List 10: head = 0x7f272c53332e8 --> <null>
- List 11: head = 0x7f272c5333300 --> <null>
- List 12: head = 0x7f272c5333318 --> <null>
- List 13: head = 0x7f272c5333330 --> <null>
- List 14: head = 0x7f272c5333348 --> <null>
- List 15: head = 0x7f272c5333360 --> <null>
- List 16: head = 0x7f272c5333378 --> <null>
- List 17: head = 0x7f272c5333390 --> [tag=1,kval=17,addr=0x1143000] --> <null>
- List 18: head = 0x7f272c53333a8 --> [tag=1,kval=18,addr=0x1163000] --> <null>
- List 19: head = 0x7f272c53333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
- List 20: head = 0x7f272c53333d8 --> <null>

Number of available blocks = 3
Buddy System—Example 2

Buddy system lists after malloc’ing 35KB.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c5332d0 --> <null>
List 10: head = 0x7f272c5332e8 --> <null>
List 11: head = 0x7f272c533300 --> <null>
List 12: head = 0x7f272c533318 --> <null>
List 13: head = 0x7f272c533330 --> <null>
List 14: head = 0x7f272c533348 --> <null>
List 15: head = 0x7f272c533360 --> <null>
List 16: head = 0x7f272c533378 --> [tag=1,kval=16,addr=0x1153000] --> <null>
List 17: head = 0x7f272c533390 --> <null>
List 18: head = 0x7f272c5333a8 --> [tag=1,kval=18,addr=0x1163000] --> <null>
List 19: head = 0x7f272c5333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 20: head = 0x7f272c5333d8 --> <null>

Number of available blocks = 3
Buddy System—Example 2

Buddy system lists after malloc’ing 80KB.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c533300 --> <null>
List 10: head = 0x7f272c533318 --> <null>
List 11: head = 0x7f272c533330 --> <null>
List 12: head = 0x7f272c533348 --> <null>
List 13: head = 0x7f272c533360 --> <null>
List 14: head = 0x7f272c533378 --> [tag=1,kval=16,addr=0x1153000] --> <null>
List 15: head = 0x7f272c533390 --> [tag=1,kval=17,addr=0x1183000] --> <null>
List 16: head = 0x7f272c5333a8 --> <null>
List 17: head = 0x7f272c5333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 18: head = 0x7f272c5333d8 --> <null>
List 19: head = 0x7f272c533400 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 20: head = 0x7f272c5333d8 --> <null>

Number of available blocks = 3
Buddy System–Example 2

Buddy system lists after free’ing the 70KB block.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c5332d0 --> <null>
List 10: head = 0x7f272c5332e8 --> <null>
List 11: head = 0x7f272c533300 --> <null>
List 12: head = 0x7f272c533318 --> <null>
List 13: head = 0x7f272c533330 --> <null>
List 14: head = 0x7f272c533348 --> <null>
List 15: head = 0x7f272c533360 --> <null>
List 16: head = 0x7f272c533378 --> [tag=1,kval=16,addr=0x1153000] --> <null>
List 17: head = 0x7f272c533390 --> [tag=1,kval=17,addr=0x1123000] --> [tag=1,kval=17,addr=0x1183000] --> <null>
List 18: head = 0x7f272c5333a8 --> <null>
List 19: head = 0x7f272c5333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 20: head = 0x7f272c5333d8 --> <null>

Number of available blocks = 4
Buddy System—Example 2

Buddy system lists after malloc’ing 60KB.

List 0: head = 0x7f272c5331f8 --&gt; <null>
List 1: head = 0x7f272c533210 --&gt; <null>
List 2: head = 0x7f272c533228 --&gt; <null>
List 3: head = 0x7f272c533240 --&gt; <null>
List 4: head = 0x7f272c533258 --&gt; <null>
List 5: head = 0x7f272c533270 --&gt; <null>
List 6: head = 0x7f272c533288 --&gt; <null>
List 7: head = 0x7f272c5332a0 --&gt; <null>
List 8: head = 0x7f272c5332b8 --&gt; <null>
List 9: head = 0x7f272c5332d0 --&gt; <null>
List 10: head = 0x7f272c5332e8 --&gt; <null>
List 11: head = 0x7f272c533300 --&gt; <null>
List 12: head = 0x7f272c533318 --&gt; <null>
List 13: head = 0x7f272c533330 --&gt; <null>
List 14: head = 0x7f272c533348 --&gt; <null>
List 15: head = 0x7f272c533360 --&gt; <null>
List 16: head = 0x7f272c533378 --&gt; <null>
List 17: head = 0x7f272c533390 --&gt; [tag=1,kval=17,addr=0x1123000] --&gt;
                      [tag=1,kval=17,addr=0x1183000] --&gt; <null>
List 18: head = 0x7f272c5333a8 --&gt; <null>
List 19: head = 0x7f272c5333c0 --&gt; [tag=1,kval=19,addr=0x11a3000] --&gt; <null>
List 20: head = 0x7f272c5333d8 --&gt; <null>

Number of available blocks = 3
Buddy System—Example 2

Buddy system lists after free’ing the 35KB block.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c5332d0 --> <null>
List 10: head = 0x7f272c5332e8 --> <null>
List 11: head = 0x7f272c533300 --> <null>
List 12: head = 0x7f272c533318 --> <null>
List 13: head = 0x7f272c533330 --> <null>
List 14: head = 0x7f272c533348 --> <null>
List 15: head = 0x7f272c533360 --> <null>
List 16: head = 0x7f272c533378 --> [tag=1,kval=16,addr=0x1143000] --> <null>
List 17: head = 0x7f272c533390 --> [tag=1,kval=17,addr=0x1123000] -->
[  tag=1,kval=17,addr=0x1183000] --> <null>
List 18: head = 0x7f272c5333a8 --> <null>
List 19: head = 0x7f272c5333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 20: head = 0x7f272c5333d8 --> <null>

Number of available blocks = 4
Buddy System—Example 2

Buddy system lists after free'ing the 60KB block.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c5332d0 --> <null>
List 10: head = 0x7f272c5332e8 --> <null>
List 11: head = 0x7f272c533300 --> <null>
List 12: head = 0x7f272c533318 --> <null>
List 13: head = 0x7f272c533330 --> <null>
List 14: head = 0x7f272c533348 --> <null>
List 15: head = 0x7f272c533360 --> <null>
List 16: head = 0x7f272c533378 --> <null>
List 17: head = 0x7f272c533390 --> [tag=1,kval=17,addr=0x1183000] --> <null>
List 18: head = 0x7f272c5333a8 --> [tag=1,kval=18,addr=0x1123000] --> <null>
List 19: head = 0x7f272c5333c0 --> [tag=1,kval=19,addr=0x11a3000] --> <null>
List 20: head = 0x7f272c5333d8 --> <null>

Number of available blocks = 3
Buddy System—Example 2

Buddy system lists after free’ing the 80KB block.

List 0: head = 0x7f272c5331f8 --> <null>
List 1: head = 0x7f272c533210 --> <null>
List 2: head = 0x7f272c533228 --> <null>
List 3: head = 0x7f272c533240 --> <null>
List 4: head = 0x7f272c533258 --> <null>
List 5: head = 0x7f272c533270 --> <null>
List 6: head = 0x7f272c533288 --> <null>
List 7: head = 0x7f272c5332a0 --> <null>
List 8: head = 0x7f272c5332b8 --> <null>
List 9: head = 0x7f272c533300 --> <null>
List 10: head = 0x7f272c533318 --> <null>
List 11: head = 0x7f272c533330 --> <null>
List 12: head = 0x7f272c533348 --> <null>
List 13: head = 0x7f272c533360 --> <null>
List 14: head = 0x7f272c533378 --> <null>
List 15: head = 0x7f272c533390 --> <null>
List 16: head = 0x7f272c5333a8 --> <null>
List 17: head = 0x7f272c5333c0 --> <null>
List 18: head = 0x7f272c5333d8 --> [tag=1,kval=20,addr=0x1123000] --> <null>

Number of available blocks = 1
Buddy System—Analysis

Advantages:

- Searching a block of size $k$ requires searching only one list of free blocks of size $k$ instead of all the free blocks.
- Merging free blocks is much faster by design.

Disadvantages:

- All memory requests have to be rounded up to the nearest power of two, which may cause significant internal fragmentation.

A modified version of the buddy system is used in Linux.
Buddy System References


So what is used in modern operating systems?

Modern operating systems all use some form of variable partitioning. However, memory is allocated in fixed-size blocks (called “pages”), which greatly simplifies free list management.

- The Linux kernel also uses the buddy system, with further modifications to minimize external fragmentation, along with various slab allocators to manage the memory within blocks. These are listed below:

  - SLAB is a complex allocator that performs well on a variety of workloads. See article at [http://en.wikipedia.org/wiki/Slab_allocation](http://en.wikipedia.org/wiki/Slab_allocation)
  - SLUB, the kernel’s default, has a much simpler design and superior debugging features. However it has significant regressions on some benchmarks.
  - SLOB (Simple List of Blocks) allocator for embedded devices and machines that require a very small kernel footprint.

- BSD-based systems like Mac OS X also use a slab-based allocation system.

- jemalloc is a modern memory allocator that is a replacement for malloc that employs, among others, the buddy technique.
So what is used in modern operating systems? (contd)

Microsoft Windows memory management:

- Multiple memory pools of two types: Nonpaged pool and Paged pool. System starts with four paged pools and one non-paged pool and can grow up to 64 pools to support multicore architecture.
- On 64-bit systems, nonpaged pool has a maximum size of 75% of the physical memory or 128GB, whichever is smaller. Paged pool has a maximum size of 128GB.
- Use process explorer (Sysinternals tools) to see pool information. Click on View and then System Information.
- Look-aside Lists for faster allocation of fixed-size blocks. These lists will automatically grow or shrink depending upon usage.
- Heap Manager manages memory inside smaller chunks. It has a granularity of 8/16 bytes on 32-bit/64-bit systems.
Program and Data Locality

- Program locality...most programs spend 90% of the time in 10% of the code.
- How good is the data locality for the following data structures?
  - stacks,
  - queues,
  - linked lists,
  - heap data-structure,
  - binary search trees.
Swapping relies on dynamic relocation hardware. The decision as to when to swap is made by the memory manager.

The memory manager can deallocate the memory for a blocked process and allocate the memory to other processes.

In a time sharing system, a process could be swapped out even if it is not blocked to equitably share memory and the CPU. (for example, swapping can be activated when the number of active users exceeds a certain threshold.)

When a swapped out process returns to ready state, the process manager informs the memory manager to swap it back in.

Swapping takes considerable time. Hence the memory manager should swap only when it is needed.
Introduction to Virtual Memory

- Allows a process to use more memory than present physically.
- Also allows only part of the address space of a process to be present in the primary memory. This makes multiprogramming more effective.
- Relies on spatial reference locality of program text and data.
- Relies on dynamic relocation hardware as well as other specialized hardware support.
Processes (potentially running on different CPUs) communicate using shared memory.

The simplest way to set up shared memory is to let processes share parts of their address space.

If each CPU has its own cache, then we have the problem of cache coherence. Caches could be strongly consistent or weakly consistent.

Shared memory can also be set up among unrelated processes on a single processor system as an efficient means of communication. (example on next few frames).
Shared Memory Segments

- Older style shared memory calls: `shmget(...)`, `shmat(...)`, `shmdt(...)`, `shmdtl(...)`
  Check the shared memory segments with the command: `ipcs`

- POSIX standard shared memory calls. (supported under Linux and Mac OS X) `shm_open(...)`, `ftruncate(...)`, `mmap(...)`, `shm_unlink(...)`
  Under Linux, check the shared memory segments with the command: `ls -l /dev/shm`

- Under MS Windows use `CreateFileMapping(...), MapViewOfFile(...), UnmapViewOfFile(...)`. See examples in `ms-windows/memory-management/`
Memory Mapping

- **Under Linux/Mac OS X.** Use `mmap(...)` and `munmap(...)`. 
  //.. appropriate header files
  void *mmap(void *start, size_t length, int prot, int flags, 
             int fd, off_t offset);
  int munmap(void *start, size_t length);

- **Under MS Windows.** Use `VirtualAlloc(...)` and `VirtualFree(...)`
  LPVOID WINAPI VirtualAlloc(
      __in_opt LPVOID lpAddress,
      __in SIZE_T dwSize,
      __in DWORD flAllocationType,
      __in DWORD flProtect
  );
  BOOL WINAPI VirtualFree(
      __in LPVOID lpAddress,
      __in SIZE_T dwSize,
      __in DWORD dwFreeType
  );
Creating a Shared Memory Segment (POSIX)

```c
/* lab/memory-management/create_posix_shmem.c */
/* appropriate header files */
#define ARRAY_SIZE 10000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W) /* user read/write */
char buffer[ARRAY_SIZE];

int main(void) {
    int shmid;
    char *shmptr;
    int status;

    strcpy(buffer, "Hello World");
    shmid = shm_open("/amit", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    if (shmid == -1) {
        perror("Error in creating shared memory:");
    }
    status = ftruncate(shmid, SHM_SIZE);
    if (status == -1) {
        perror("Error in sizing shared memory:");
    }
    if ((shmptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE,
                      MAP_FILE | MAP_SHARED, shmid, 0)) == (void *)-1)
        perror("mmap error for shmem:");
    memcpy(shmptr, buffer, ARRAY_SIZE);
    sleep(25);
    shm_unlink="/amit"
    exit(0);
}
```
/* lab/memory-management/access_posix_shmem.c */
/* appropriate header files */
#define ARRAY_SIZE 10000
#define SHM_SIZE 100000
char buffer[ARRAY_SIZE];

int main(void) {
    int shmid;
    char *shmptr;

    shmid = shm_open("/amit", O_RDWR, S_IRWXU);
    if (shmid == -1) {
        perror("Error in creating shared memory: ");
    }
    if ((shmptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE,
        MAP_FILE | MAP_SHARED, shmid, 0)) == (void *)-1)
        perror("mmap error for shmem:");
    memcpy(buffer, shmptr, ARRAY_SIZE);
    printf("Read from shared memory: \"%s\", buffer);
    exit(0);
}
Observing POSIX Shared Memory Segments

The kernel keeps the POSIX shared memory segments in a virtual file system `/dev/shm`. By default, it is usually equal to half the installed memory size on the system (but can be altered in `/etc/fstab`). Use `ls` on that folder to see the shared memory segments.

```bash
[amit@kohinoor memory-management]: ls -l /dev/shm
total 5784
-rw-rw-r-- 1 amit amit 100000 Nov 7 13:58 amit
-r-------- 1 amit amit 67108904 Oct 19 09:47 pulse-shm-1210480228
-r-------- 1 amit amit 67108904 Oct 19 11:20 pulse-shm-1469689579
-r-------- 1 gdm gdm 67108904 Oct 19 09:47 pulse-shm-2908712043
-r-------- 1 amit amit 67108904 Oct 26 14:24 pulse-shm-3200708632
-r-------- 1 amit amit 67108904 Oct 19 09:47 pulse-shm-3973641630
-r-------- 1 amit amit 67108904 Nov 4 03:27 pulse-shm-722623938
-rw-rw-rw- 1 amit amit 16 Oct 20 09:45 sem.ADBE_ReadPrefs_amit
-rw-rw-rw- 1 amit amit 16 Oct 20 09:45 sem.ADBE_REL_amit
-rw-rw-rw- 1 amit amit 16 Oct 20 09:45 sem.ADBE_WritePrefs_amit
```

[amit@kohinoor memory-management]:

Synchronization and Shared Segments

- If multiple processes are modifying data structures stored in a shared memory segment, then we need to synchronize them (similar to global variables in a multi-threaded program).
- Note that the mutexes, semaphores built in with Pthreads library are not visible outside of a process so they cannot be used in this case.
- We can use global semaphores provided via system calls in Linux. See examples memory-management/semdemo.c and memory-management/semrm.c.
- In the MS Windows API, sempahores and mutexes can be assigned a string handle and shared between processes.
Creating Shared Memory Between Processes (older style calls)

`/* lab/memory-management/create_shmem.c */`
`/* appropriate includes */`
```c
#define ARRAY_SIZE 10000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W) /* user read/write */
char buffer[ARRAY_SIZE];
```

```c
int main(void) {
    int shmid;
    char *shmptr;
    key_t key;

    strcpy(buffer, "Hello World");
    key = 1;
    if ((shmid = shmget(key, SHM_SIZE, SHM_MODE|IPC_CREAT)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *) -1)
        err_sys("shmat error");
    memcpy(shmptr, buffer, ARRAY_SIZE);
    sleep(25); // to allow another process to access the shared memory
    /* remove shared memory segment (optional since shared memory segments
     * can be persistent beyond the program that created them */
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");
    exit(0);
}
```
/* lab/memory-management/access_shmem.c */

#define ARRAY_SIZE 10000
#define SHM_SIZE 100000
#define SHM_MODE (SHM_R | SHM_W) /* user read/write */

char buffer[ARRAY_SIZE]; /* uninitialized data = bss */

int main(void)
{
    int shmid;
    char *shmptr;
    key_t key;

    key = 1;
    if ( (shmid = shmget(key, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ( (shmptr = shmat(shmid, 0, 0)) == (void *) -1)
        err_sys("shmat error");
    memcpy(buffer, shmptr, ARRAY_SIZE);
    printf("From shared memory: %s\n", buffer);
    if (shmdt(shmptr) < 0)
        err_sys("shmdt error");
    exit(0);
}
Observing Shared Memory Segments (older style)

Also note the utility `ipcs` lets you find out about active shared memory segments in the system (as well as message queues and semaphores). Here is a sample output from the `ipcs` command.

```
[amit@kohinoor]: ipcs
------ Shared Memory Segments --------
key   shmid  owner  perms  bytes  nattch  status
0x73727372 0  root  666  44172  0
0x7b01333d 1  amit  600  1024  3
0x00000000 1486850 amit  777  196608 2  dest
------ Semaphore Arrays ---------
key   semid  owner  perms  nsems  status
0x6c737372 0  root  666  3
------ Message Queues ---------
key   msqid  owner  perms  used-bytes  messages
```

A corresponding command `ipcrm` lets a user remove shared memory segments etc. (if they have the right permissions).