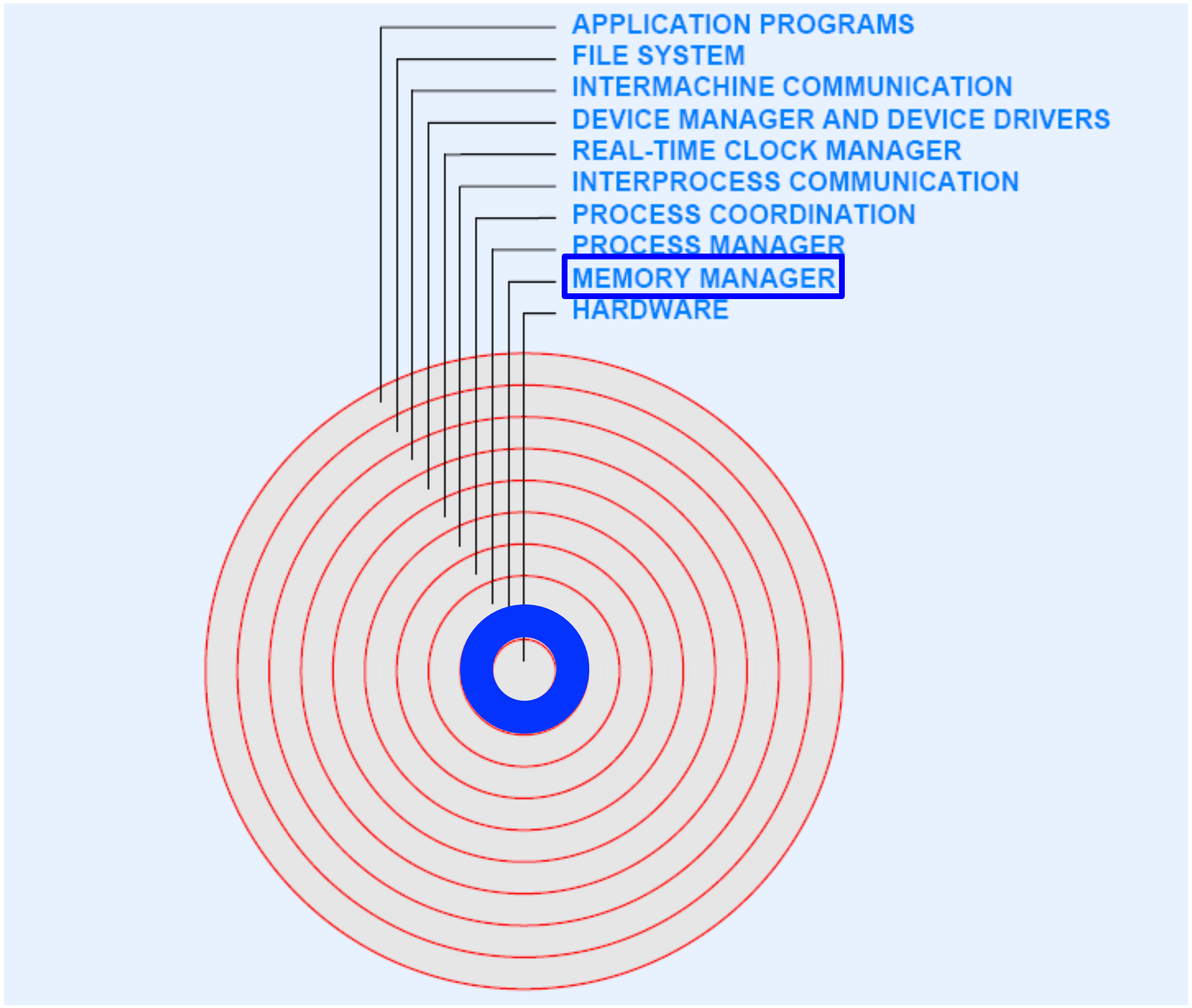
CSCI 8530
Advanced Operating Systems

Part 7

Low-level Memory Management

Location of Low-level Memory Management in the Hierarchy



Types Of Memory

- Main memory consists of a contiguous set of locations with addresses *0* through *N–1* (BeagleBone Black)
- Some systems have complex address spaces where certain regions are reserved (Galileo)
- Two categories of memory:
  - *Stable Storage*, such as *Flash memory* that retains values after the power is removed
  - *Random Access Memory* (*RAM*) that only retains values while the system is powered on
- Occupy separate locations

Apparent Impossibility of a Hierarchical OS Design

- Process manager uses memory manager to allocate space for process
- Memory manager uses device manager to page or swap to disk
- Device manager uses process manager to block and restart processes when they request I/O
- Solution: divide memory manager into two parts

Two Types of Memory Management

- Low-level memory manager
  - Manages memory within *kernel address space*
  - Allocates address spaces for processes
  - Treats memory as a single, exhaustible resource
  - Positioned in the hierarchy *below* process manager
- High-level memory manager
  - Manages pages within *address space*
  - Divides memory into abstract resources
  - Positioned in the hierarchy *above* device manager

Conceptual Uses of a Low-Level Memory Manager

- Allocation of *stack* space for a process
  - Performed by the process manager
  - Primitives needed to allocate and free stack space
  - Hold the activation record associated with each function the process invokes
- Allocation of *heap* storage
  - Performed by device manager (buffers) and other system facilities
  - Primitives needed to allocate and free heap space

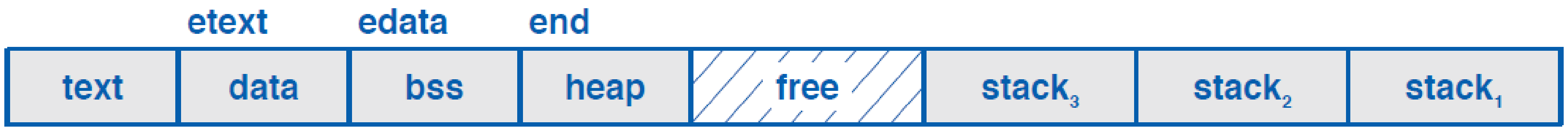


Illustration of memory after three processes have been created.

Xinu Low-Level Memory Manager

- Initialize the free memory list at startup: *meminit()*
- Two functions control allocation of *stack* storage

```
/* Allocate stack space when a process is created */
addr = getstk(numbytes);
/* Release a stack when a process terminates */
freestk(addr, numbytes);
```

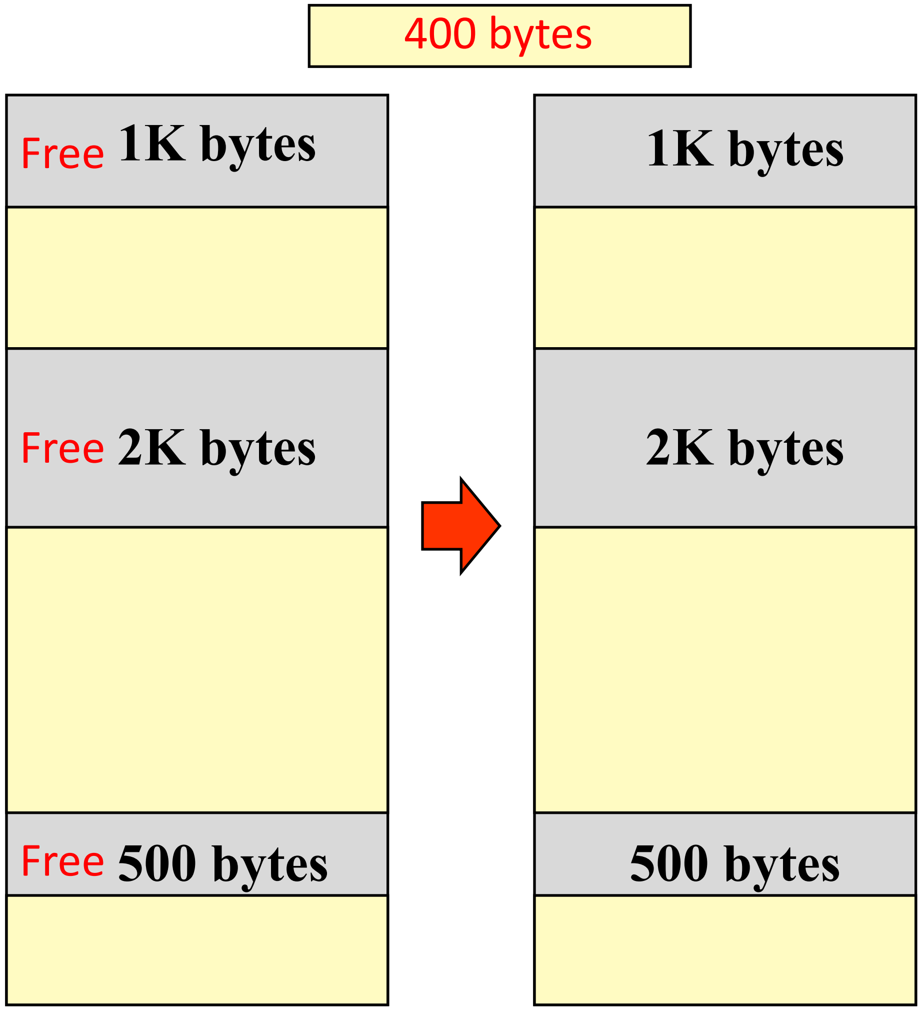
- Two functions control allocation of *heap* storage

```
/* Allocate heap storage on demand */
addr = getmem(numbytes);
/* Release heap storage as requested */
freemem(addr, numbytes);
```

- Memory allocated until none remains
- Only *getmem /freemem* are intended for use by application processes; *getstk / freestk* are restricted to the OS

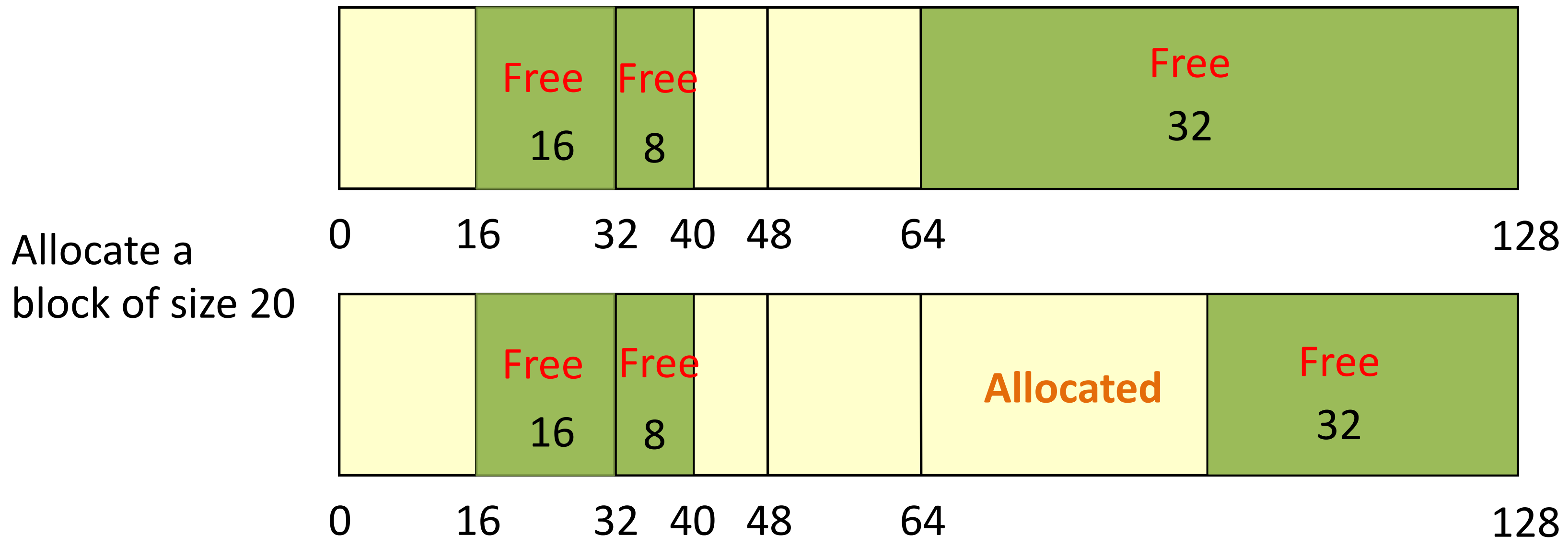
Possible Allocation Strategies (1/2)

- Stack and heap
  - Allocated from same free area
  - Allocated from separate free areas
- Single free list within an area
  - First-fit
  - Best-fit
  - Worst-fot
- Multiple free lists within an area
  - By exact size (static / dynamic)
  - By range



Possible Allocation Strategies (2/2)

- Hierarchical data structure (tree)
  - Binary size allocation (e.g. buddy system)
  - Other
- FIFO cache with above methods



Practical Considerations

- Sharing
  - Stack cannot be shared
  - Multiple processes may share access to a heap
- Persistence
  - Stack object is associated with one process
  - Heap object may persist longer than the process that created it
- Stack objects tend to be uniform size

Xinu Low-Level Allocation

- One free area (as a single resource)
- Single free list used for both heap and stack allocation
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to contain *all* free memory
- Allocation policy
  - Heap: first-fit
  - Stack: last-fit
  - Results in two conceptual pools

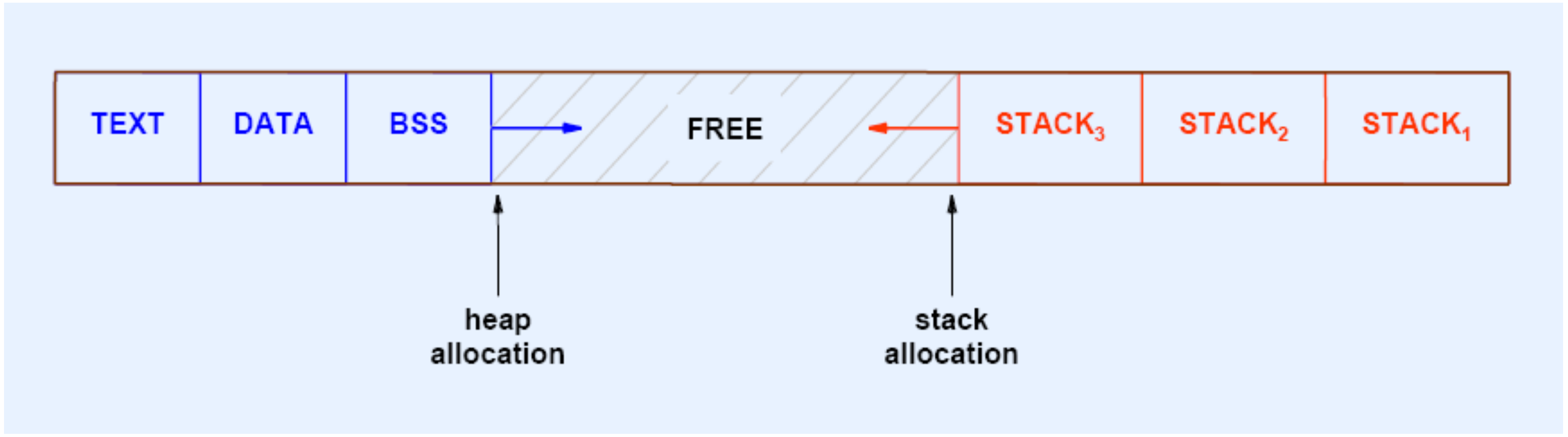
Allocating Heap Storage

- Allocate heap storage by finding a free block that is sufficient for the request (*getmem*)
- Use a *first-fit* allocation strategy by allocating the first block on the free list that satisfies a request
- Subtract the requested memory from the free block and adjusts the free list accordingly (*getmem*)

Allocating Stack Storage

- Allocate a block of memory for a process stack
- Search the entire list of free blocks because the free list is kept in order by memory address
- Record the address of any block that satisfies the request
  - The last recorded address points to the free block with the highest address that satisfies the request (*last-fit* strategy)
- Whenever a block is found that has sufficient size to satisfy the request, *getstk* sets variable *fits* points to the usable free block with the highest memory address

Result of Xinu Allocation Policy



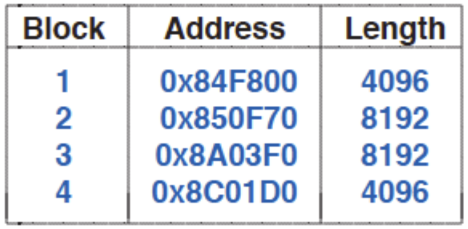
- First-fit allocates heap from lowest free address
- Last-fit allocates stack from highest free memory
- Uniform stack object size means higher probability of reuse

Memory Protection and Stack Overflow

- If memory management hardware supports protection
  - Assign protection bits to each stack
  - Set protection key during context switch
  - Hardware raises exception if stack overflows
- If no hardware protection available
  - Mark top of next stack
  - Check when scheduling
  - Provides partial protection against overflow

Memory Allocation Granularity

- Facts
  - Memory is byte addressable
  - Some hardware requires alignment
    - For process stack
    - For I /O buffers
    - For pointers
  - Free memory blocks are kept on free list
  - Cannot allocate / free individual bytes
- Solution: choose minimum granularity and round all requests



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

An example set of free memory blocks

Example Code to Round Memory Requests

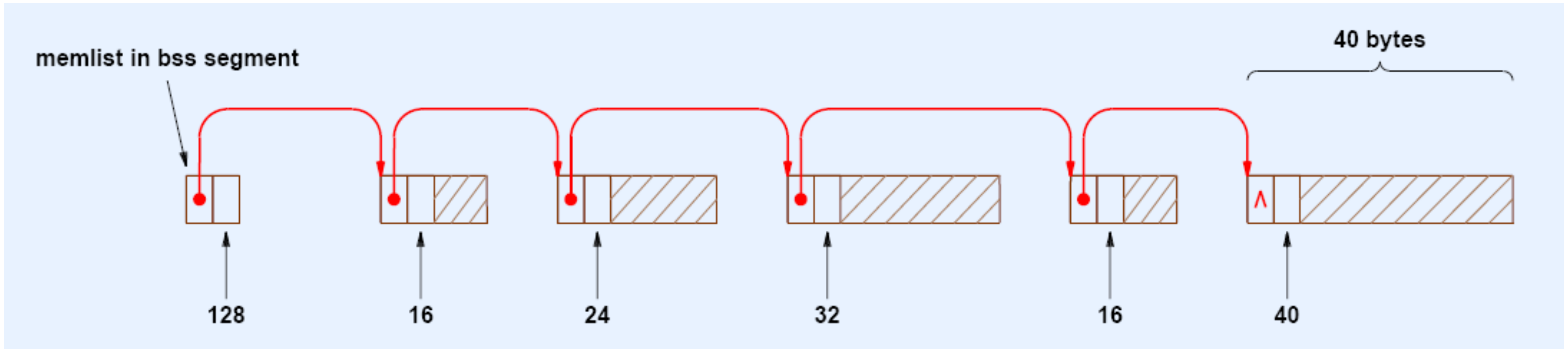
```
/* excerpt from memory.h */

/*----------------------------------------------------------------------
 * roundmb, truncmb - Round or truncate address to memory block size
 *----------------------------------------------------------------------
 */
```

Round or truncate address to memory block size

```
#define roundmb(x)      (char *)( (7 + (uint32)(x)) & (~7) )
#define truncmb(x)      (char *)( ((uint32)(x)) & (~7) )
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 *minheap;                  /* Start of heap                 */
extern void *maxheap;                  /* Highest valid heap address    */
```

- Note the efficient implementation
  - Size of *memblk* is chosen to be a power of 2
  - Bit manipulation used for rounding and truncation

Illustration of Xinu Free List



- Each memory block: a pointer and an integer
- Free memory blocks used to store list pointers
- Items on list ordered by increasing address
- All allocations rounded to size of struct *memblk*
- Length in *memlist* counts total free memory bytes

Rounding all requests to multiples of the memblk *structure ensures that each request satisfies the size constraint and guarantees that no free block will ever be too small to link into the free list.*

Allocation Technique

- Round up request to a multiple of memory blocks
- *Getmem* use two pointers to walk free memory list
- Choose either
  - First free block that is large enough (*getmem*)
  - Last free block that is large enough (*getstk*)
- If a free block is larger than the request, extract a piece for the request and leave the unused part on the free list
- Invariant used during search:
  - *curr* points to a node on the free list (or *NULL*)
  - *prev* points to previous node (or *memlist*)

Xinu Getmem (part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 * getmem - Allocate heap storage, returning lowest word address
 *------------------------------------------------------------------------
 */
char *getmem(
        uint32 nbytes                   /* Size of memory requested */
     )
{
        intmask mask;                   /* Saved interrupt mask     */
        struct memblk *prev, *curr, *leftover;

        mask = disable();
        if (nbytes == 0) {
                restore(mask);
                return (char *)SYSERR;
        }
        nbytes = (uint32) roundmb(nbytes); /* Use memblk multiples    */
```

Round the memory request

Xinu Getmem (part 2)

getmem compares the size of the current block to nbytes, the size of the request.

```
        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);
                } 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;
}
```

Deallocation Technique

- Round size to a multiple of memory blocks
- Walk the free list, using *next* to point to a block on the free list, and *prev* to point to the previous block (or *memlist*)
- Stop when the address of the block being freed lies between *prev* and *next*
- Either
  - Coalesce with the previous block if new block is contiguous
  - Add the new block to the free list
- Coalesce with the next block, if the result of the above is adjacent with the next block
- Coalescing helps avoid memory fragmentation

Xinu Freemem (1/3)

```
/* 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;
```

Run through the list of free blocks

Xinu Freemem (2/3)

Coalescing with the previous block

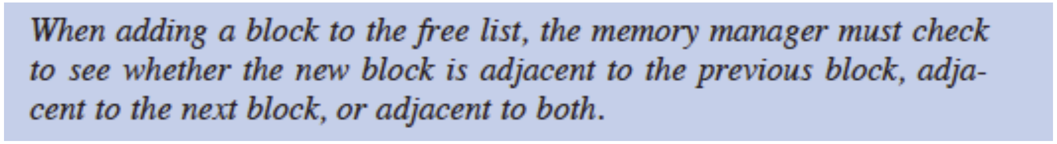
```
prev = &memlist;                        /* Walk along free list */
next = memlist.mnext;
while ((next != NULL) && (next < block)) {
        prev = next;
        next = next->mnext;
}
if (prev == &memlist) {         /* Compute top of previous block */
        top = (uint32) NULL;
} else {
        top = (uint32) prev + prev->mlength;
}

/* Ensure 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;
```

Find the address of the block being inserted

Xinu Freemem (3/3)

```
        /* 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;
}
```

Test whether the address is equal to the address of the next block.


When adding a block to the free list, the memory manager must check to see whether the new block is adjacent to the previous block, adjacent to the next block, or adjacent to both.

Xinu Getstk (1/2)

Invoked whenever a process is created.

```
/* 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;
```

Xinu Getstk (2/2)

```
        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));
}
```

Point to the predecessor of the free block

Point to a free block of memory

The last recorded address points to the free block with the highest address

Partition the block into two pieces

Xinu Freestk

```
/* excerpt from memory.h */

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

- Implemented as an inline function to invoke *freemem*
- Technique: convert address from the highest address in block being freed to the lowest address in the block, and call *freemem*

PROCESS CREATION AND TERMINATION

Process Creation

- Must
  - Search the process table for a free slot
  - Allocate a stack for the new process
  - Fill in process table entry
  - *Create* calls *getstk* to allocate space for a stack as if new process was "called"
    - Refer to the initial configuration as a pseudo call
- Design decisions
  - Choose an *initial state* for the process
  - Choose an action for the case where a process "returns" from the top-level function

Xinu Approach

- New process created in suspended state
  - Consequence: execution can only begin after process is resumed
- If process returns from top-level function, process should exit
- Implementation: set the return address in the pseudo-call
  - Initialized using constant *INITRET*
  - *INITRET* is defined to be function *userret*
  - Function userret causes current process to exit

Xinu Function Userret

```
/* userret.c - userret */

#include <xinu.h>

/*------------------------------------------------------------------------
 * userret - Called when a process returns from the top-level function
 *------------------------------------------------------------------------
 */
void userret(void)
{
        kill(getpid());                 /* Force process to exit */
}
```

1. *create.c* stores the address of *userret* in the stack location where a return address would appear (using the symbolic constant *INITRET*).
2. Place the return address field
3. *userret*.c terminate the current process by calling kill

Implementation of Killing a Process

- Implemented by function *kill*
- Remove a process from the system
- Action depends on state of process
  - If a process in the ready, sleeping, or waiting states is on a list, must remove it
  - If process is waiting on a semaphore, must adjust the semaphore count

Xinu Implementation of Kill (1/2)

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 * kill - Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
          pid32 pid                          /* ID of process to kill      */
        )
{
        intmask mask;                        /* Saved interrupt mask       */
        struct procent *prptr;               /* Ptr to process' table entry */
        int32 i;                             /* Index into descriptors     */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)
            || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--prcount <= 1) {               /* Last user process completes */
                xdone();
        }
```

decrements the count of active processes

Xinu Implementation of Kill (2/2)

```
        send(prptr->prparent, pid);
        freestk(prptr->prstkbase, prptr->prstklen);

        switch (prptr->prstate) {
        case PR_CURR:
                prptr->prstate = PR_FREE;       /* Suicide */
                resched();

        case PR_SLEEP:
        case PR_RECTIM:
                unsleep(pid);
                prptr->prstate = PR_FREE;
                break;

        case PR_WAIT:
                semtab[prptr->prsem].scount++;
                /* Fall through */

        case PR_READY:
                getitem(pid);           /* Remove from queue */
                /* Fall through */

        default:
                prptr->prstate = PR_FREE;
        }

        restore(mask);
        return OK;
}
```

free memory has been allocated for the process's stack.

The entry in the process table can be reused.

The process is waiting for a timer or receiving a message

The process is waiting on semaphore queue

kill removes the process from the ready list and then frees the process table entry

Kill and the Current Process

- Look carefully at the code
  - Step 1: free the process's stack
  - Step 2: Perform other actions
- Consider a current process killing itself
  - The call to *resched* occurs after the process's stack has been freed
  - Why does it work?

Xdone Function

```
/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 * xdone - Print system completion message as last process exits
 *------------------------------------------------------------------------
 */
void xdone(void)
{
        kprintf("\n\nAll user processes have completed.\n\n");
        halt();                                 /* Halt the processor    */
}
```

Process Creation

- Allocate a process table entry
- Allocate a *stack*
- Place values on the stack as if the top-level function was called (pseudo-call)
- Arrange saved state so context switch can switch to the process
- Details depend on calling conventions
- Consider example code for an x86

Process Creation on X86 (1/5)

```
/* create.c - create, newpid */

#include <xinu.h>

local int newpid();

/*------------------------------------------------------------------------
 * create - Create a process to start running a function on x86
 *------------------------------------------------------------------------
 */
pid32 create(
        void *funcaddr,          /* Address of the function       */
        uint32 ssize,            /* Stack size in words           */
        pri16 priority,          /* Process priority > 0          */
        char *name,              /* Name (for debugging)          */
        uint32 nargs,            /* Number of args that follow    */
        ...
      )
{
        uint32 savsp, *pushsp;
        intmask mask;            /* Interrupt mask                */
        pid32 pid;               /* Stores new process id         */
        struct procent *prptr;   /* Pointer to proc. table entry  */
        int32 i;
        uint32 *a;               /* Points to list of args        */
        uint32 *saddr;           /* Stack address                 */
```

Process Creation on X86 (2/5)

```
mask = disable();
if (ssize < MINSTK)
        ssize = MINSTK;
ssize = (uint32) roundmb(ssize);
if ( (priority < 1) || ((pid=newpid()) == SYSERR) ||
     ((saddr = (uint32 *)getstk(ssize)) == (uint32 *)SYSERR) ) {
        restore(mask);
        return SYSERR;
}

prcount++;
prptr = &proctab[pid];

/* Initialize process table entry for new process */
prptr->prstate = PR_SUSP;          /* Initial state is suspended */
prptr->prprio = priority;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;        /* PNMLEN: Length of process "name" */
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prsem = -1;                                /* Record semaphore ID */
prptr->prparent = (pid32)getpid();                /* ID of the creating process */
prptr->prhasmsg = FALSE;                          /* Message sent to this process */
```

Process Creation on X86 (3/5)

```
        /* Set up stdin, stdout, and stderr descriptors for the shell
*/
        prptr->prdesc[0] = CONSOLE;
        prptr->prdesc[1] = CONSOLE;
        prptr->prdesc[2] = CONSOLE;

        /* Initialize stack as if the process was called
*/
        *saddr = STACKMAGIC;
        savsp = (uint32)saddr;

        /* Push arguments */
        a = (uint32 *)(&nargs + 1);     /* Start of args
*/
        a += nargs -1;                  /* Last argument                */
        for ( ; nargs > 0 ; nargs--)    /* Machine dependent; copy args */
        *--saddr = *a--;                /* onto created process' stack  */
        *--saddr = (long)INITRET;       /* Push on return address       */
                              /* INITRET: Defined function name userret */
```

/* Marker for the top of a process stack (used to help detect overflow) */

Process Creation on X86 (4/5)

```
        /* The following entries on the stack must match what ctxsw        */
        /*   expects a saved process state to contain: ret address,        */
        /*   ebp, interrupt mask, flags, registers, and an old SP          */
        *--saddr = (long)funcaddr;        /* Make the stack look like it's */
                                          /*   half-way through a call to  */
                                          /*   ctxsw that "returns" to the */
                                          /*   new process                 */
        *--saddr = savsp;                 /* This will be register ebp     */
                                          /*   for process exit            */
        savsp = (uint32) saddr;           /* Start of frame for ctxsw      */
        *--saddr = 0x00000200;            /* New process runs with         */
                                          /*   interrupts enabled          */

        /* Basically, the following emulates an x86 "pushal" instruction    */
        *--saddr = 0;                     /* %eax */
        *--saddr = 0;                     /* %ecx */
        *--saddr = 0;                     /* %edx */
        *--saddr = 0;                     /* %ebx */
        *--saddr = 0;                     /* %esp; value filled in below   */
        pushsp = saddr;                   /* Remember this location        */
        *--saddr = savsp;                 /* %ebp (while finishing ctxsw)  */
        *--saddr = 0;                     /* %esi */
        *--saddr = 0;                     /* %edi */
        *pushsp = (unsigned long) (prptr->prstkptr = (char *)saddr);
        restore(mask);
        return pid;
}
```

Process Creation on X86 (5/5)

Search the process table for a free (i.e., unused) slot.

```
/*------------------------------------------------------------------------
 * newpid - Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local pid32 newpid(void)
{
        uint32 i;                       /* Iterate through all processes */
        static pid32 nextpid = 1;       /* Position in table to try or   */
                                        /* one beyond end of table       */

        /* Check all NPROC slots */

        for (i = 0; i < NPROC; i++) {
                nextpid %= NPROC;       /* Wrap around to beginning      */
                if (proctab[nextpid].prstate == PR_FREE) {
                        return nextpid++;
                } else {
                        nextpid++;
                }
        }
        return (pid32) SYSERR;
}
```

Summary (1/2)

- To preserve OS multi-level hierarchy, divide memory manager into two pieces
  - Low-level used in kernel to allocate address spaces
  - High-level used to handle abstractions of virtual memory and paging within an address space
- Two conceptual memory types in low-level piece
  - Storage for process stack
  - Heap storage
- Stack requests tend to repeat the same size

Summary (2/2)

- Xinu low-level memory manager
  - Places all free memory on a single list
  - Rounds all requests to multiples of *struct memblk*
  - Uses *first-fit* allocation for heap requests and *last-fit* allocation for stack requests
- Process creation and termination use the memory manager to allocate and free process stacks
- *Create* hand-crafts an initial stack as if the top-level function had been called; the stack includes a return address given by constant *INITRET*