CIS 450 – Computer Architecture and Organization

Lecture 24: Dynamic Memory Allocation

Mitch Neilsen

(neilsen@ksu.edu)

219D Nichols Hall

Topics

- **From last time … Memory Mapping**
- **Dynamic Memory Allocation**
- **Simple explicit allocators**
  - **Data structures**
  - **Mechanisms**
  - **Policies**
- **BrickOS Example**

Memory Mapping

Creation of new VM *area* done via "memory mapping"

- **Create new vm_area_struct and page tables for area**
- **Area can be backed by (i.e., get its initial values from) :**
  - **Regular file on disk (e.g., an executable object file)**
    - **Initial page bytes come from a section of a file**
  - **Nothing (e.g., bss)**
    - **Initial page bytes are zeros**
- **Dirty pages are swapped back and forth between a special swap file.**

<u>Key point</u>: no virtual pages are copied into physical memory until they are referenced!

- **Known as "demand paging"**
- **Crucial for time and space efficiency**

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

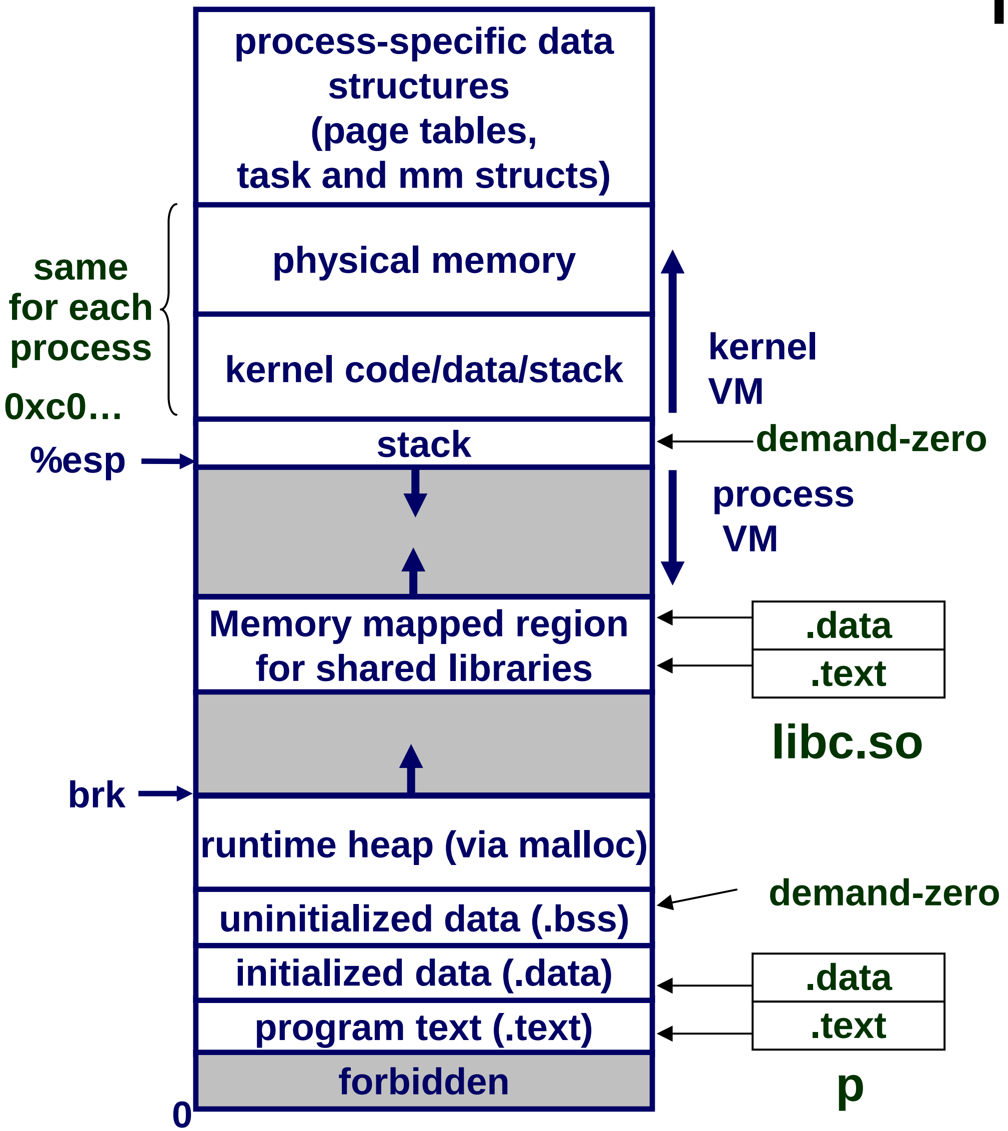
- **Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don't care).**
  - **`prot`: MAP_READ, MAP_WRITE**
  - **`flags`: MAP_PRIVATE, MAP_SHARED**
- **Return a pointer to the mapped area.**
- **Example: fast file copy**
  - **Useful for applications like Web servers that need to quickly copy files.**
  - **`mmap` allows file transfers without copying into user space.**

mmap() Example: Fast File Copy

```
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * mmap.c - a program that uses mmap
 * to copy itself to stdout
 */
```

```
int main() {
  struct stat stat;
  int i, fd, size;
  char *bufp;

  /* open the file & get its size*/
  fd = open("./mmap.c", O_RDONLY);
  fstat(fd, &stat);
  size = stat.st_size;
  /* map the file to a new VM area */
  bufp = mmap(0, size, PROT_READ,
    MAP_PRIVATE, fd, 0);

  /* write the VM area to stdout */
  write(1, bufp, size);
}
```

Exec() Revisited



To run a new program p in the current process using exec( ):

- Free vm_area_struct's and page tables for old areas.
- Create new vm_area_struct's and page tables for new areas.
  - Stack, bss, data, text, shared libs.
  - Text and data backed by ELF executable object file.
  - bss and stack initialized to zero.
- Set PC to entry point in .text
  - Linux will swap in code and data pages as needed.

Fork() Revisited

To create a new process using `fork()`:

- **Make copies of the old process's mm_struct, vm_area_struct's, and page tables.**
  - **At this point the two processes are sharing all of their pages.**
  - **How to get separate spaces without copying all the virtual pages from one space to another?**
    - **"copy on write" technique.**
- **Copy-on-write**
  - **Make pages of writeable areas read-only**
  - **Flag vm_area_struct's for these areas as private "copy-on-write".**
  - **Writes by either process to these pages will cause page faults.**
    - **Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.**

Net result:

- **Copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).**

Memory System Summary

Cache Memory

- Purely a speed-up technique
- Behavior invisible to application programmer and (mostly) OS
- Implemented totally in hardware

Virtual Memory

- Supports many OS-related functions
  - Process creation
  - Task switching
  - Protection
- Combination of hardware & software implementation
  - Software management of tables, allocations
  - Hardware access of tables
  - Hardware caching of table entries (TLB)

Harsh Reality

Memory Matters

Memory is not unbounded

- It must be allocated and managed
- Many applications are memory dominated
  - Especially those based on complex, graph algorithms

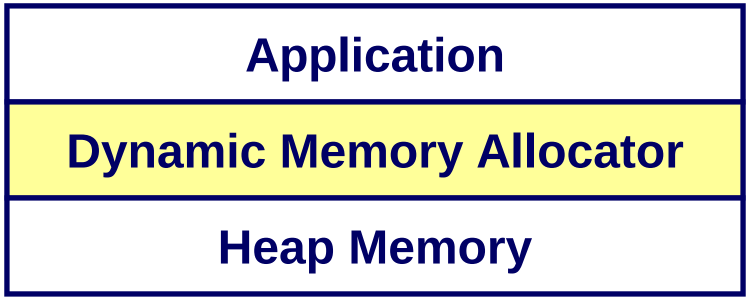
Memory referencing bugs especially pernicious

- Effects are distant in both time and space

Memory performance is not uniform

- Cache and virtual memory effects can greatly affect program performance
- Adapting program to characteristics of memory system can lead to major speed improvements

Dynamic Memory Allocation



Explicit vs. Implicit Memory Allocator

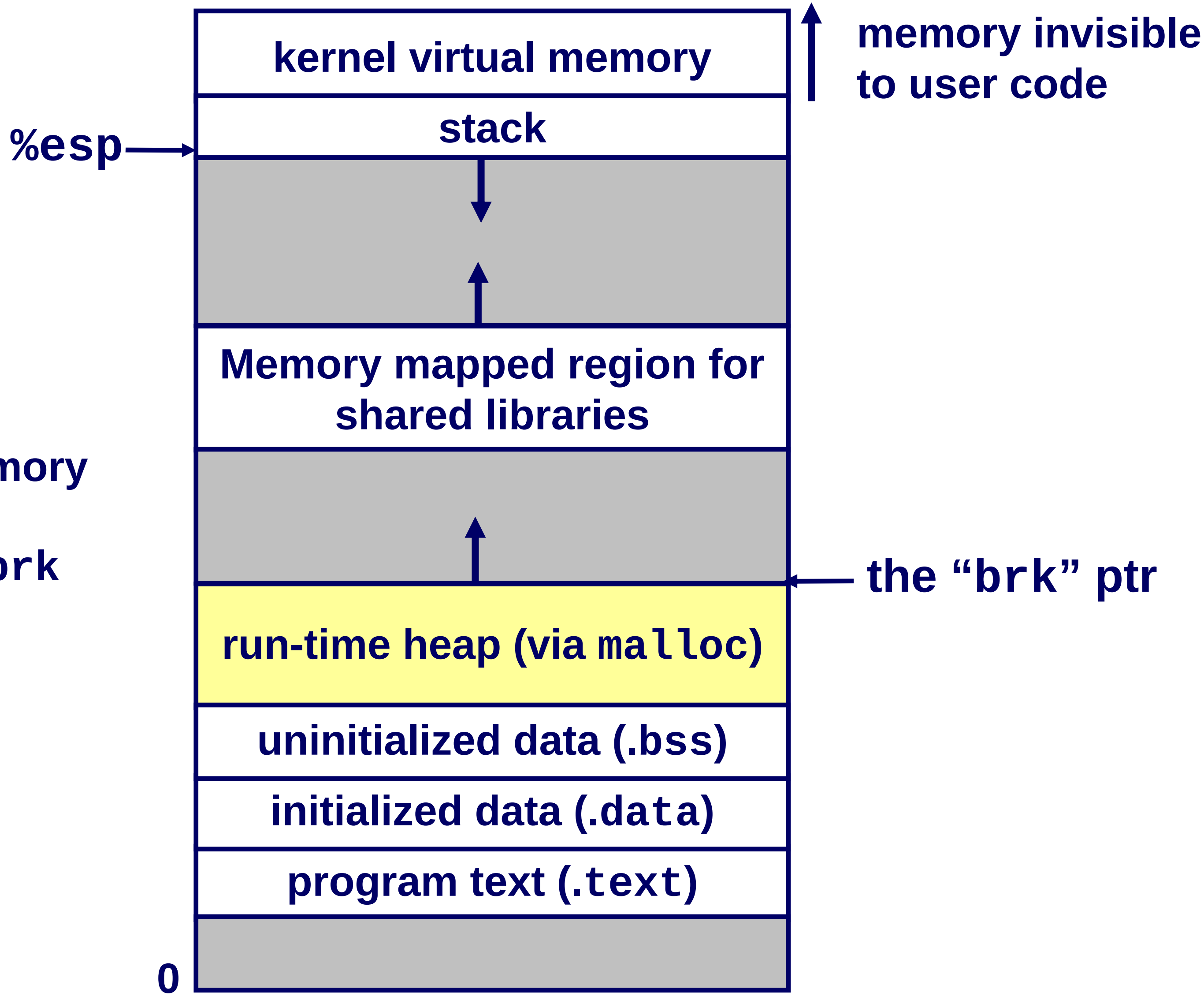
- **Explicit: application allocates and frees space**
  - **E.g., `malloc` and `free` in C**
- **Implicit: application allocates, but does not free space**
  - **E.g. garbage collection in Java, ML or Lisp**

Allocation

- **In both cases the memory allocator provides an abstraction of memory as a set of blocks**
- **Doles out free memory blocks to application**

Will discuss simple explicit memory allocation today

Process Memory Image



Allocators request additional heap memory from the operating system using the `sbrk` function.

Malloc Package

`#include <stdlib.h>`

`void *malloc(size_t size)`

- **If successful:**
  - **Returns a pointer to a memory block of at least `size` bytes, (typically) aligned to 8-byte boundary.**
  - **If `size == 0`, returns NULL**
- **If unsuccessful: returns NULL (0) and sets `errno`.**

`void free(void *p)`

- **Returns the block pointed at by p to pool of available memory**
- **p must come from a previous call to `malloc` or `realloc`.**

`void *realloc(void *p, size_t size)`

- **Changes size of block p and returns pointer to new block.**
- **Contents of new block unchanged up to min of old and new size.**

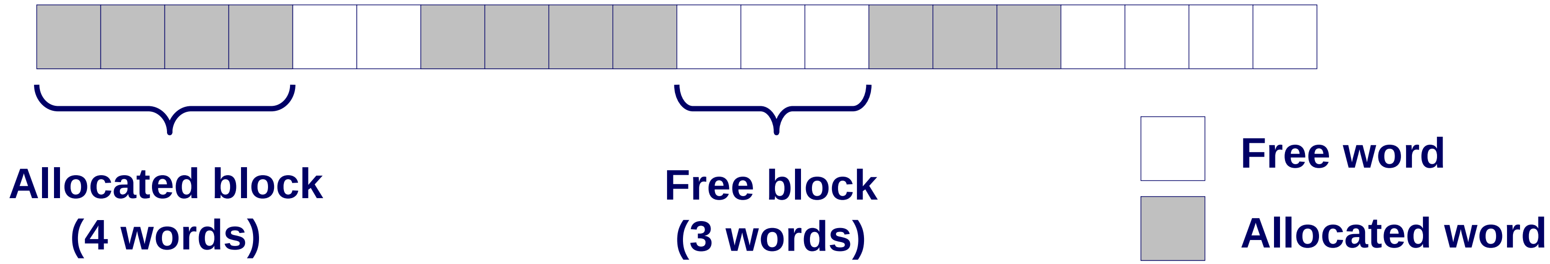
Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
   p = (int *)malloc(n * sizeof(int));
   if (p == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *) realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++) p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

Assumptions

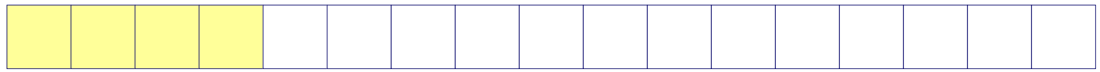
Assumptions made in this lecture

- Memory is word addressed (each word can hold a pointer)

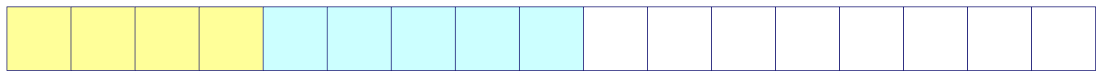


Allocation Examples

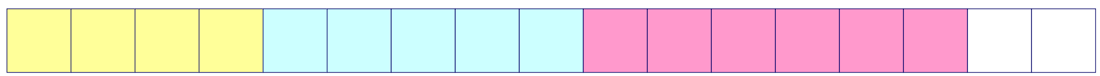
`p1 = malloc(4)`



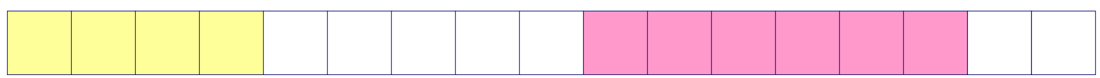
`p2 = malloc(5)`



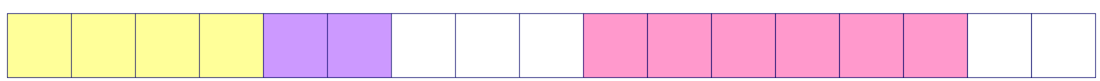
`p3 = malloc(6)`



`free(p2)`



`p4 = malloc(2)`



Constraints

Applications:

- Can issue arbitrary sequence of allocation and free requests
- Free requests must correspond to an allocated block

Allocators

- Can't control number or size of allocated blocks
- Must respond immediately to all allocation requests
  - *i.e.*, can't reorder or buffer requests
- Must allocate blocks from free memory
  - *i.e.*, can only place allocated blocks in free memory
- Must align blocks so they satisfy all alignment requirements
  - 8 byte alignment for GNU malloc (`libc` malloc) on Linux boxes
- Can only manipulate and modify free memory
- Can't move the allocated blocks once they are allocated
  - *i.e.*, compaction is not allowed

Performance Goals: Throughput

Given some sequence of `malloc` and `free` requests:

- $R_0, R_1, ..., R_k, ... , R_{n-1}$

Want to maximize throughput and peak memory utilization.

- **These goals are often conflicting**

Throughput:

- **Number of completed requests per unit time**
- **Example:**
  - **5,000 `malloc` calls and 5,000 `free` calls in 10 seconds**
  - **Throughput is 1,000 operations/second.**

Performance Goals: Peak Memory Utilization

Given some sequence of `malloc` and `free` requests:

- $R_0, R_1, ..., R_k, ... , R_{n-1}$

Def: Aggregate payload P_k:

- **`malloc(p)` results in a block with a *payload* of p bytes.**
- **After request R_k has completed, the *aggregate payload* P_k is the sum of currently allocated payloads.**

Def: Current heap size is denoted by H_k

- **Assume that H_k is monotonically nondecreasing**

Def: Peak memory utilization:

- **After *k* requests, *peak memory utilization* is:**
  - $U_k = (\max_{i<k} P_i) \ / \ H_k$

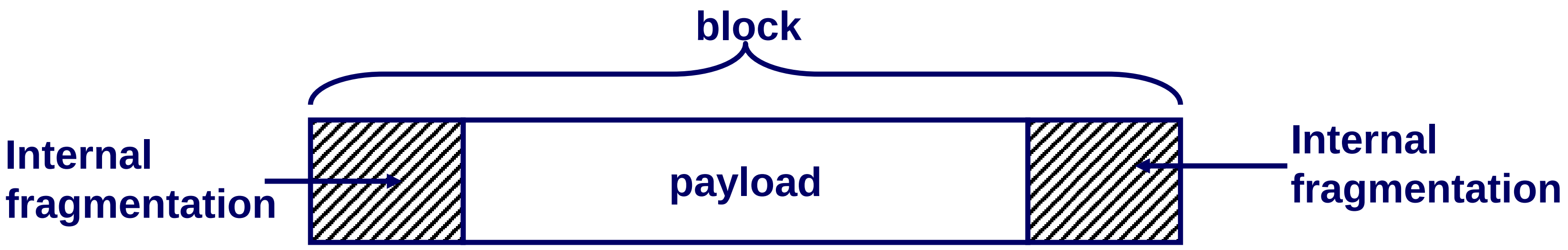
Internal Fragmentation

Poor memory utilization caused by *fragmentation*.

- **Comes in two forms: *internal* and *external* fragmentation**

Internal fragmentation

- **For some block, *internal fragmentation* is the difference between the block size and the payload size.**

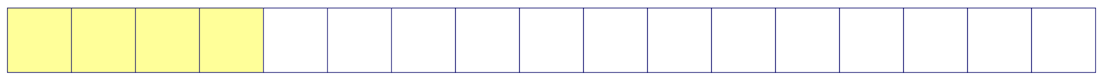


- **Caused by overhead of maintaining heap data structures, padding for alignment purposes, or explicit policy decisions (e.g., not to split the block).**
- **Depends only on the pattern of *previous* requests, and thus is easy to measure.**

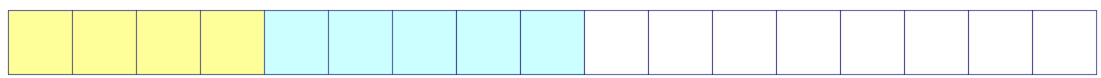
External Fragmentation

Occurs when there is enough aggregate heap memory, but no single free block is large enough

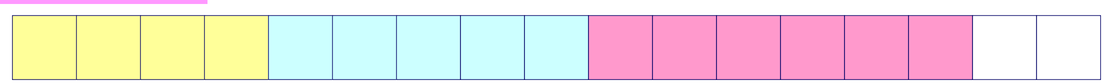
```
p1 = malloc(4)
```



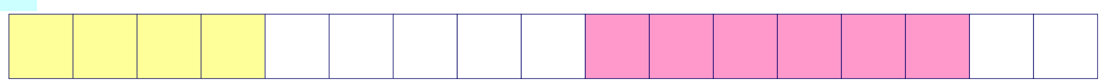
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



```
free(p2)
```



```
p4 = malloc(6)
```

oops!

External fragmentation depends on the pattern of *future* requests, and thus is difficult to measure.

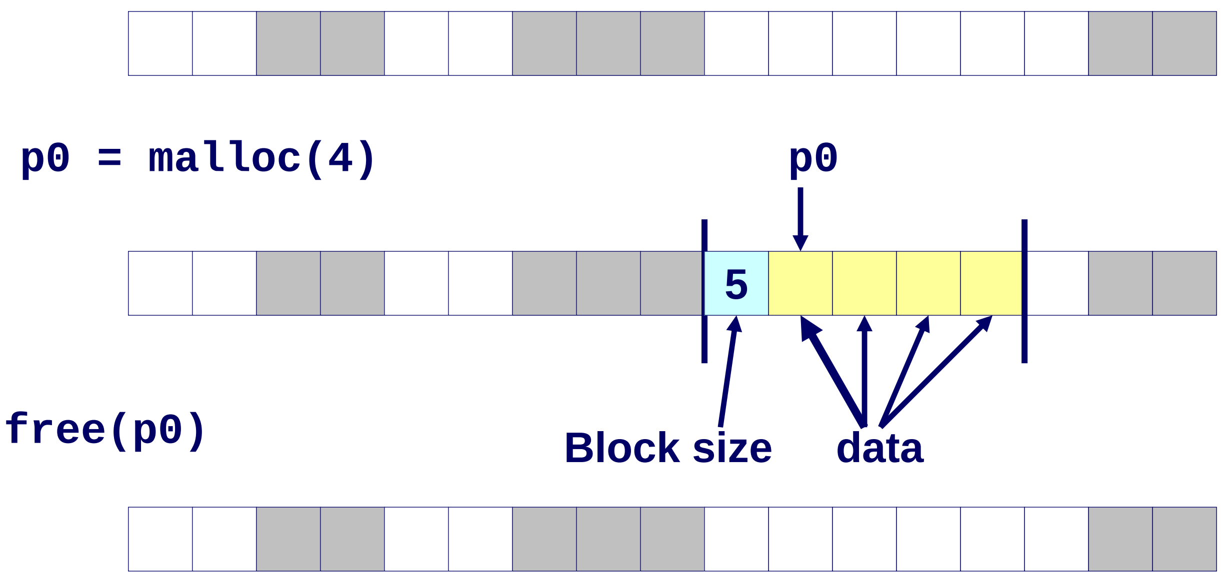
Implementation Issues

- **How do we know how much memory to free just given a pointer?**
- **How do we keep track of the free blocks?**
- **What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?**
- **How do we pick a block to use for allocation -- many might fit?**
- **How do we reinsert freed block?**

Knowing How Much to Free

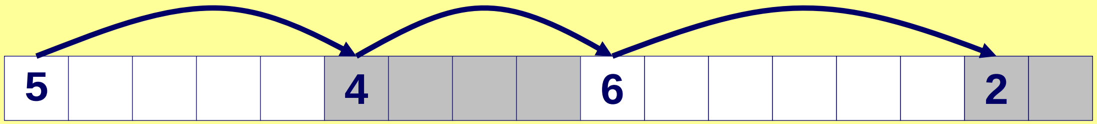
Standard method

- **Keep the length of a block in the word preceding the block.**
  - **This word is often called the *header field* or *header***
- **Requires an extra word for every allocated block**

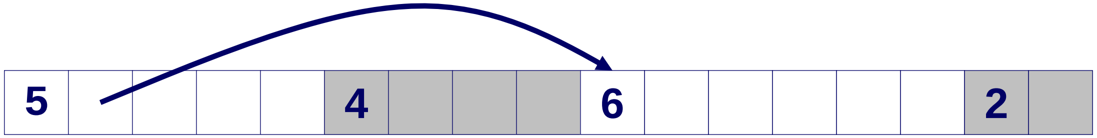


Keeping Track of Free Blocks

***<u>Method 1</u>*: *Implicit list* using lengths -- links all blocks**



***<u>Method 2</u>*: *Explicit list* among the free blocks using pointers within the free blocks**



<u>Method 3</u>*: *Segregated free list

- Different free lists for different size classes

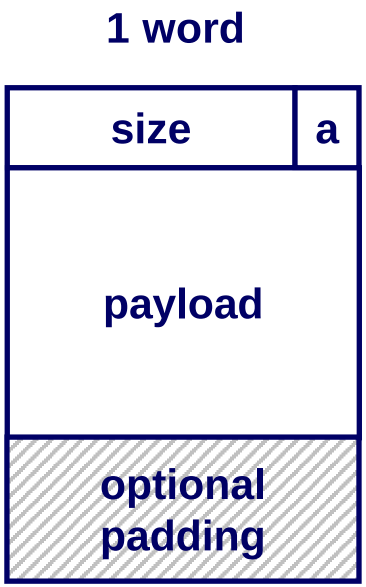
***<u>Method 4</u>*: Blocks sorted by size**

- Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Method 1: Implicit List

- **Need to identify whether each block is free or allocated**
  - **Can use extra bit**
  - **Bit can be put in the same word as the size if block sizes are always multiples of two (mask out low order bit when reading size).**

Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

payload: application data (allocated blocks only)

Implicit List: Finding a Free Block

First fit:

- Search list from beginning, choose first free block that fits

```
p = start;
while ((p < end) &&      \\ not passed end
       ((*p & 1) ||      \\ already allocated
       (*p <= len)))     \\ too small
  p = p + (*p & -2);     \\ goto next block
```

- Can take linear time in total number of blocks (allocated and free)
- In practice it can cause “splinters” at beginning of list

Next fit:

- Like first-fit, but search list from location of end of previous search
- Research suggests that fragmentation is worse

Best fit:

- Search the list, choose the free block with the closest size that fits
- Keeps fragments small --- usually helps fragmentation
- Will typically run slower than first-fit

Bitfields

How to represent the Header:

- **Masks and bitwise operators**

```
#define SIZEMASK              (~0x7)
#define PACK(size, alloc)     ((size) | (alloc))
#define GET_SIZE(p)           ((p)->size & SIZEMASK)
```

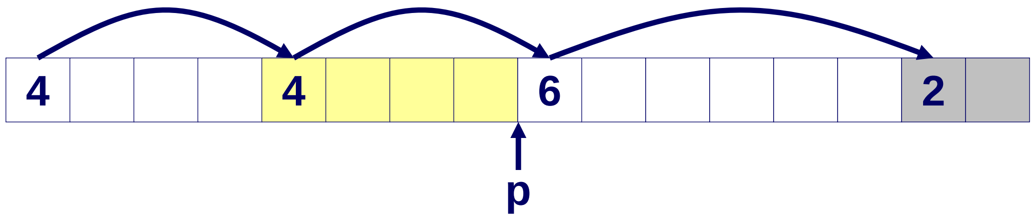
- **Bitfields**

```
  struct {
      unsigned allocated:1;
      unsigned size:31;
} Header;
```

Implicit List: Allocating in Free Block

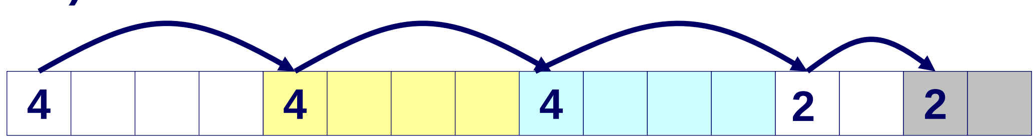
Allocating in a free block - *splitting*

- Since allocated space might be smaller than free space, we might want to split the block




```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // add 1 and round up
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

`addblock(p, 2)`



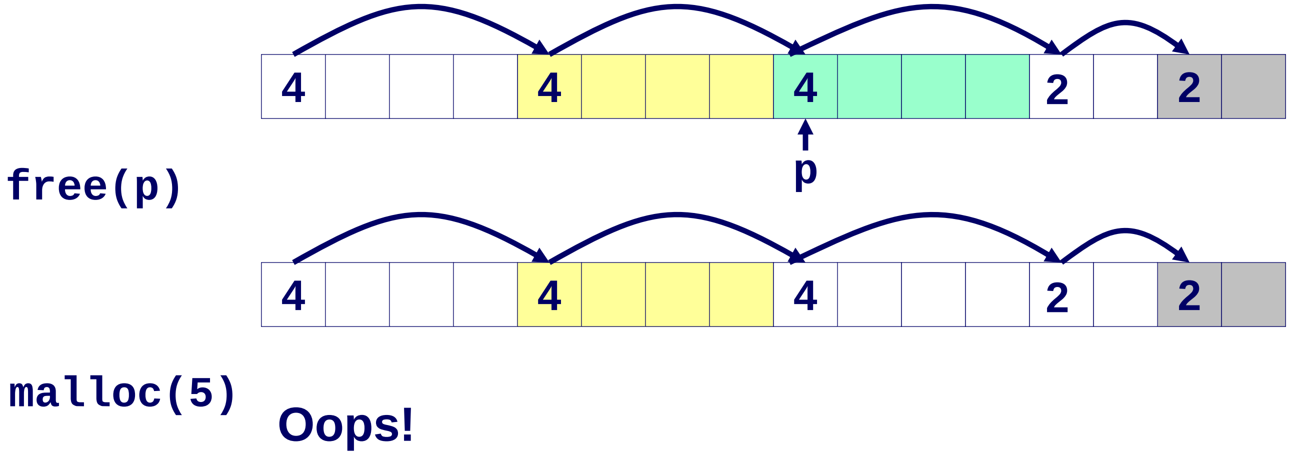
Implicit List: Freeing a Block

Simplest implementation:

- **Only need to clear allocated flag**

```
void free_block(ptr p) { *p = *p & -2}
```

- **But can lead to "false fragmentation"**



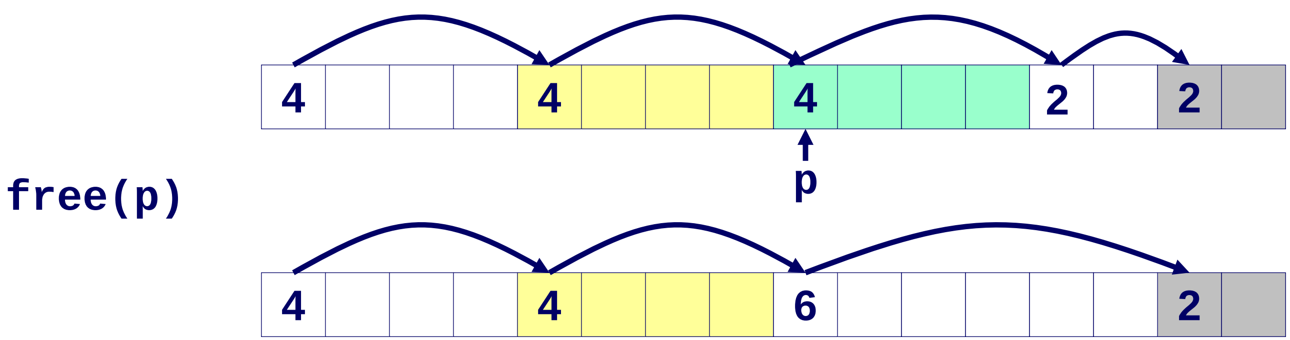
There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

Join (*coalesce*) with next and/or previous block if they are free

- **Coalescing with next block**

```
void free_block(ptr p) {
    *p = *p & -2;           // clear allocated flag
    next = p + *p;          // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;     // add to this block if
}                          //    not allocated
```

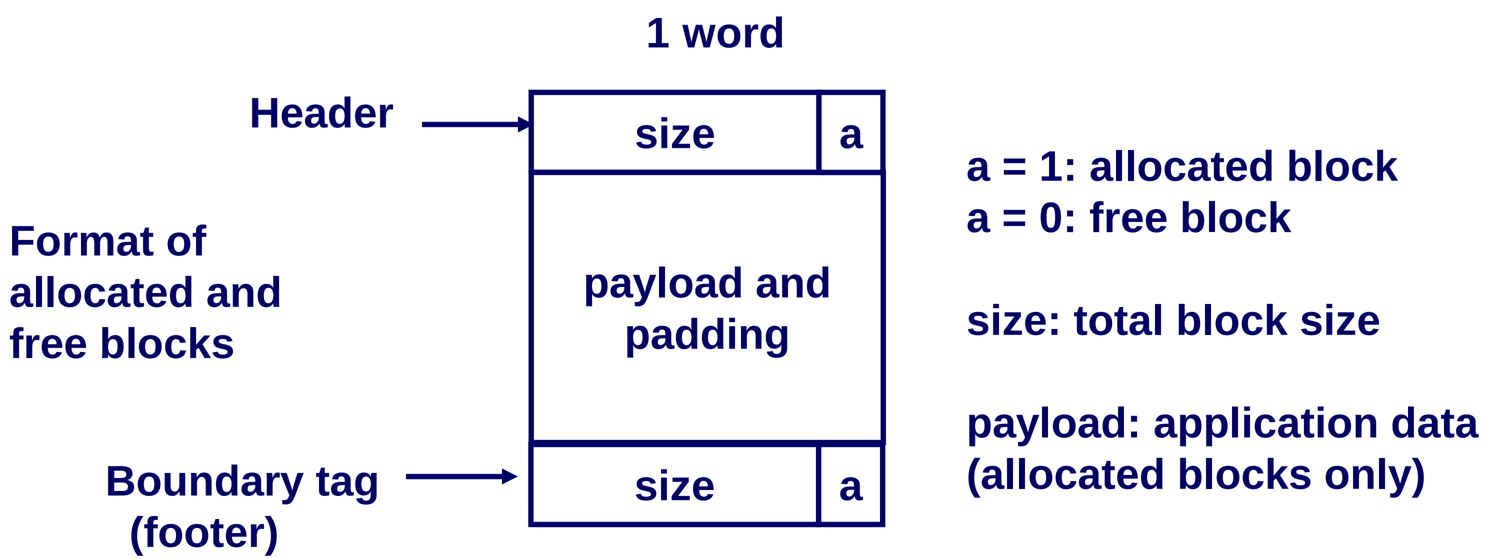


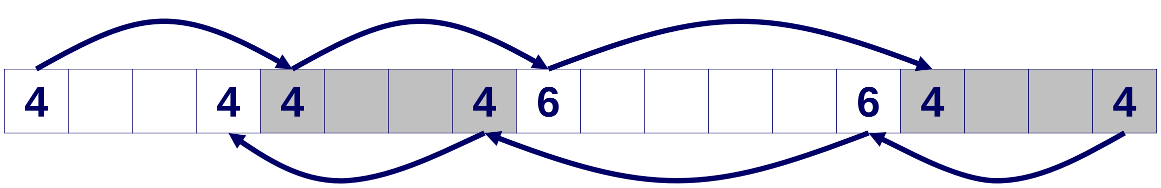
- **But how do we coalesce with previous block?**

Implicit List: Bidirectional Coalescing

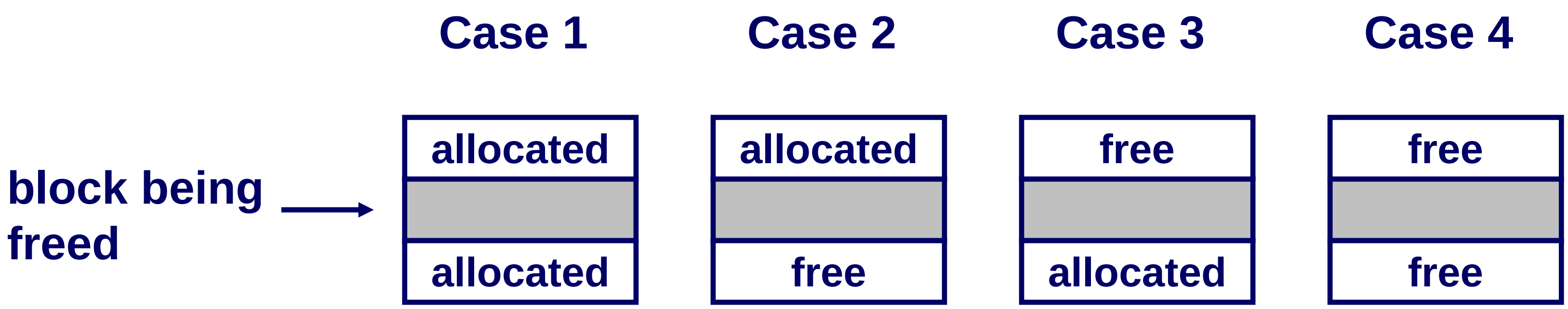
Boundary tags **[Knuth73]**

- **Replicate size/allocated word at bottom of free blocks**
- **Allows us to traverse the “list” backwards, but requires extra space**
- **Important and general technique!**

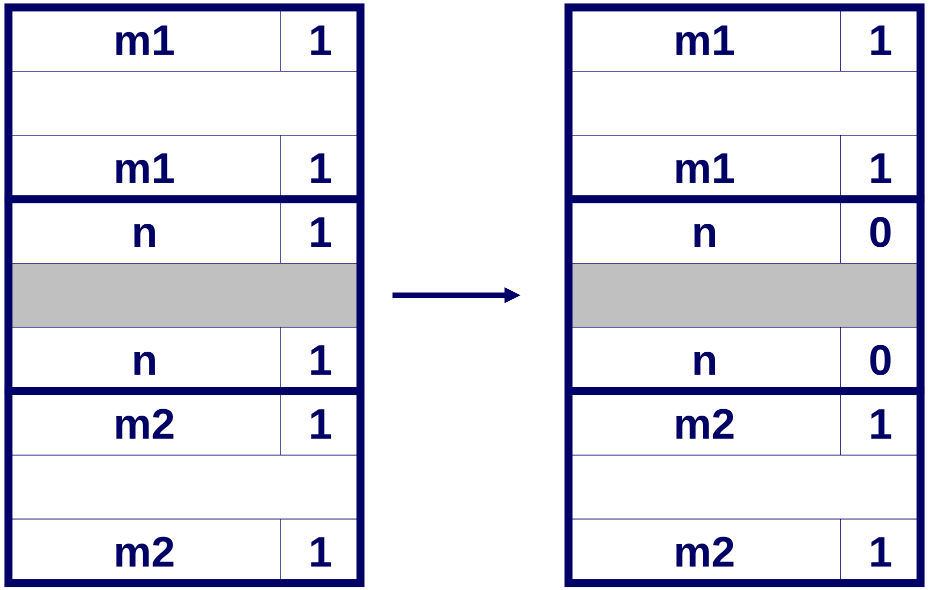




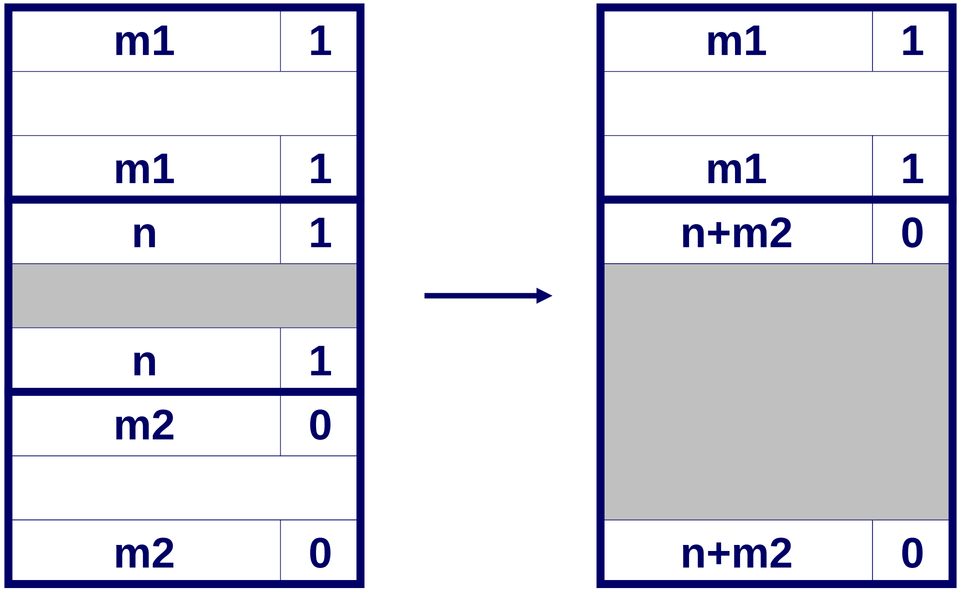
Constant Time Coalescing



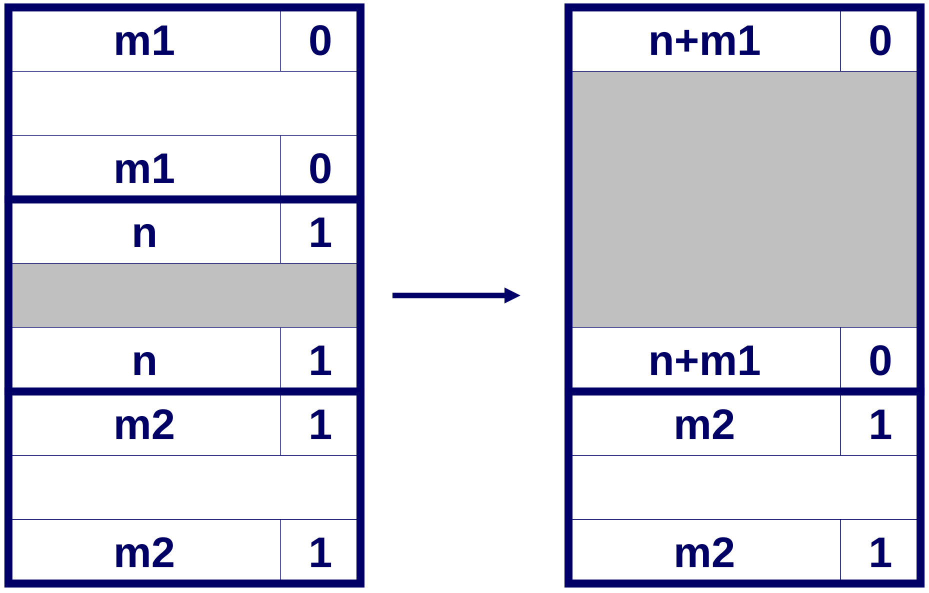
Constant Time Coalescing (Case 1)



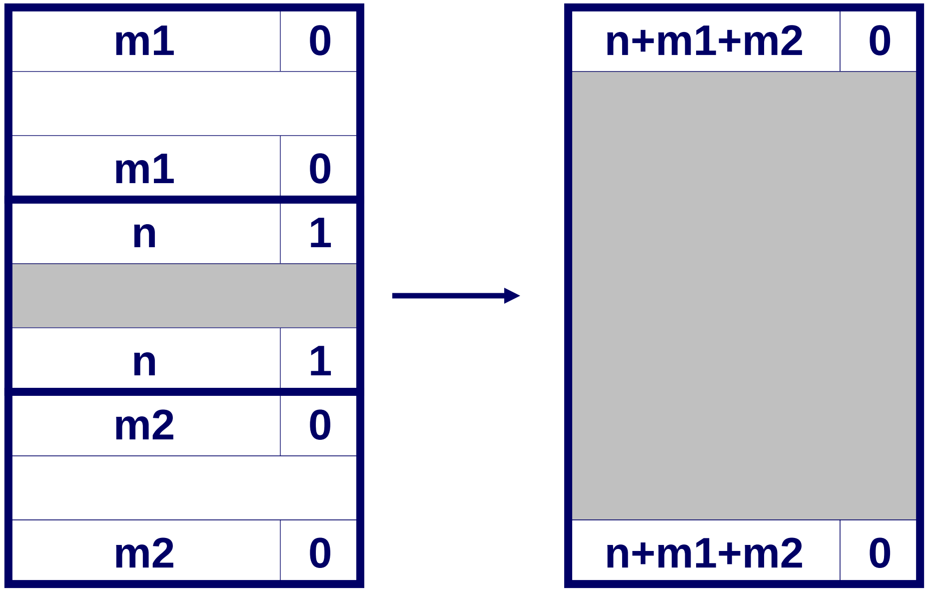
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Summary of Key Allocator Policies

Placement policy:

- First fit, next fit, best fit, etc.
- Trades off lower throughput for less fragmentation
  - *Interesting observation*: segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list.

Splitting policy:

- When do we go ahead and split free blocks?
- How much internal fragmentation are we willing to tolerate?

Coalescing policy:

- *Immediate coalescing:* coalesce each time `free` is called
- *Deferred coalescing:* try to improve performance of `free` by deferring coalescing until needed. e.g.,
  - Coalesce as you scan the free list for malloc.
  - Coalesce when the amount of external fragmentation reaches some threshold.

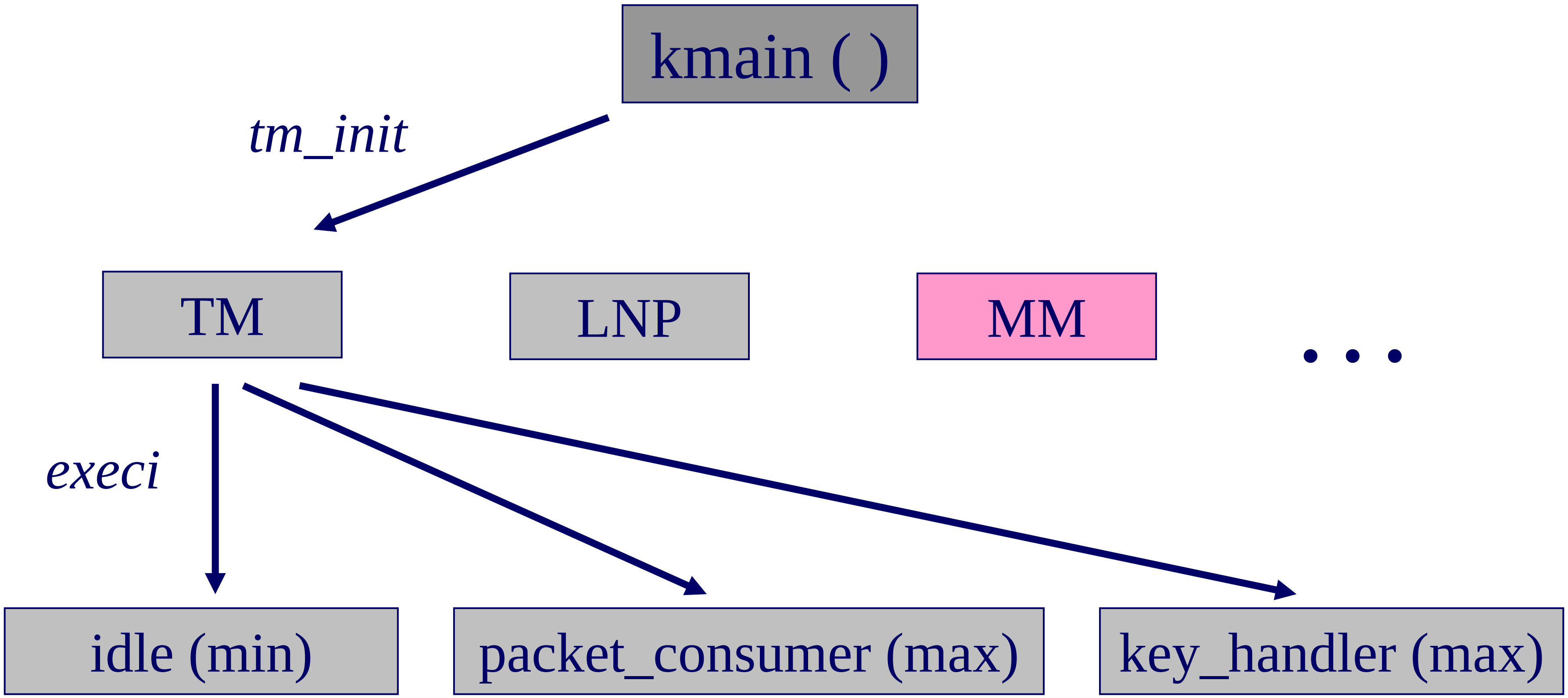
Implicit Lists: Summary

- **Implementation:** **very simple**
- **Allocate cost:** **linear time worst case**
- **Free cost:** **constant time worst case -- even with coalescing**
- **Memory usage:** **will depend on placement policy**
  - **First fit, next fit or best fit**

Not used in practice for `malloc/free` because of linear time allocate. Used in many special purpose applications.

However, the concepts of splitting and boundary tag coalescing are general to *all* allocators.

BrickOS Memory Management (mm.c)



BrickOS mm.c

```
#include <sys/mm.h>

#ifdef CONF_MM
#include <stdlib.h>
#include <sys/tm.h>
#include <sys/critsec.h>
#include <string.h>
////////////////////////////////////////////////////////////////////////////////
// Variables
////////////////////////////////////////////////////////////////////////////////
size_t  *mm_first_free;         //!< first free block

#ifndef CONF_TM
typedef size_t tid_t;           //! dummy process ID type
//!  current process ID
/*!  we need a non-null, non-0xffff current pid even if there is no task
  management. */
const tid_t ctid=0x0001;
#endif
```

BrickOS mm.c

```
///////////////////////////////////////////////////////////////////////////////
// Functions
///////////////////////////////////////////////////////////////////////////////
// memory block structure:
// 0 1          : pid of owner (0=empty)
// 2 3          : size of data block >> 1  (n 16-bit words)
// 4 ... 4+2n-1 : data

//! check for free blocks after this one and join them if possible
/* \param ptr pointer to size field of current block
   \return size of block */
size_t mm_try_join(size_t *ptr) {
  size_t *next=ptr+*ptr+1;
  size_t increase=0;

  while(*next==MM_FREE && next>=&mm_start) {
    increase+=*(next+1) + MM_HEADER_SIZE;
    next    +=*(next+1) + MM_HEADER_SIZE;
  }
  return (*ptr)+=increase;
}
```

BrickOS mm.c

```
//! defragment free blocks
/*! use mm_try_join on each free block of memory
*/
void mm_defrag() {
  size_t *ptr = &mm_start;
#ifdef CONF_TM
  ENTER_KERNEL_CRITICAL_SECTION();
#endif
  while(ptr >= &mm_start) {
    if(*ptr == MM_FREE)
      mm_try_join(ptr+1);
    ptr += *(ptr+1);
    ptr += MM_HEADER_SIZE;
  }
#ifdef CONF_TM
  LEAVE_KERNEL_CRITICAL_SECTION();
#endif
}
```

BrickOS mm.c

```
//! update first free block pointer
/*! \param start pointer to owner field of a memory block to start with.
*/
void mm_update_first_free(size_t *start) {
  size_t *ptr=start;

  while((*ptr!=MM_FREE) && (ptr>=&mm_start))
    ptr+=*(ptr+1)+MM_HEADER_SIZE;

  mm_first_free=ptr;
}
```

BrickOS mm.c

```
//! initialize memory management
void mm_init() {
  size_t *current,*next;

  current=&mm_start;

  // memory layout
  MM_BLOCK_FREE    (&mm_start);   // ram
                                   // something at 0xc000 ?
  MM_BLOCK_RESERVED(0xef30);      // lcddata
  MM_BLOCK_FREE    (0xef50);      // ram2
  MM_BLOCK_RESERVED(0xf000);      // motor
  MM_BLOCK_FREE    (0xfe00);      // ram4
  MM_BLOCK_RESERVED(0xff00);      // stack, onchip

  // expand last block to encompass all available memory
  *current=(int)(((-(int) current)-2)>>1);

  mm_update_first_free(&mm_start);
}
```

BrickOS malloc( )

```
//! allocate a block of memory
/*! \param size requested block size, return 0 on error, else pointer to block.*/
void *malloc(size_t size) {
  size_t *ptr,*next;
  size=(size+1)>>1;       // only multiples of 2
#ifdef CONF_TM
  ENTER_KERNEL_CRITICAL_SECTION();
#endif
  ptr=mm_first_free;
  while(ptr>=&mm_start) {
    if(*(ptr++)==MM_FREE) {     // free block?
#ifdef CONF_TM
      mm_try_join(ptr);   // unite with later blocks
#endif
      if(*ptr>=size) {    // big enough?
          *(ptr-1)=(size_t)ctid;  // set owner
        if((*ptr-size)>=MM_SPLIT_THRESH) {
          next=ptr+size+1;
          *(next++)=MM_FREE;
          *(next)=*ptr-size-MM_HEADER_SIZE;
          mm_try_join(next);
```

```
          *ptr=size;
        }
        // was it the first free one?
        if(ptr==mm_first_free+1)
          mm_update_first_free(ptr+*ptr+1);
#ifdef CONF_TM
        LEAVE_KERNEL_CRITICAL_SECTION();
#endif
        return (void*) (ptr+1);
      }
    }
    ptr+=(*ptr)+1;        // find next block.
  }
#ifdef CONF_TM
  LEAVE_KERNEL_CRITICAL_SECTION();
#endif
  return NULL;
}
```

```
//! free a previously allocated block of memory.
void free(void *the_ptr) {
    size_t *ptr=the_ptr;
#ifndef CONF_TM
        size_t *p2,*next;
#endif
  if(ptr==NULL || (((size_t)ptr)&1) )
    return;
  ptr-=MM_HEADER_SIZE;
  *((size_t*) ptr)=MM_FREE;     // mark as free
#ifdef CONF_TM
  // for task-safe ops, free must be atomic and nonblocking, update mm_first_free
  if(ptr<mm_first_free || mm_first_free<&mm_start)
    mm_first_free=ptr;          // update mm_first_free
#else // without task management, we have time to unite neighboring mem. blocks.
  p2=&mm_start;
  while(p2!=ptr) {              // we could make free
    next=p2+*(p2+1)+MM_HEADER_SIZE;   // O(1) if we included
    if(*p2==MM_FREE && next==ptr)   // a pointer to the
      break;        // previous block.
    p2=next;        // I don't want to.
  }
  mm_try_join(p2+1);        // defragment free areas
  if(ptr<mm_first_free || mm_first_free<&mm_start)
    mm_update_first_free(ptr);    // update mm_first_free
#endif
```

```c
//! allocate adjacent blocks of memory
/*! \param nmemb number of blocks (must be > 0)
    \param size  individual block size (must be >0)
    \return 0 on error, else pointer to block
*/
void *calloc(size_t nmemb, size_t size) {
  void *ptr;
  size_t original_size = size;

  if (nmemb == 0 || size == 0)
    return 0;

  size*=nmemb;

  // if an overflow occurred, size/nmemb will not equal original_size
  if (size/nmemb != original_size)
    return 0;

  if((ptr=malloc(size))!=NULL)
    memset(ptr,0,size);

  return ptr;
}
```

```
//! free all blocks allocated by the current process (with TID = ctid).
/*! called by exit() and kmain().
*/
void mm_reaper() {
  size_t *ptr;

  // pass 1: mark as free
  ptr=&mm_start;
  while(ptr>=&mm_start) {
    if(*ptr==(size_t)ctid)
      *ptr=MM_FREE;
    ptr+=*(ptr+1)+MM_HEADER_SIZE;
  }

  // pass 2: defragment free areas
  // this may alter free blocks
  mm_defrag();
}
```

```c
//! return the number of bytes of unallocated memory
int mm_free_mem(void) {
  int free = 0;
  size_t *ptr;

#ifdef CONF_TM
  ENTER_KERNEL_CRITICAL_SECTION();
#endif

  // Iterate through the free list
  for (ptr = mm_first_free; ptr >= &mm_start; ptr += *(ptr+1) + MM_HEADER_SIZE)
    free += *(ptr+1);

#ifdef CONF_TM
  LEAVE_KERNEL_CRITICAL_SECTION();
#endif
  return free*2;
}
#endif
```