Module VI

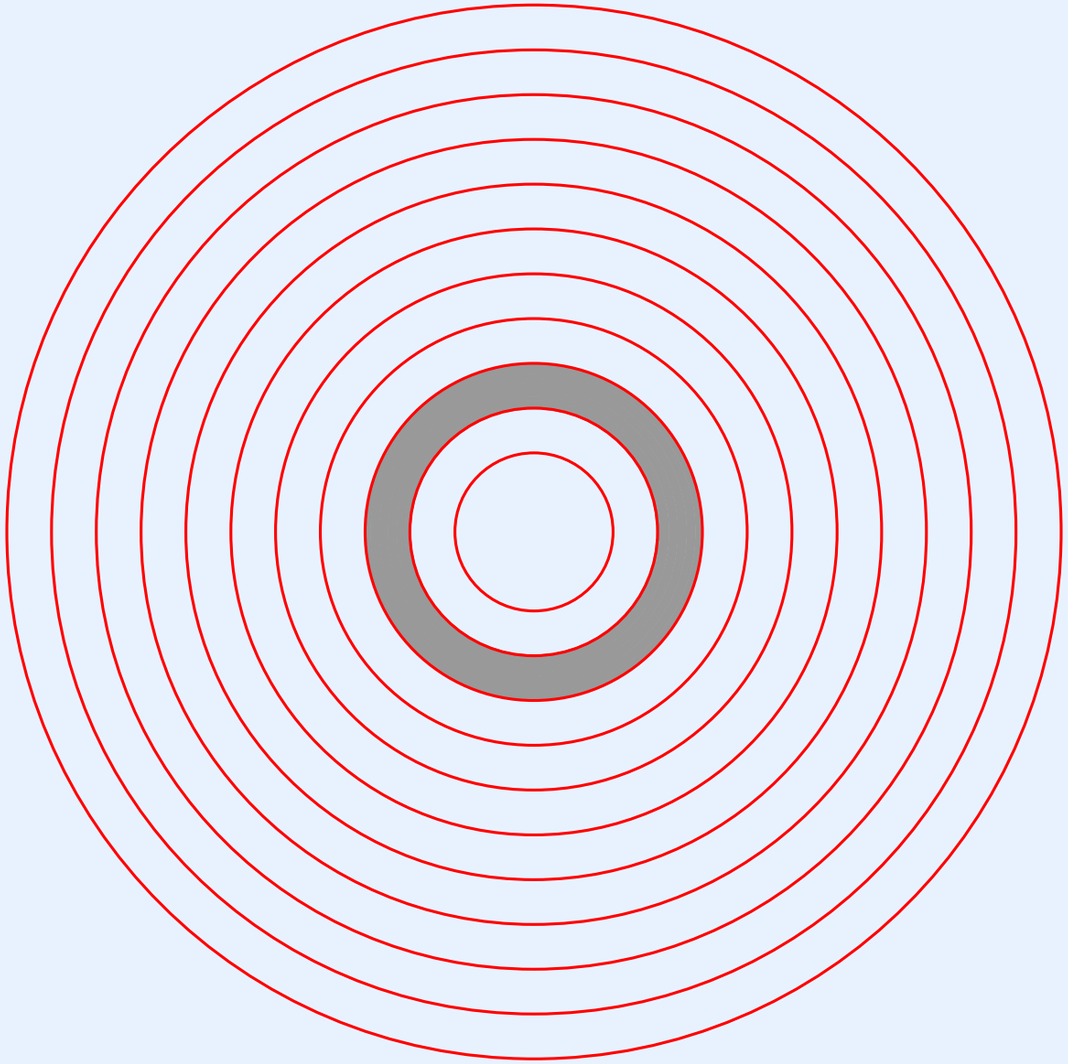
More Process Management:
Process Suspension/Resumption
Process Creation And Termination

Process Manipulation

- An OS needs system calls that can be used to control processes
- Example operations
  - Suspend a process (keep it from running)
  - Resume a previously-suspended process
  - Block a process to receive a message from another process
  - Send a message to another process
- The OS uses the process state variable to record the status of the process

Process Suspension And Resumption

Location Of Process Suspension And Resumption In The Hierarchy



Process Suspension And Resumption

- The idea
  - Temporarily “stop” a process
  - Allow the process to be resumed later
- Questions
  - What happens to the process while it is suspended?
  - Can a process be suspended at any time?
  - What happens if an attempt is made to resume a process that is not suspended?

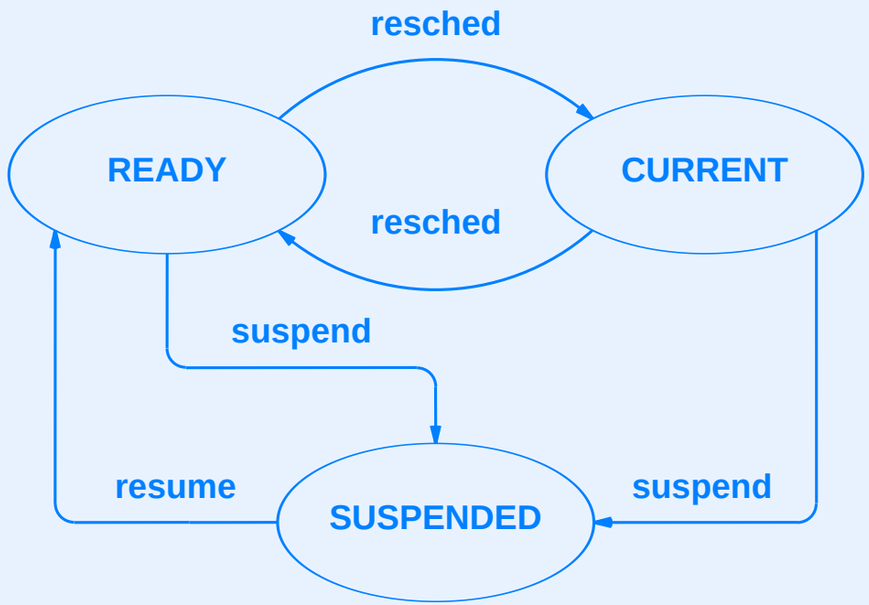
Steps In Suspension And Resumption

- Suspending a process simply means prohibiting the process from using the processor
- When suspending, the operating system must
  - Save pertinent information about the state of the process, such as where it is executing, the contents of general purpose registers, etc.
  - Set the state variable in the process table entry to indicate that the process is suspended
- When resuming, the operating system must
  - Allow the process to use the processor once again
  - Change the state to indicate that process is eligible

A State For Suspended Processes

- A suspended process is not ready, nor is it current
- Therefore, a new process state is needed
- The code uses constant *PR_SUSP* to indicate that a process is in the suspended state

State Transitions For Suspension And Resumption



- As the diagram shows, only a current or ready process can be suspended
- Only a suspended process can be resumed
- System calls *suspend* and *resume* handle the transitions

Suspended Processes

- Where is a process kept when it is suspended?
- Answer:
  - Unlike ready processes, there is no list of suspended processes
  - However, information about a suspended process remains in the process table
  - The process's stack remains allocated in memory

Suspending One's Self

- The currently executing process can suspend itself!
- Self-suspension is straightforward: just call

```
suspend(getpid())
```

- When *suspend* is asked to suspend the current process, it
  - Finds its entry in the process table, *proctab[currpid]*
  - Sets the state in its process table entry to *PR_SUSP*, indicating that it should be suspended
  - Calls *resched* to reschedule to another process

A Note About System Calls

- An operating system contains many functions that can be divided into two basic categories
  - Some functions are defined to be *system calls*, which means that applications can call them to access services
  - Other functions are merely internal functions used by other operating system functions
- We use the type *syscall* to distinguish system calls
- Notes
  - Xinu does not prohibit applications from making direct calls to internal operating system functions or referencing operating system variables
  - However, good programming practice restricts applications to system calls (e.g., use getpid() instead of referencing currpid)

Concurrent Execution Of System Calls

- Important concept: multiple processes can attempt to execute a given system call concurrently
- Concurrent execution can result in problems
  - Process A starts to change variables, such as process table entries
  - The OS switches to another process, B
  - When process B examines variables, they are inconsistent
- Even trivial operations can cause problems when performed concurrently

Preventing Concurrent Execution By Disabling Interrupts

- To prevent other processes from changing global data structures, a system call function can disable interrupts
- A later section of the course will explain interrupts; for now, it is sufficient to know that a system call must use two functions related to interrupts
  - Function *disable* is called to turn off hardware interrupts; the function returns a *mask* value that specifies whether interrupts were previously disabled or enabled
  - Function *restore* takes as an argument a mask value that was previously obtained from *disable*, and sets the hardware interrupt status according to the specified mask
- Basically, a system call uses *disable* upon being called, and uses *restore* just before it returns
- Note that *restore* must be called before *any* return
- The next slide illustrates the general structure of a system call

A Template For System Calls

```
syscall function_name( args )  {
        intmask mask;           /* interrupt mask*/
        mask = disable( );      /* disable interrupts at start of function*/
        if ( args are incorrect ) {
                restore(mask); /* restore interrupts before error return*/
                return SYSERR;
        }
        ... other processing ...
        if ( an error occurs ) {
                restore(mask); /* restore interrupts before error return*/
                return SYSERR;
        }
        ... more processing ...
        restore(mask);          /* restore interrupts before normal return*/
        return appropriate value ;
}
```

The Suspend System Call (Part 1)

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
          pid32         pid             /* ID of process to suspend     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process's table entry */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)) {
                restore(mask);
                return SYSERR;
        }
```

The Suspend System Call (Part 2)

```
        /* Only suspend a process that is current or ready */

        prptr = &proctab[pid];
        if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
                restore(mask);
                return SYSERR;
        }
        if (prptr->prstate == PR_READY) {
                getitem(pid);                   /* Remove a ready process   */
                                                /*   from the ready list    */
                prptr->prstate = PR_SUSP;
        } else {
                prptr->prstate = PR_SUSP;       /* Mark the current process */
                resched();                      /*   suspended and resched. */
        }
        prio = prptr->prprio;
        restore(mask);
        return prio;
}
```

Process Resumption

- The idea: resume execution of previously suspended process
- A detail: *resume* returns the priority of the resumed process
- Method
  - Make the process eligible to use the processor again
  - Re-establish the scheduling invariant
- Steps
  - Move the suspended process back to the ready list
  - Change the state from *suspended* to *ready*
  - Call *resched*
- Note: resumption does *not* guarantee instantaneous execution of the resumed process

Moving A Process To The Ready List

- We will see that several system calls are needed to make a process ready
- To make it easy, Xinu includes an internal function named *ready* that makes a process ready
- *Ready* takes a process ID as an argument
- The steps are
  - Change the process's state to *PR_READY*
  - Insert the process onto the ready list
  - Ensure that the scheduling invariant is enforced

An Internal Function To Make A Process Ready

```
/* ready.c - ready */

#include <xinu.h>

qid16   readylist;                      /* Index of ready list          */

/*------------------------------------------------------------------------
 *  ready  -  Make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
status  ready(
          pid32         pid             /* ID of process to make ready  */
        )
{
        register struct procent *prptr;

        if (isbadpid(pid)) {
                return SYSERR;
        }

        /* Set process state to indicate ready and add to ready list */

        prptr = &proctab[pid];
        prptr->prstate = PR_READY;
        insert(pid, readylist, prptr->prprio);
        resched();

        return OK;
}
```

Enforcing The Scheduling Invariant

- When a process is moved to the ready list, the process becomes eligible to use the processor again
- Recall that when the set of eligible processes changes, the scheduling invariant specifies that we must check whether a new process should execute
- Consequence: after it moves a process to the ready list, *ready* must re-establish the scheduling invariant
- Surprisingly, *ready* does not check the scheduling invariant explicitly, but instead simply calls *resched*
- We can now appreciate the design of *resched*: if the newly ready process has a lower priority than the current process, *resched* returns without switching context and the current process remains running

The Resume System Call (Part 1)

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
syscall resume(
          pid32         pid             /* ID of process to unsuspend   */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process's table entry */
        pri16   prio;                   /* Priority to return           */

        mask = disable();
        if (isbadpid(pid)) {
                restore(mask);
                return SYSERR;
        }
```

The Resume System Call (Part 2)

```
        prptr = &proctab[pid];
        if (prptr->prstate != PR_SUSP) {
                restore(mask);
                return SYSERR;
        }
        prio = prptr->prprio;           /* Record priority to return    */
        ready(pid);
        restore(mask);
        return 0xffff & prio;
}
```

- Consider the code for *resume* and *ready*
- By calling *ready*, *resume* does not need code to insert a process on the ready list, and by calling *resched*, *ready* does not need code to re-establish the scheduling invariant
- The point: choosing OS functions carefully means software at successive levels will be small and elegant

Keeping Processes On A List

- We have seen that suspended processes are not placed on any list
- Why not?
  - Function *resume* requires the caller to supply an argument that specifies the ID of the process to be resumed
  - We will see that no other operating system functions operate on suspended processes or handle the entire set of suspended processes
- Consequence: there is no reason to keep a list of suspended processes
- In general: an operating system only places a process on a list if a function needs to handle an entire set of processes that are in a given state (e.g., the scheduler needs to find the highest priority ready process)

Summary Of Process Suspension And Resumption

- An OS offers functions that can change a process's state
- Xinu allows a process to be
  - Suspended temporarily
  - Resumed later
- A state variable associated with each process records the process's current status
- When resuming a process, the scheduling invariant must be re-established

Something To Think About

- Resume returns the priority of the resumed process
- The code
  - Extracts the priority from the process table entry
  - Makes the process ready
  - Returns the extracted priority to its caller
- Is the value returned guaranteed to be the priority of the process?
- Remember that in a concurrent environment, other processes can run at any time, and an arbitrary amount of time can pass between any two instructions

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 arise
  - 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

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's table entry */
        int32   i;                      /* Index into descriptors       */

        mask = disable();
        if (isbadpid(pid) || (pid == NULLPROC)
            || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
                restore(mask);
                return SYSERR;
        }
        send(prptr->prparent, pid);
        for (i=0; i<3; i++) {
                close(prptr->prdesc[i]);
        }
```

Xinu Implementation Of Kill (Part 2)

```
        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           */
}
```

Process Creation

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();

/*------------------------------------------------------------------------
 *  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;
}
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 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;
}
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's 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;
}
```

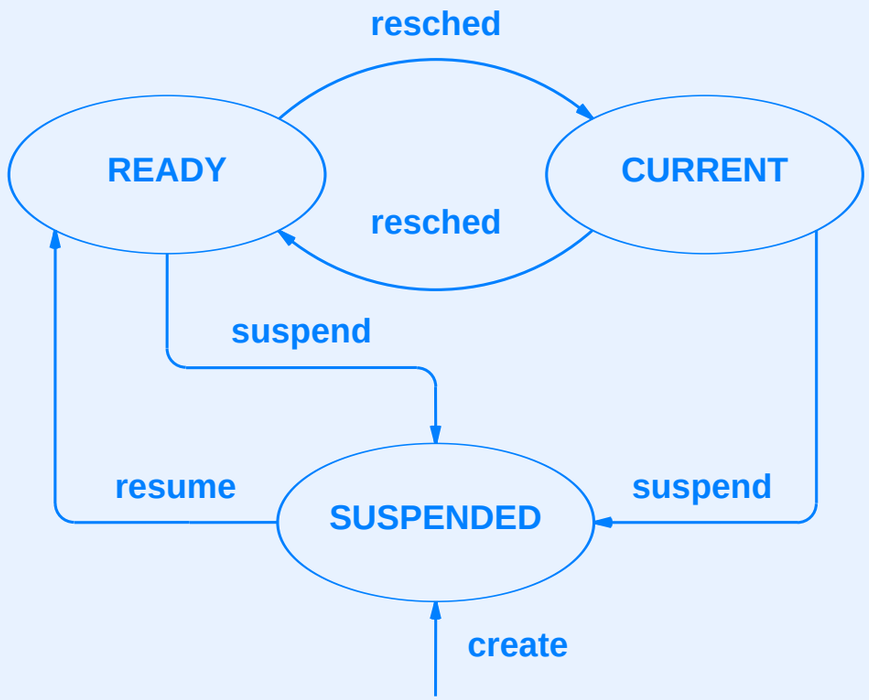
Process Creation On X86 (Part 5)

```
/*------------------------------------------------------------------------
 *  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 pieces of 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

Create Added To The State Transition Diagram



Questions?