Operating Systems

Memory

1

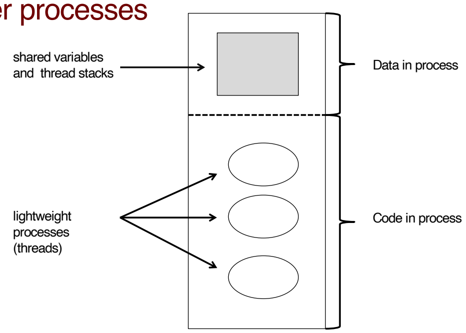
Memory

- Management of memory is a key OS task
- Competing demands
- Limited resource
- Mismanagement can be fatal

2

(Heavyweight) Process and Threads

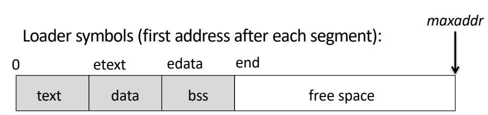
- Separate virtual address space provides protection from other processes
- Address space:



3

Segments and Regions

- Memory segments (as we have discussed)
  - Text segment
    - Starts at address 0 and contains compiled code
  - Data segment
    - Initialized global variables
  - Bss segment
    - "Block started by symbol" (from PDP-11), uninitialized globals
  - Free space



4

Xinu Dynamic Memory Allocation

- The initially free space is used for dynamic allocation
  - Single resource for all dynamic allocations which can be exhausted
- Stacks – dynamically allocated by the system from the higher addresses
  - Calls: *getstk()*, *freestk()*
- Heap – dynamic allocation by processes from the lower addresses
  - Calls: *getmem()*, *freemem()*

text	data	bss	heap	free	stack3	stack2	stack1

Memory Persistence

- Process creation routines *create* and *kill* manage stack space
  - Maintenance of free space is guaranteed
- Heap space (from *getmem*) cannot be automatically released
  - The responsibility of the program to release it before exiting
  - Or not (Xinu's sharing of all memory means that allocations will persist)
- Available heap space may become fragmented into small pieces
  - Effectively exhausting the resources

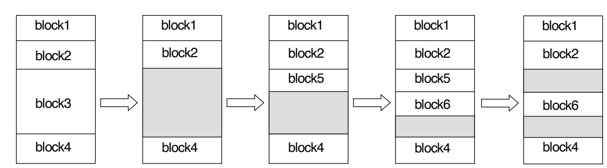
Keeping Track of Free Memory

- A memory manager must maintain a list of free blocks
- At allocation time, search the list and allocate a block of the requested size
  - When freed, put it back in the free list
- Coalesce
  - Increase length
  - Merge blocks if newly-freed block fills (empties) the gap

Block	Address	Length
1	0x84F800	4096
2	0x850F70	8192
3	0x8A03F0	8192
4	0x8C01D0	4096

Allocation

- Allocation and deallocation leads to "holes"
- Even coalescing adjacent holes, it is possible that a request can't be fulfilled, even though sufficient memory exists
  - Relocation is possible, but a challenge (coming soon)

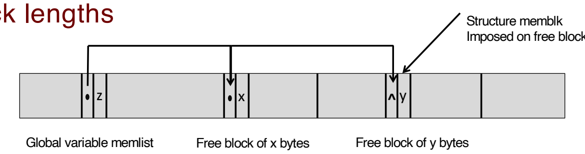


Storage Allocation Problem

- How to satisfy a request of size *n* from a list of free holes?
- First-fit - allocate the first hole that is big enough
- Last-fit – allocate the last hole that is big enough
- Best-fit - allocate the smallest hole that is big enough
  - Must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- Worst-fit - allocate from the largest hole
  - must also search entire list
  - Produces the largest leftover hole
- Studies show that first-fit and best-fit better than worst-fit in terms of speed and storage utilization

Implementation

- Xinu uses a standard approach of storing the free list “in place”, using the free memory itself to hold the list
- The global *memlist* has a pointer to the first free block
- struct *memblk* has a pointer and a length
- In the head (*memlist*), the length is the total of all block lengths



Size Considerations

- Since each free block stores a *memblk* when not used, the minimum block size is 8 bytes
- All requests are rounded to a multiple of 8 bytes
  - This ensures that no free block will ever be less than 8 bytes
  - *roundmb* and *truncmb* manage the sizes – truncmb is used initially on the free space

Xinu’s Implementation of Low-Level Memory Management

```
/* memory.h - roundmb, truncmb, freestk */

#define PAGE_SIZE       4096
#define MAXADDR         0x01000000  /* 16MB of RAM */


/*----------------------------------------------------------------------
 * roundmb, truncmb - round or truncate address to memory block size
 *----------------------------------------------------------------------
 */
#define roundmb(x)  (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)  (char *)( ((uint32)(x)) & (~7) )
```

```
/*------------------------------------------------------------------------
 *  freestk  --  free stack memory allocated by getstk
 *------------------------------------------------------------------------
 */
#define freestk(p,len) freemem((char *)((uint32)(p)      \
            - ((uint32)roundmb(len))  \
            + (uint32)sizeof(uint32)),\
            (uint32)roundmb(len) )

struct  memblk  {              /* see roundmb & truncmb         */
   struct   memblk *mnext;    /* ptr to next free memory blk    */
   uint32   mlength;          /* size of blk (includes memblk) */
   };
extern  struct  memblk  memlist;   /* head of free memory list   */
extern  void    *maxheap;      /* max free memory address*/
extern  void    *minheap;      /* address beyond loaded memory   */

/* added by linker */

extern  int end;           /* end of program     */
extern  int edata;         /* end of data segment    */
extern  int etext;         /* end of text segment    */
```

13

Allocating Heap Storage

```
/* getmem.c - getmem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getmem  -  Allocate heap storage, returning lowest word address
 *             of selected block using first-fit allocation
 *------------------------------------------------------------------------
 */
char    *getmem(
     uint32 nbytes          /* size of memory requested   */
   )
{
   intmask  mask;          /* saved interrupt mask       */
   struct   memblk  *prev, *curr, *leftover;
```

14

```
   mask = disable();
   if (nbytes == 0) {
    restore(mask);
    return (char *)SYSERR;
   }

   nbytes = (uint32) roundmb(nbytes);     /* use memblk multiples   */

   prev = &memlist;
   curr = memlist.mnext;
   while (curr != NULL) {                 /* search free list   */

        if (curr->mlength == nbytes) {    /* block is exact match   */
            prev->mnext = curr->mnext;
            memlist.mlength -= nbytes;
            restore(mask);
            return (char *)(curr);

        }
```

15

```
   else if (curr->mlength > nbytes) { /* split big block*/
        leftover = (struct memblk *)((uint32) curr + nbytes);
        prev->mnext = leftover;
        leftover->mnext = curr->mnext;
        leftover->mlength = curr->mlength - nbytes;
        memlist.mlength -= nbytes;
        restore(mask);
        return (char *)(curr);
    } else {                           /* move to next block */
        prev = curr;
        curr = curr->mnext;
    }
   }
   restore(mask);
   return (char *)SYSERR;
}
```

16

Allocating Stack Storage

- *getstk* is called when a process is created
- The free list is ordered by address and the goal is to allocate the highest available block
  - Of sufficient size
  - last-fit strategy
- Thus getstk must search the entire list
  - As it goes, it records the address of blocks that satisfy the the request (size)
  - Since the list is singly linked, it stores a pointer to the block and to its predecessor
- When the search completes, *fits* points to the highest-addressed block

Allocating Stack Storage

```
/* getstk.c - getstk */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getstk  -  Allocate stack memory, returning highest word address
 *------------------------------------------------------------------------
 */
char    *getstk(
      uint32 nbytes                     /* size of memory requested   */
    )
{
   intmask mask;                        /* saved interrupt mask       */
   struct  memblk  *prev, *curr;      /* walk through memory list   */
   struct  memblk  *fits, *fitsprev; /* record block that fits      */
```



```
    mask = disable();
    if (nbytes == 0) {
         restore(mask);
         return (char *)SYSERR;
    }

    nbytes = (uint32) roundmb(nbytes); /* use mblock multiples    */

    prev = &memlist;
    curr = memlist.mnext;
    fits = NULL;
    fitsprev = NULL;                /* to avoid a compiler warning */

    while (curr != NULL) {             /* scan entire list        */
       if (curr->mlength >= nbytes) {  /* record block address    */
           fits = curr;                /*   when request fits     */
           fitsprev = prev;
       }
     prev = curr;
     curr = curr->mnext;
    }
```



```
if (fits == NULL) {           /* no block was found */
    restore(mask);
    return (char *)SYSERR;
}

if (nbytes == fits->mlength) {       /* block is exact match   */
    fitsprev->mnext = fits->mnext;
} else {                             /* remove top section */
    fits->mlength -= nbytes;
    fits = (struct memblk *)((uint32)fits + fits->mlength);
}

memlist.mlength -= nbytes;
restore(mask);
return (char *)((uint32) fits + nbytes - sizeof(uint32));
}
```

Releasing Heap and Stack Storage

- When a process is finished with heap storage, it calls *freemem*
- Uses the block address to find the correct location in the list
- As mentioned, the memory manager must check to see if the newly freed block can be coalesced
  - Adjacent to the previous block, the next block or adjacent to both
- The differences in heap and stack are only relevant for allocation – the logic to free is the same so *freestk* is an inline for *freemem*

Releasing Heap and Stack Storage

```
/* freemem.c - freemem */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  freemem  -  Free a memory block, returning the block to the free list
 *------------------------------------------------------------------------
 */
syscall freemem(
     char      *blkaddr,  /* pointer to memory block*/
     uint32     nbytes    /* size of block in bytes */
   )
{
   intmask  mask;          /* saved interrupt mask   */
   struct   memblk *next, *prev, *block;
   uint32   top;
```



```
	mask = disable();
	if ((nbytes == 0) || ((uint32) blkaddr < (uint32) minheap)
	       || ((uint32) blkaddr > (uint32) maxheap)) {
	 restore(mask);
	 return SYSERR;
	}

	nbytes = (uint32) roundmb(nbytes); /* use memblk multiples   */
	block = (struct memblk *)blkaddr;

	prev = &memlist;                  /* walk along free list   */
	next = memlist.mnext;
	while ((next != NULL) && (next < block)) {
	    prev = next;
	    next = next->mnext;
	}

	if (prev == &memlist) {    /* compute top of previous block*/
	    top = (uint32) NULL;
	} else {
	    top = (uint32) prev + prev->mlength;
	}
```



```
	/* Insure new block does not overlap previous or next blocks */

	if (((prev != &memlist) && (uint32) block < top)
	    || ((next != NULL)  && (uint32) block+nbytes>(uint32)next)) {
	    restore(mask);
	    return SYSERR;
	}

	memlist.mlength += nbytes;

	/* Either coalesce with previous block or add to free list */

	if (top == (uint32) block) {   /* coalesce with previous block   */
	    prev->mlength += nbytes;
	    block = prev;
	} else {                       /* link into list as new node */
	    block->mnext = next;
	    block->mlength = nbytes;
	    prev->mnext = block;
	}
```

```
/* Coalesce with next block if adjacent */

   if (((uint32) block + block->mlength) == (uint32) next) {
       block->mlength += next->mlength;
       block->mnext = next->mnext;
   }
   restore(mask);
   return OK;
}
```


Xinu Memory Allocation Summary

- Memory allocation of arbitrary blocks provides flexibility but potentially leads to fragmentation
  - Known as **external fragmentation**
- Managing memory in fixed-size blocks addresses that but has other issues
  - One is when a block is not fully utilized – known as **internal fragmentation**

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Pin (latch) job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Programs in Memory

- In general-purpose OSes, a program is generally loaded from disk into memory and placed within a process in order to be run
  - Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory takes many cycles
- Cache sits between main memory and CPU registers
- Protection of memory can help ensure correct operation

Types of Memory

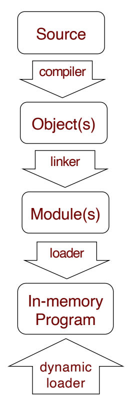
- Read-Only Memory (ROM)
  - Program code, constants
- Random Access Memory (RAM)
  - Changes during execution
- Dynamic RAM (DRAM)
  - slower, cheaper
- Static RAM (SRAM)
  - faster, more expensive
- Content Addressable Memory (CAM)
  - Indexed by value, rather than address
  - Useful for caches, more expensive than SRAM

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Creating a Runnable Program

- Source program compiled to object file at compile time
- Linking resolves references to symbols in other object files
- Loading relocates objects in memory, "fixing up" references
  - This may be performed in the linker
- Dynamic loading ensures system libraries are mapped and that the program can call them



Dynamic Loading

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required – can be implemented through program design

Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Logical vs. Physical Address Space

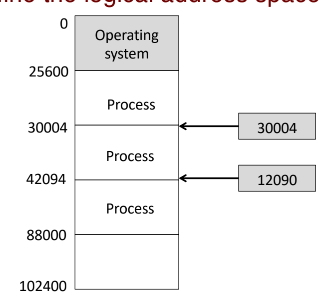
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Simplest example – a relocation register

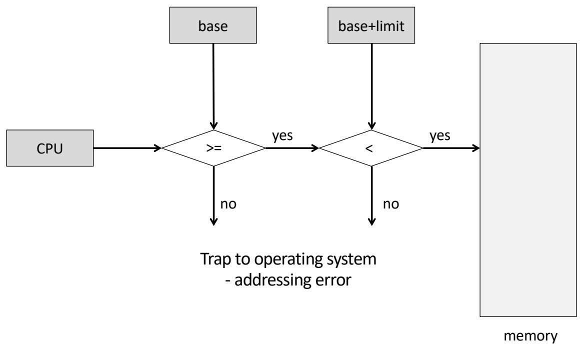
- A relocation register contains an offset to be added to each logical address generated by a program on the CPU to map it to a real location in physical memory
- Can be changed as part of the context switch so each process uses distinct physical memory

Base and Limit Registers

- A simple approach for logical addressing is to utilize a pair of registers (base and limit) to define the logical address space



HW address protection with base and limit registers



Memory Management Unit - MMU

- Hardware that maps virtual to physical addresses dynamically
- Using an MMU, a “relocation register” value can be added to every address generated by the CPU when memory is accessed
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
- To have more than one relocation value, a lookup table is needed to map a logical address to its relocation value, and thus its physical mapping
- In the MMU, this is generally constructed with Content Addressable Memory (CAM)

Memory Layout and Allocation

- Physical memory is often logically divided into two partitions
  - The operating system kernel memory is usually held in low memory with interrupt vector
  - User processes held in high memory
- Logical addressing is often the opposite
  - Userspace addresses start at 0x00 and the kernel is mapped into the upper addresses
- Per-process relocation mappings give each process a distinct logical address space
  - This protects user processes from one another
- Protection information can be used to prevent processes from changing operating-system code and data as well