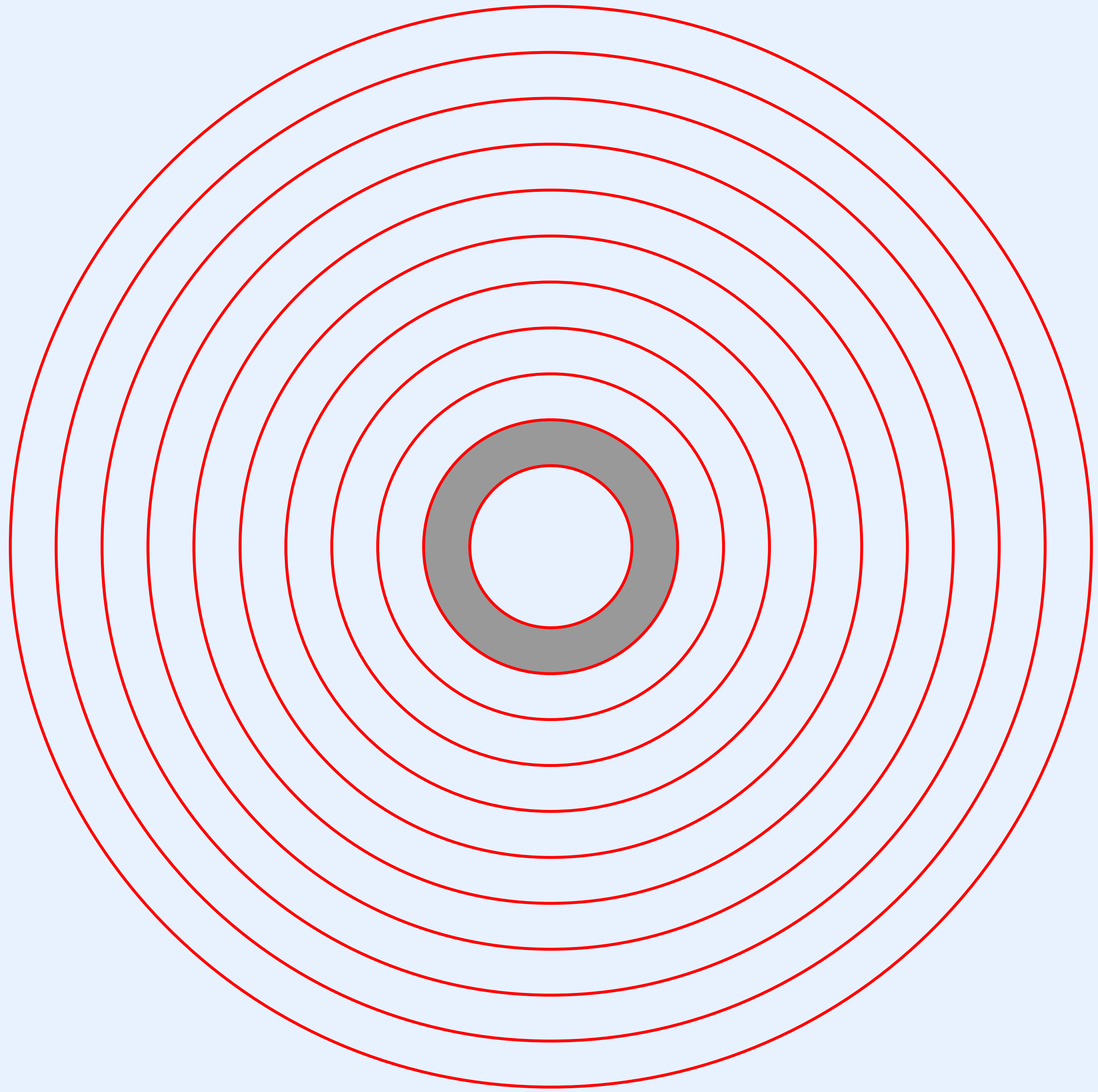
Module V

Low-Level Memory Management
Process Creation And Termination

Low-Level
Memory Management

Location Of Low-Level Memory Management In The Hierarchy



The Apparent Impossibility Of A Hierarchical OS Design

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

The Two Types Of Memory Management

- Low-level memory manager
  - Manages memory within the kernel address space
  - Used to allocate 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 a process’s address space
  - Positioned in the hierarchy above the device manager
  - Divides memory into abstract resources

Conceptual Uses Of A Low-Level Memory Manager

- Allocate stack space for a process
  - Performed by the process manager when a process is created
  - The memory manager must include functions to allocate and free stacks
- Allocation of heap storage
  - Performed by the device manager (buffers) and other system facilities
  - The memory manager must include functions to allocate and free heap space

The Xinu Low-Level Memory Manager

- Two functions control allocation of stack storage

```
addr = getstk(numbytes);

freestk(addr, numbytes);
```

- Two functions control allocation of heap storage

```
addr = getmem(numbytes);

freemem(addr, numbytes);
```

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

Well-Known Memory Allocation Strategies

- Stack and heap can be
  - Allocated from the same free area
  - Allocated from separate free areas
- The memory manager can use a single free list and follow a paradigm of
  - First-fit
  - Best-fit
  - The free list can be circular with a roving pointer
- The memory manager can maintain multiple free lists
  - By exact size (static / dynamic)
  - By range

Well-Known Memory Allocation Strategies
(continued)

- The free list can be kept in a hierarchical data structure (e.g., a tree)
  - Binary sizes of nodes can be used
  - Other sequences of sizes are also possible (e.g., Fibonacci)
- To handle repeated requests for the same size blocks, a cache can be combined with any of the above methods

Practical Considerations

- Sharing
  - A stack can never be shared
  - Multiple processes may share access to a given block allocated from the heap
- Persistence
  - A stack is associated with one process, and is freed when the process exists
  - An item allocated from a heap may persist longer than the process that created it
- Stacks tend to be one size, but heap requests vary in size
- Fragmentation can occur

Memory Fragmentation

- Can occur if processes allocate and then free arbitrary-size blocks
- Symptom: after many requests to allocate and free blocks of memory, small blocks of allocated memory exist between blocks of free memory
- The problem: although much of the memory is free, each block on the free list is small
- Example
  - Assume a free memory consists of 1 Gigabyte total
  - A process allocates 1024 blocks of one Megabyte each (1 Gigabyte)
  - The process then frees every other block
  - Although 512 Megabytes of free memory are available, the largest free block is only 1 Megabyte

The Xinu Low-Level Allocation Scheme

- All free memory is treated as one resource
- A single free list is used for both heap and stack allocation
- The free list is
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to contain *all* free memory
- The Xinu allocation policies
  - Heap allocation uses the first-fit approach
  - Stack allocation uses the last-fit approach
  - The design results in two conceptual pools of memory

Consequence Of The Xinu Allocation Policy

- The first-fit policy means heap storage is allocates from lowest part of free memory
- The last-fit policy means stack storage is allocated from the highest part of free memory
- Note: because stacks tend to be uniform size, there is higher probability of reuse and lower probability of fragmentation

Protecting Against Stack Overflow

- Note that the stack for a process can grow downward into the stack for another
- Some memory management hardware supports protection
  - The memory for a process stack is assigned the process's protection key
  - When a context switch occurs the processor protection key is set
  - If a process overflows its stack, hardware will raise an exception
- If no hardware protection is available
  - Mark the top of each stack with a reserved value
  - Check the value when scheduling
  - The approach provides a little 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
  - One cannot allocate / free individual bytes
- Solution: choose a minimum granularity and round all requests to the minimum

Example Code To Round Memory Requests

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

/*----------------------------------------------------------------------
 * 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) )

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
  - The size of *memblk* is chosen to be a power of 2
  - The code implements rounding and truncation with bit manipulation

The Xinu Free List

- Employs a well-known trick: to link together a list of free blocks, place all pointers *in the blocks themselves*
- Each block on the list contains
  - A pointer to the next block
  - An integer giving the size of the block
- A fixed location (*memlist* contains a pointer to the first block on the list
- Look again at the definitions in memory.h

Declarations For The Free List

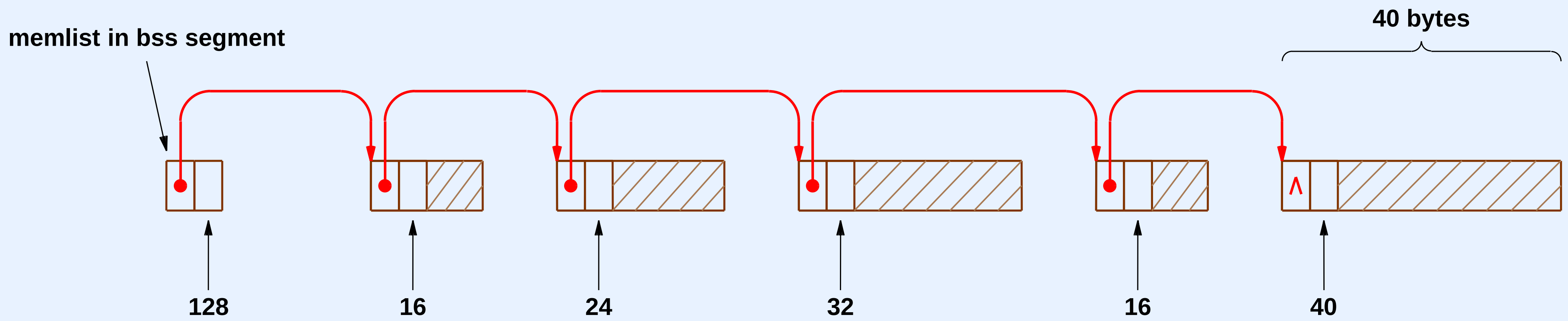
```
/* excerpt from memory.h */

/*----------------------------------------------------------------------
 * 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) )

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   */
```

- Struct *memblk* defines the two items stored in every block
- Variable *memlist* is the head of the free list
- Making the head of the list have the same structure as other nodes reduces special cases in the code

Illustration Of Xinu Free List



- Free memory blocks are used to store list pointers
- Items on the list are ordered by increasing address
- All allocations rounded to size of struct *memblk*
- The length in *memlist* counts total free memory bytes

Allocation Technique

- Round up the request to a multiple of memory blocks
- Walk the 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 part that is left over on the free list

When Searching The Free List

- Use two pointers that point to two successive nodes on the list
- An invariant is used during the search
  - Pointer *curr* points to a node on the free list (or *NULL*)
  - Pointer *prev* points to the previous node (or *memlist*)
- The invariant is established initially by making *prev* point to *memblk* and making *curr* point to the item to which *memblk* points
- The invariant must be maintained each time pointers move along the list

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 */
```

Xinu Getmem (Part 2)

```
	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;
}
```

Splitting A Block

- Occurs when *getmem* chooses a block that is larger then the requested size
- *Getmem* performs three steps
  - Compute the address of the piece that will be left over (i.e., the right-hand side of the block)
  - Link the leftover piece into the free list
  - Return the original block to the caller
- Note: the address of the leftover piece is curr + nbytes (the addition must be performed using unsigned arithmetic because the high-order bit may be on)

Deallocation Technique

- Round up the specified size to a multiple of memory blocks (allows the user to specify the same value during deallocation that was used during allocation)
- 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: insert the block into the list or handle coalescing

Coalescing Blocks

- The term *coalescing* refers to the opposite of splitting
- Coalescing occurs when a block being freed is adjacent to an existing free block
- Technique: instead of adding the new block to the list, combine the new and existing block into one larger block
- Note: the code must check for coalescing with the preceding block, the following block, or both

Xinu Freemem (Part 1)

```
/* 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;
```

Xinu Freemem (Part 2)

```
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;
```

Xinu Freemem (Part 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;
}
```

Xinu Getstk (Part 1)

```
/* 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 (Part 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));
}
```

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
- 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

- Process creation and termination use the memory manager
- Creation
  - Allocates a stack for the process being created
  - Fills in process table entry
  - Fills in the process’s stack to have a valid frame
- Two design decisions
  - Choose an initial state for the process
  - Choose an action for the case where a process “returns” from the top-level function

The Xinu Design

- The initial state of a new process
  - A process is created in the suspended state
  - Consequence: execution can only begin after the process is resumed
- Return from top-level function
  - Causes the process to exit (similar to Unix)
  - Implementation: place a “pseudo call” on the stack (make it appear that the top-level function in the process was called)
  - Initialize the return address in the pseudo call to *INITRET*
- Note: *INITRET* is defined to be function *userret*
- Function *userret* causes the 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 */
}
```

The Pseudo Call On An Initial Stack

- Seems straightforward
- Is actually extremely tricky
- The trick: arrange the stack as if the new process was stopped in a call to *ctxsw*
- Several details make it difficult
  - *Ctxsw* runs with interrupts disabled, but a new process should start with interrupts enabled
  - We must store arguments for the new process so that the top-level function receives them
- We will examine code for process creation after looking at process termination

Killing A Process

- Formally known as *process termination*
- The action taken depends on the state of the process
  - If a process is on a list, it must be removed
  - If a process is waiting on a semaphore, the semaphore count must be adjusted
- In Xinu, function *kill* implements process termination

Xinu Implementation Of Kill (Part 1)

```
/* 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();
        }
```

Xinu Implementation Of Kill (Part 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;
}
```

Killing The Current Process

- Look carefully at the code
  - Step 1: free the process's stack
  - Step 2: perform other actions
- Consider what happens when a current process kills itself: the call to *resched* occurs after the process's stack has been freed
- Why does it work?
- Answer: because in Xinu, even after stack has been freed, the memory is still available to the process

The Xdone Function

- Function *xdone* is called when the count of user processes reaches zero
- Nothing further will happen — only the null process remains running
- The function prints a warning message for the user

```
/* 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           */
}
```

The Steps For 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 the saved state on the stack so context switch can switch to the process
- Details depend on
  - The hardware and calling conventions
  - The way context switch is written
- Consider example code for ARM and x86 processors

Process Creation On ARM (Part 1)

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

#include <xinu.h>

local   int newpid();

#define roundew(x)      ( (x+3)& ~0x3)

/*------------------------------------------------------------------------
 *  create  -  create a process to start running a procedure
 *------------------------------------------------------------------------
 */
pid32   create(
          void          *procaddr,      /* procedure address            */
          uint32        ssize,          /* stack size in bytes          */
          pri16         priority,       /* process priority > 0         */
          char          *name,          /* name (for debugging)         */
          uint32        nargs,          /* number of args that follow   */
          ...
        )
{
        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 ARM (Part 2)

```
mask = disable();
if (ssize < MINSTK)
        ssize = MINSTK;
ssize = (uint32) roundew(ssize);
if (((saddr = (uint32 *)getstk(ssize)) ==
    (uint32 *)SYSERR ) ||
    (pid=newpid()) == SYSERR || priority < 1 ) {
        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;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prsem = -1;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;
```

Process Creation On ARM (Part 3)

```
        /* set up initial device descriptors for the shell              */
        prptr->prdesc[0] = CONSOLE;     /* stdin  is CONSOLE device     */
        prptr->prdesc[1] = CONSOLE;     /* stdout is CONSOLE device     */
        prptr->prdesc[2] = CONSOLE;     /* stderr is CONSOLE device     */

        /* Initialize stack as if the process was called                */

        *saddr = STACKMAGIC;

        /* push arguments */
        a = (uint32 *)(&nargs + 1);     /* start of args                */
        a += nargs -1;                  /* last argument                */
        for ( ; nargs > 4 ; nargs--)    /* machine dependent; copy args */
                *--saddr = *a--;        /* onto created process's stack */
        *--saddr = (long)procaddr;
        for(i = 11; i >= 4; i--)
                *--saddr = 0;
        for(i = 4; i > 0; i--) {
                if(i <= nargs)
                        *--saddr = *a--;
                else
                        *--saddr = 0;
        }
        *--saddr = (long)INITRET;       /* push on return address       */
        *--saddr = (long)0x00000053;    /* CPSR F bit set,              */
                                        /* Supervisor mode              */
        prptr->prstkptr = (char *)saddr;
        restore(mask);
        return pid;
}
```

Process Creation On ARM (Part 4)

```
/*------------------------------------------------------------------------
 *  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;
}
```

Process Creation On X86 (Part 1)

```
/* 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 bytes          */
          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 (Part 2)

```
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;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH; i++)
        ;
prptr->prsem = -1;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;
```

Process Creation On X86 (Part 3)

```
/* 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       */
```

Process Creation On X86 (Part 4)

```
	/* 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 (Part 5)

```
/*------------------------------------------------------------------------
 *  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;
}
```

An Assessment Of Process Creation

- Process creation code is among the most difficult code to understand
- One must know
  - The hardware architecture
  - The function calling conventions
  - The way *ctxsw* chooses to save state
  - How interrupts are handled
- As you struggle to understand it, imagine trying to write such code

Summary

- To preserve a multi-level hierarchy, the memory manager is divided into two pieces
  - A low-level manager is used in kernel to allocate address spaces
  - A high-level manager is used to handle abstractions of virtual memory and paging within a process's address space
- The Xinu low-level manager offers two types of allocation
  - Memory for a process stack
  - Memory from the heap
- Stack requests tend to repeat the same size

Summary
(continued)

- The 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* handcrafts an initial stack as if the top-level function had been called; the stack includes a return address given by constant *INITRET*

Questions?