ADVANCED OPERATING SYSTEMS

UNIT PROCESS ENVIRONMENT, PROCESS CONTROL AND PROCESS RELATIONSHIPS

BY

MR.PRASAD SAWANT

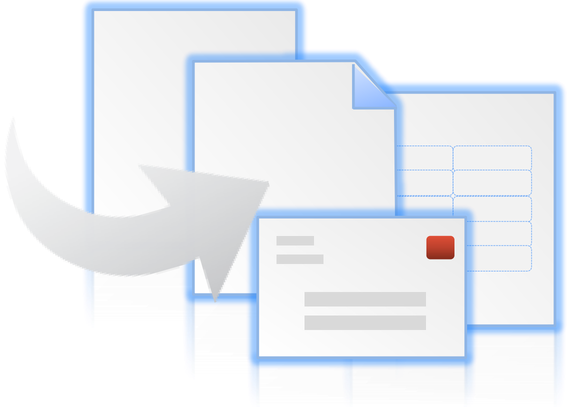
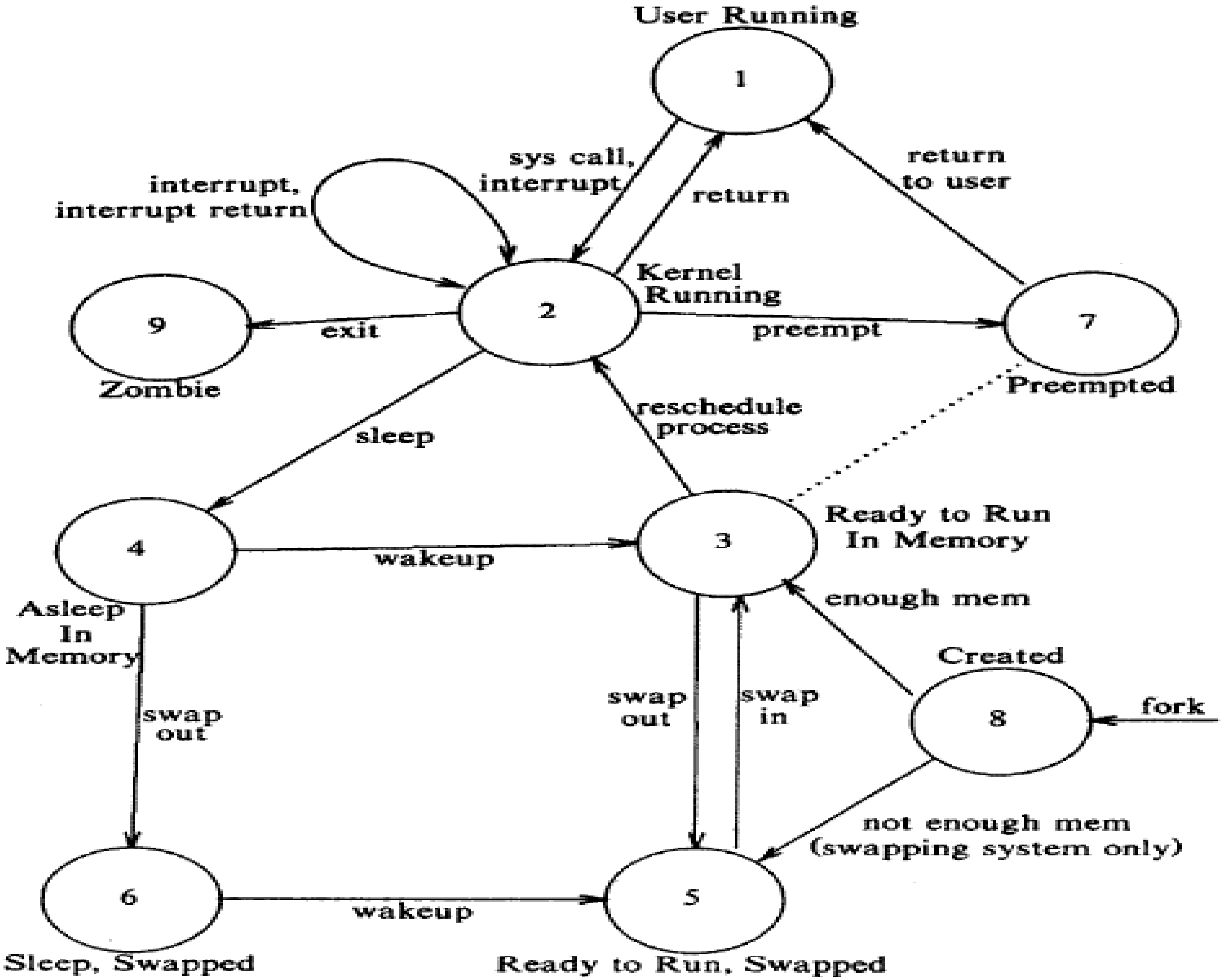


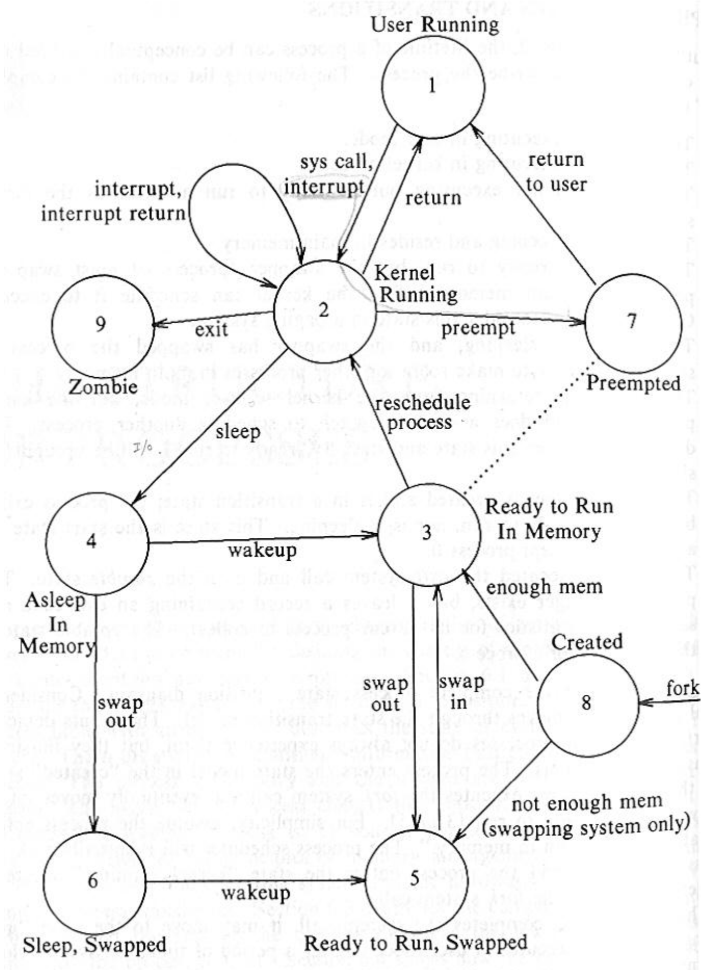
TABLE OF CONTENTS


User Running
1
sys call, interrupt
return
return to user
interrupt, interrupt return
Kernel Running
2
9
exit
Zombie
preempt
7
Preempted
reschedule process
sleep
4
wakeup
3
Ready to Run In Memory
enough mem
Asleep In Memory
swap out
swap out
swap in
Created
8
fork
not enough mem (swapping system only)
6
wakeup
5
Sleep, Swapped
Ready to Run, Swapped

PROCESS STATES & TRANSITIONS

Process State Transition

- No process can preempt another process executing in the kernel.
- The process has no control over those state transitions.
- A process can make system calls to move from state "user running" to state "kernel running".
- A process can *exit* of its own volition, but external events may dictate that it *exit*s without explicitly invoking the *exit* system call.



PROCESS TABLE ENTRY & U AREA

PTE & *U Area*

- Kernel data structures that describe the state of a process

Process Table Entry

- Always be accessible to the kernel
- Fields
    - Process state
    - Pointers to process and its *u area* – context switch, swapping
    - Process size
    - User ID
    - Process ID
    - Event descriptor table
    - Scheduling parameters
    - Signal enumerated fields
    - Timers

PROCESS TABLE ENTRY & U AREA

U Area

- Need to be accessible only to the running process.
- The kernel alocates space for the *u area* only when creating a process.
- Fields
    - Pointer to process table entry
    - Real and effective user IDs
    - Timers – time the process spent executing
    - An array for the process to react to signals
    - Control terminal – if one exists
    - Error field, return value – system call
    - I/O parameters
    - Current directory, current root
    - User file descriptor table
    - Limits – process, file
    - Permission – used on creating the process

LAYOUT OF SYSTEM MEMORY

Problem of Physical Address Space

- A Process on the UNIX system consists of three logical sections : text, data, and stack
- If the machine were to treat the generated addresses as address locations in physical memory, it would be impossible for two processes to execute concurrently if their set of generated addresses overlapped.

Virtual Address Space

- The machine’s memory management unit translates the virtual addresses generated by the compiler into address locations in physical memory.
- The compiler does not have to know where in memory the kernel later load the program for execution.

REGIONS

Regions

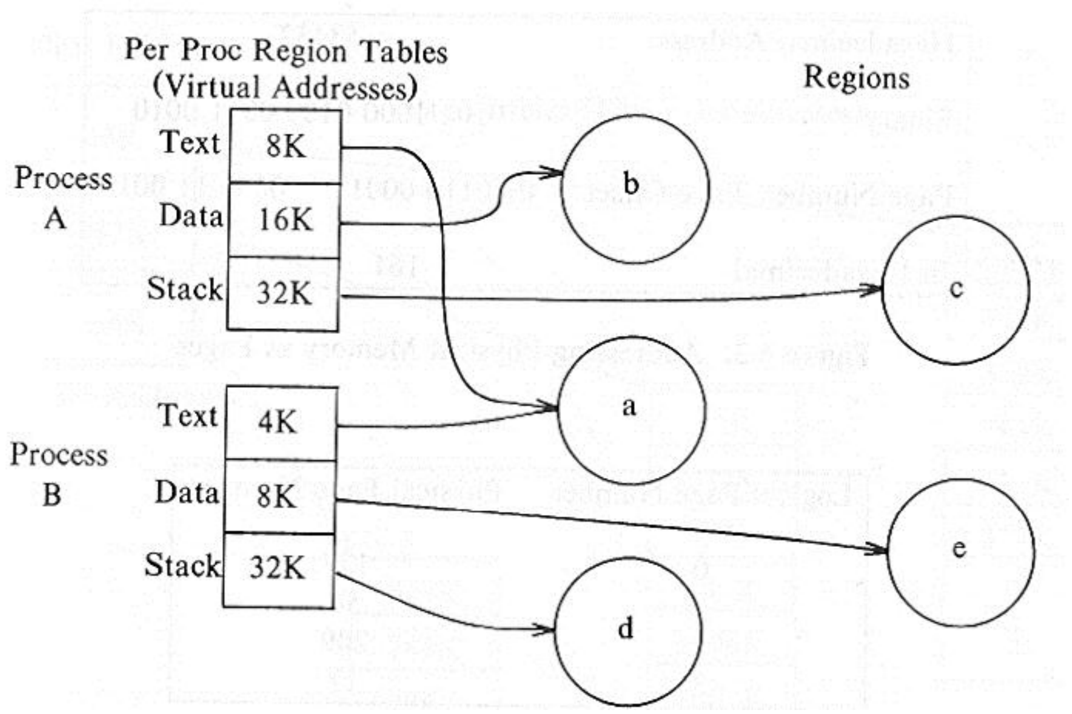
- A contiguous area of the virtual address space of a process
- It can be treated as a distinct object to be shared/protected.
- Thus text, data, and stack usually form separate regions of a process. – Several processes can share a region.

Region Table

- It contains the information to determine where its contents are located in physical memory.
- Each pregion(Per Process Region Table) entry contains:
    - Pointer to a region table entry
    - Starting virtual address in the process
    - Permission field : read-only, read-write or read-execute
- Several processes can share parts of their address space via a region.

REGIONS

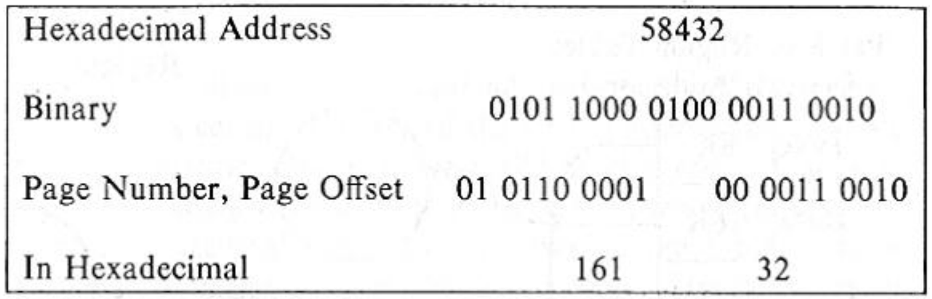
Processes and Regions



PAGES & PAGE TABLES

Page

- In a memory management architecture based on pages, the memory management hardware divides physical memory into a set of equal-sized blocks called pages. (512~4K bytes)
- Memory location can be addressed by a(page number, byte offset in page)
- Example:
  - 2^{32} bytes of physical memory
  - Page size : 1K bytes (2^{22} pages of physical memory)



Hexadecimal Address	58432	
Binary	0101 1000 0100 0011 0010	
Page Number, Page Offset	01 0110 0001	00 0011 0010
In Hexadecimal	161	32

PAGES & PAGE TABLES

Page Table

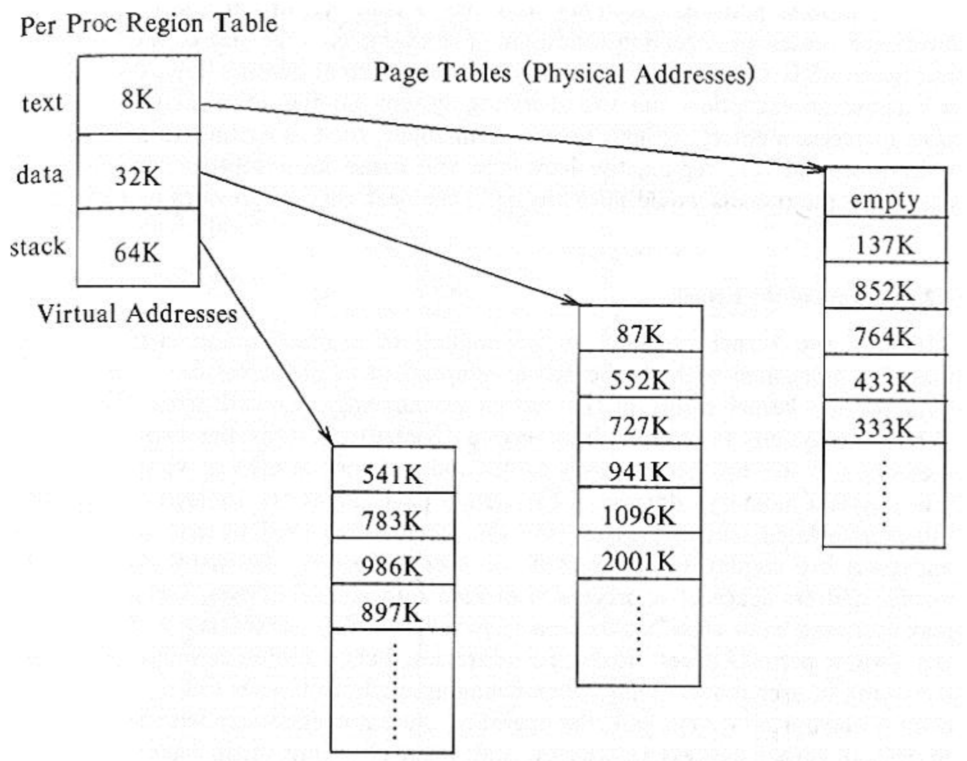
- When the kernel assigns physical pages of memory to a region, it need not assign the pages contiguously or in a particular order.
- The region table entry contains a pointer to a table of physical page numbers called a page table.
- The logical page number is the index into an array of physical page numbers.

Address Translation

- Modern machines use a variety of hardware registers and caches to speed up the address translation procedure, because the memory references and address calculations would otherwise be too slow.
- Such operations are machine dependent and vary from one implementation to another.

PAGES & PAGE TABLES

Mapping Virtual Addresses to Physical Addresses



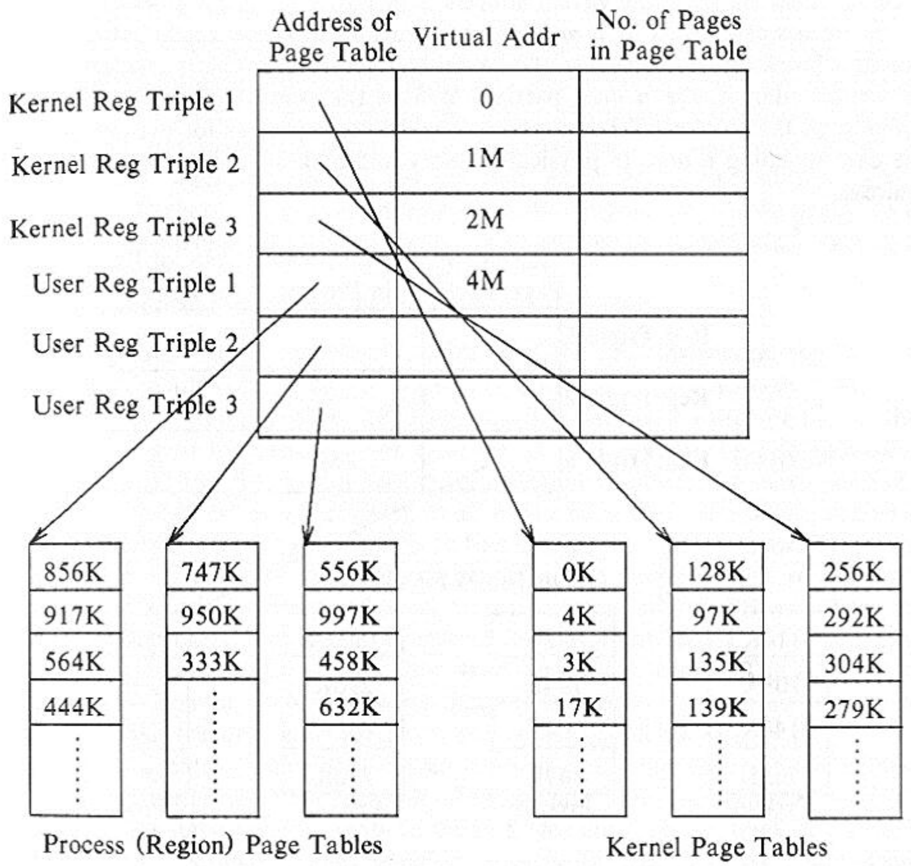
LAYOUT OF THE KERNEL

Layout of the Kernel

- Although the kernel executes in the context of a process, the virtual memory mapping associated with the kernel is independent of all processes.
- The code and data for the kernel reside in the system permanently, and all processes share it.
- The kernel page tables are analogous to the page tables associated with a process.
- In many machines, the virtual address space of a process is divided into several classes, including system and user, and each class has its own page tables.
- When executing in kernel mode, the system permits access to kernel addresses, but it prohibits such access when executing in user mode.

LAYOUT OF THE KERNEL

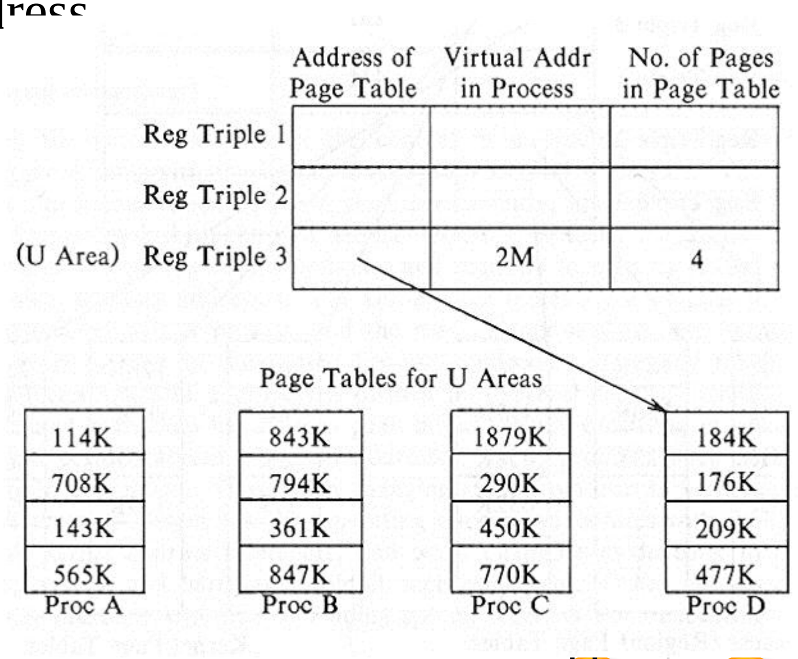
Changing Mode from User to Kernel



THE U AREA

Memory Map of U Area in the Kernel

- The kernel access *u area* as if there were only one *u area* in the system, that of the running process.
- When compiling the operating system, the loader assigns the variable *u area* a fixed virtual addr
- Example:
    - *u area* is 4K bytes long at Virtual Address 2M
    - 1st – kernel text
    - 2nd – kernel data
    - 3rd – *u area* for process D



THE CONTEXT OF A PROCESS

The Context of a Process

- It consists of its (user) address space, hardware registers and kernel data structures that relate to the process.
- Formally, the union of its user-level context, register context, and system-level context.

User-level Context

- Process text, data, user stack, and shared memory
- Parts of the virtual address space of a process periodically do not reside in main memory for swapping or paging.

Register Context

- Program counter, process status register, stack pointer, general-purpose registers

THE CONTEXT OF A PROCESS

System-level Context

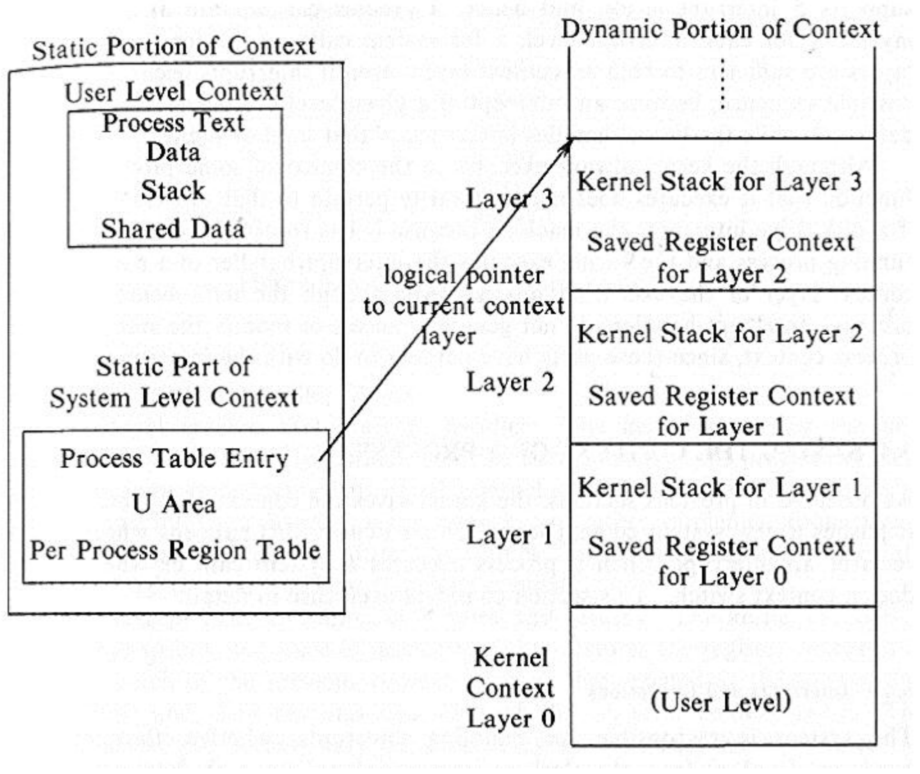
- Static part : process table entry, *u area*, region / page tables
- Dynamic part : kernel stack, system-level context layer (including register context)
- The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does a context switch.

Context Layer

- A process runs within its context or, more precisely, within its current context layer.
- The number of context layers is bounded by the number of interrupt levels the machine supports.
- Ex) 5 for interrupt level + 1 for system call + 1 for user-level => 7 context layers are sufficient to hold all context layers.

THE CONTEXT OF A PROCESS

Components of the Context of a Process



Static Portion of Context
User Level Context
Process Text
Data
Stack
Shared Data
Static Part of System Level Context
Process Table Entry
U Area
Per Process Region Table
logical pointer to current context layer
Dynamic Portion of Context
Layer 3
Kernel Stack for Layer 3
Saved Register Context for Layer 2
Layer 2
Kernel Stack for Layer 2
Saved Register Context for Layer 1
Layer 1
Kernel Stack for Layer 1
Saved Register Context for Layer 0
Kernel Context Layer 0
(User Level)

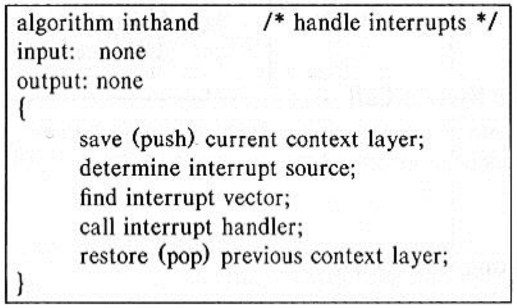
INTERRUPTS & EXCEPTIONS

Interrupt / Exception Handling

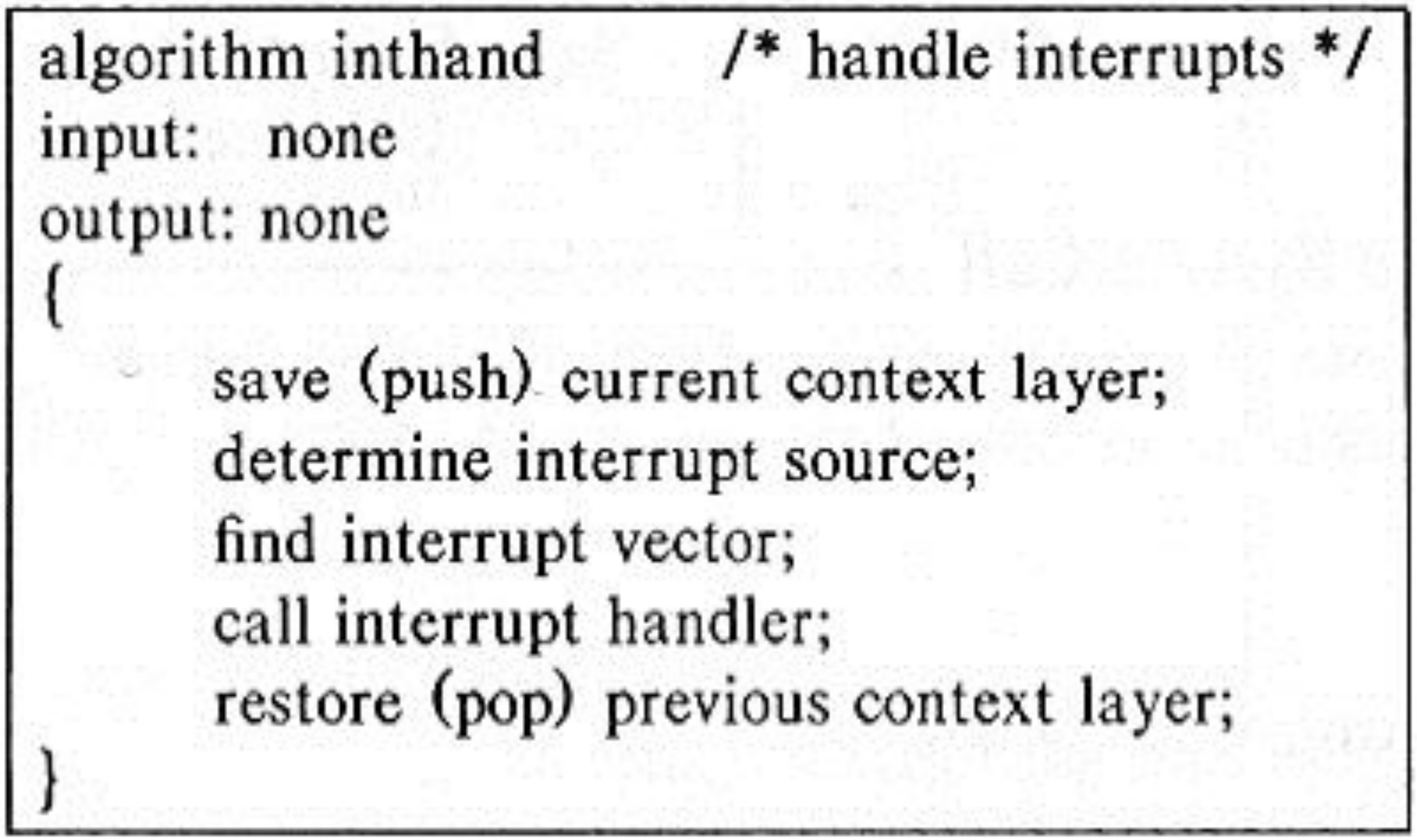
- The system is responsible for handling interrupts:
    - Results from hardware (such as the clock or peripheral devices)
    - Programmed interrupts (software interrupts)
    - Exceptions (such as page faults)

How the Kernel Handles Interrupts

- Some machines do part of the sequence of operations in hardware or micro-code to get better performance than if all operation were done by software.
- Algorithm for handling interrupts



```
algorithm inthand        /* handle interrupts */
input:   none
output: none
{
        save (push) current context layer;
        determine interrupt source;
        find interrupt vector;
        call interrupt handler;
        restore (pop) previous context layer;
}
```

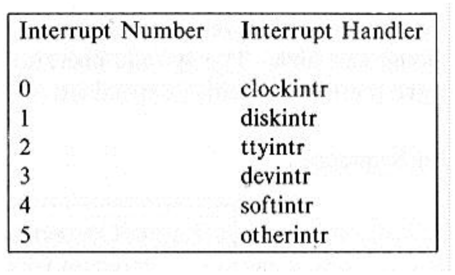


```
algorithm inthand          /* handle interrupts */
input:   none
output: none
{
        save (push) current context layer;
        determine interrupt source;
        find interrupt vector;
        call interrupt handler;
        restore (pop) previous context layer;
}
```

INTERRUPTS & EXCEPTIONS

Interrupt Vector

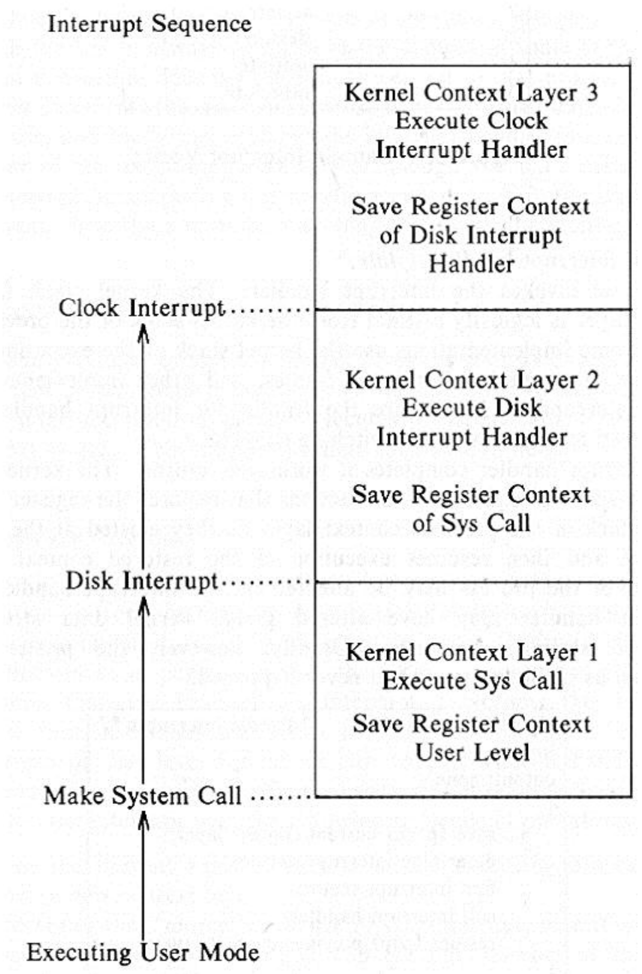
- It contains the address of the interrupt handler for the corresponding interrupt source and a way of finding a parameter for the interrupt handler.



Interrupt Number	Interrupt Handler
0	clockintr
1	diskintr
2	ttyintr
3	devintr
4	softintr
5	otherintr

Example of Interrupts

system call -> disk intr. ->clock intr.

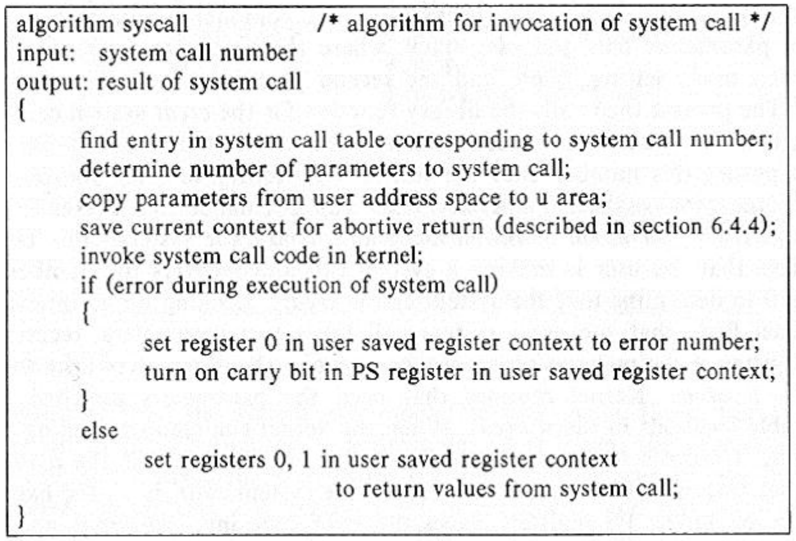

Interrupt Sequence
Kernel Context Layer 3
Execute Clock Interrupt Handler
Save Register Context of Disk Interrupt Handler
Clock Interrupt
Kernel Context Layer 2
Execute Disk Interrupt Handler
Save Register Context of Sys Call
Disk Interrupt
Kernel Context Layer 1
Execute Sys Call
Save Register Context User Level
Make System Call
Executing User Mode

SYSTEM CALL INTERFACE

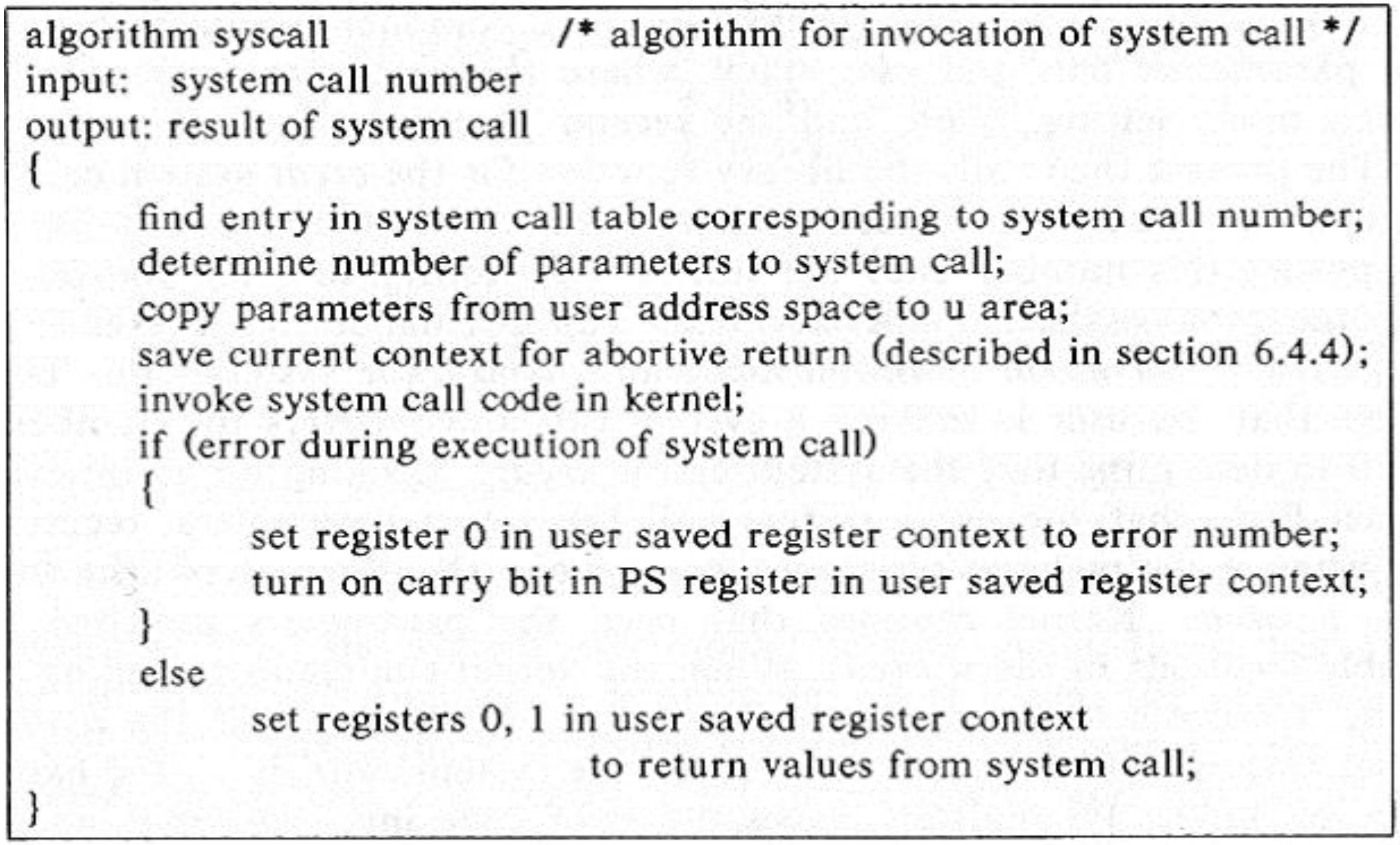
System Call

- The library functions typically invoke an instruction that changes the process execution mode to kernel mode and cause the kernel to start executing code for system calls.
- The system call interface is a special case of an interrupt handler. (operating system trap)

- When the kernel returns from the operating system trap to user mode, it returns to the library instruction after the trap. The library interprets the return values from the kernel and returns a value to the user program.



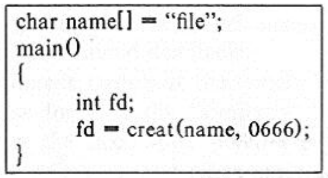
```
algorithm syscall            /* algorithm for invocation of system call */
input:  system call number
output: result of system call
{
    find entry in system call table corresponding to system call number;
    determine number of parameters to system call;
    copy parameters from user address space to u area;
    save current context for abortive return (described in section 6.4.4);
    invoke system call code in kernel;
    if (error during execution of system call)
    {
        set register 0 in user saved register context to error number;
        turn on carry bit in PS register in user saved register context;
    }
    else
        set registers 0, 1 in user saved register context
              to return values from system call;
}
```



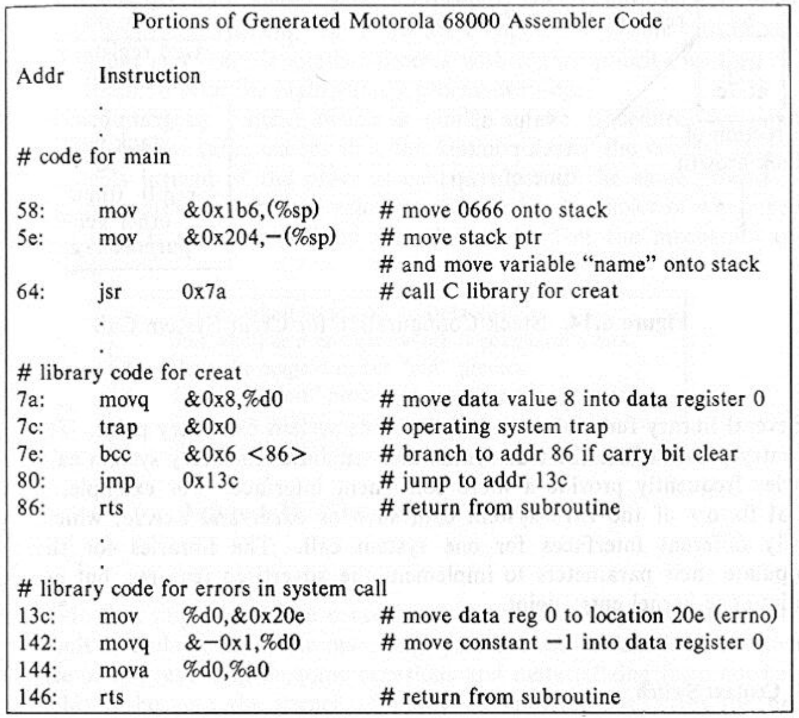
```
algorithm syscall              /* algorithm for invocation of system call */
input:   system call number
output: result of system call
{
        find entry in system call table corresponding to system call number;
        determine number of parameters to system call;
        copy parameters from user address space to u area;
        save current context for abortive return (described in section 6.4.4);
        invoke system call code in kernel;
        if (error during execution of system call)
        {
                set register 0 in user saved register context to error number;
                turn on carry bit in PS register in user saved register context;
        }
        else
                set registers 0, 1 in user saved register context
                                        to return values from system call;
}
```

SYSTEM CALL INTERFACE

Example: Create System Call

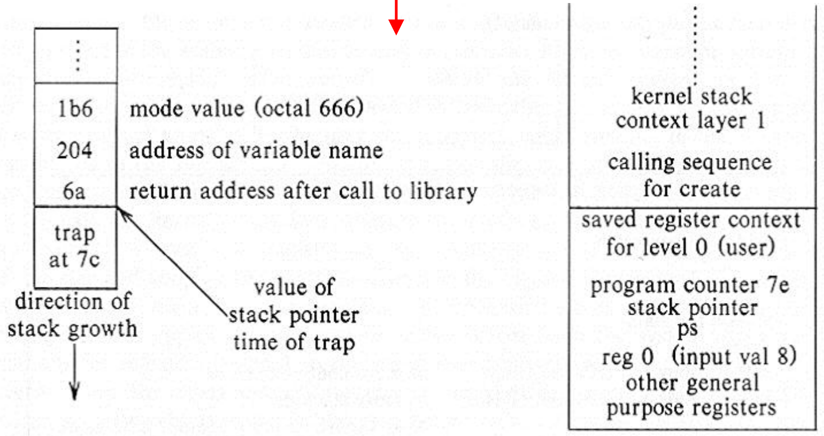


```
char name[] = "file";
main()
{
        int fd;
        fd = creat(name, 0666);
}
```



```
              Portions of Generated Motorola 68000 Assembler Code
Addr    Instruction
        .
        .
# code for main
        .
58:     mov     &0x1b6,(%sp)      # move 0666 onto stack
5e:     mov     &0x204,-(%sp)     # move stack ptr
                                  # and move variable "name" onto stack
64:     jsr     0x7a              # call C library for creat
        .
        .
# library code for creat
7a:     movq    &0x8,%d0          # move data value 8 into data register 0
7c:     trap    &0x0              # operating system trap
7e:     bcc     &0x6 <86>         # branch to addr 86 if carry bit clear
80:     jmp     0x13c             # jump to addr 13c
86:     rts                       # return from subroutine
        .
        .
# library code for errors in system call
13c:    mov     %d0,&0x20e        # move data reg 0 to location 20e (errno)
142:    movq    &-0x1,%d0         # move constant -1 into data register 0
144:    mova    %d0,%a0
146:    rts                       # return from subroutine
```

Stack Configuration



1b6
mode value (octal 666)
204
address of variable name
6a
return address after call to library
trap at 7c
direction of stack growth
value of stack pointer time of trap
kernel stack context layer 1
calling sequence for create
saved register context for level 0 (user)
program counter 7e
stack pointer
ps
reg 0 (input val 8)
other general purpose registers

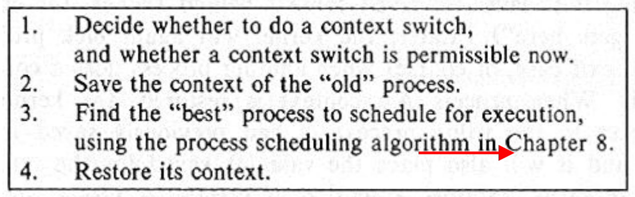
Generated Code for Motorola 68000

CONTEXT SWITCH

Context Switch Mechanism

- The kernel permits it under four circumstances:
    - When a process puts itself to sleep
    - When it exits
    - When it returns from a system call to user mode
    - When it returns to user mode after interrupt handling
- The kernel ensures integrity and consistency of internal data structures by prohibiting arbitrary context switches.
- The procedure for a context switch is similar to the procedures for handling interrupts and system calls, except that the kernel restores the context layer of a different process instead of the previous context layer of the same process.

Steps for a Context Switch

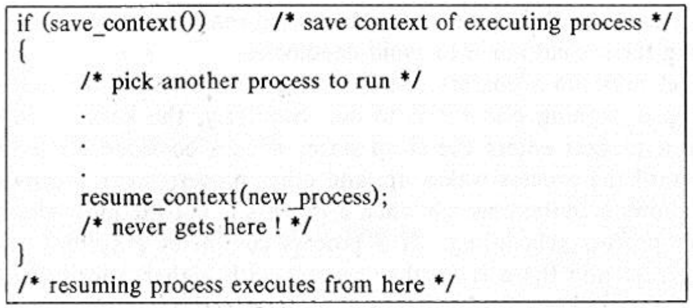


1. Decide whether to do a context switch, and whether a context switch is permissible now.
2. Save the context of the "old" process.
3. Find the "best" process to schedule for execution, using the process scheduling algorithm in Chapter 8.
4. Restore its context.

CONTEXT SWITCH

Doing a Context Switch

- The context switch code is usually the most difficult to understand in the operating system, because function calls give the appearance of not returning on some occasions and materializing from nowhere on others.
- Scenario for context switch
    - The function *save_context* saves information about the context of the running process and returns the value 1.
    - Among other pieces of information, the kernel saves the value of the current program counter (in the function *save_context*) and the value 0, to be used later as the return value.



```
if (save_context())         /* save context of executing process */
{
        /* pick another process to run */
        .
        .
        .
        resume_context(new_process);
        /* never gets here ! */
}
/* resuming process executes from here */
```

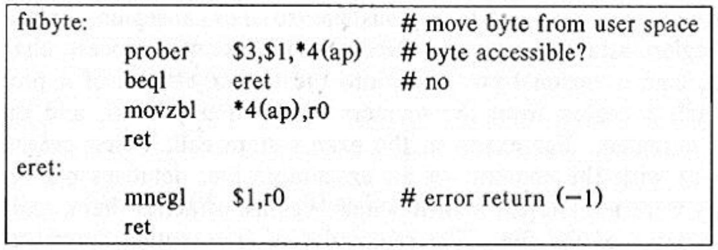
CONTEXT SWITCH

Saving Context for Abortive Returns

- The algorithm to save a context is *setjmp* and one to restore the context is *longjmp*.
- It stores/resumes the context in/from the *u area*.

Copying Data between System & User Address Space

- The kernel must ascertain that the address being read or written is accessible as if it has been executing in user mode.
- Therefore, copying data between kernel space and user space is an expensive proposition, requiring more than one instruction.
- Sample) Moving data from user to system space on a VAX



```
fubyte:                                 # move byte from user space
        prober  $3,$1,*4(ap)            # byte accessible?
        beql    eret                    # no
        movzbl  *4(ap),r0
        ret
eret:
        mnegl   $1,r0                   # error return (-1)
        ret
```

SESSION OUT LINE

1. Process Creation -- fork
2. Signals
3. Process Termination -- exit
4. Awaiting Process Termination -- wait
5. Invoking Other Programs -- exec
6. The User ID of A Process
7. Changing The Size of A Process -- brk
8. The SHELL

System Boot and The Init Process

PROCESS SYSTEM CALLS

System Calls Dealing with Memory Management				System Calls Dealing with Synchronization			Miscellaneous	
Fork	Exec	Brk	exit	wait	signal	kill	setpgrp	setid
dupreg Attachreg	Detachreg Allocreg Growreg Loadreg Mapreg	growreg	detachreg					

PROCESS CREATION

Fork

The syntax for the fork system call

- Pid = fork();
    - In the parent process, pid is the child process ID
    - In the child process, pid is 0

Sequence of operations for fork.

- 1. It allocates a slot in the process table for the new process
- 2. It assigns a unique ID number to the child process
- 3. It makes a logical copy of the context of the parent process.
    - Sometimes increment a region reference count
- 4. It increments file and inode table counters for files associated with the process
- 5. It returns the ID number of the child to the parent process, and a 0 value to the child process.

Algorithm fork

```
Input : none
Output : to parent process, child PID number
                  to child process, 0
{
              check for available kernel resources;
              get free proc table slot, unique PID number;
              check that user not running too many processes;

   mark child state "being created;"
              copy data from parent proc table slot to new child slot;

   increment counts on current directory inode and changed root (if applicable)
              increment open file counts in file table;

   make copy of parent context (u area, text, data, stack) in memory;
              push dummy system level context layer onto child system level context;
                            dummy context contains data allowing child process to recognize itself,
       and start running from here when scheduled;

   if( executing process is parent process )
              {
                            change child state to "ready to run;"
                            return( child ID );              /* from system to user */
              }
              else
              {
                            initialize u area timing fields;
                            return( 0 );                                  /* to user */
              }
}
```

1. On swapping system, it need space either in memory or on disk to hold the child process; on paging system, it has to allocate memory for auxiliary tables such as page tables.

2. ID number for the new process, one greater than the most recently assigned ID number.

3. Limit on the number of processes

the last remaining slot

Superuser take drastic action and spawn a process that forces other processes to exit

Process Creation

1. the child “inherit” the parent process real and effective user ID, parent process group, parent nice value

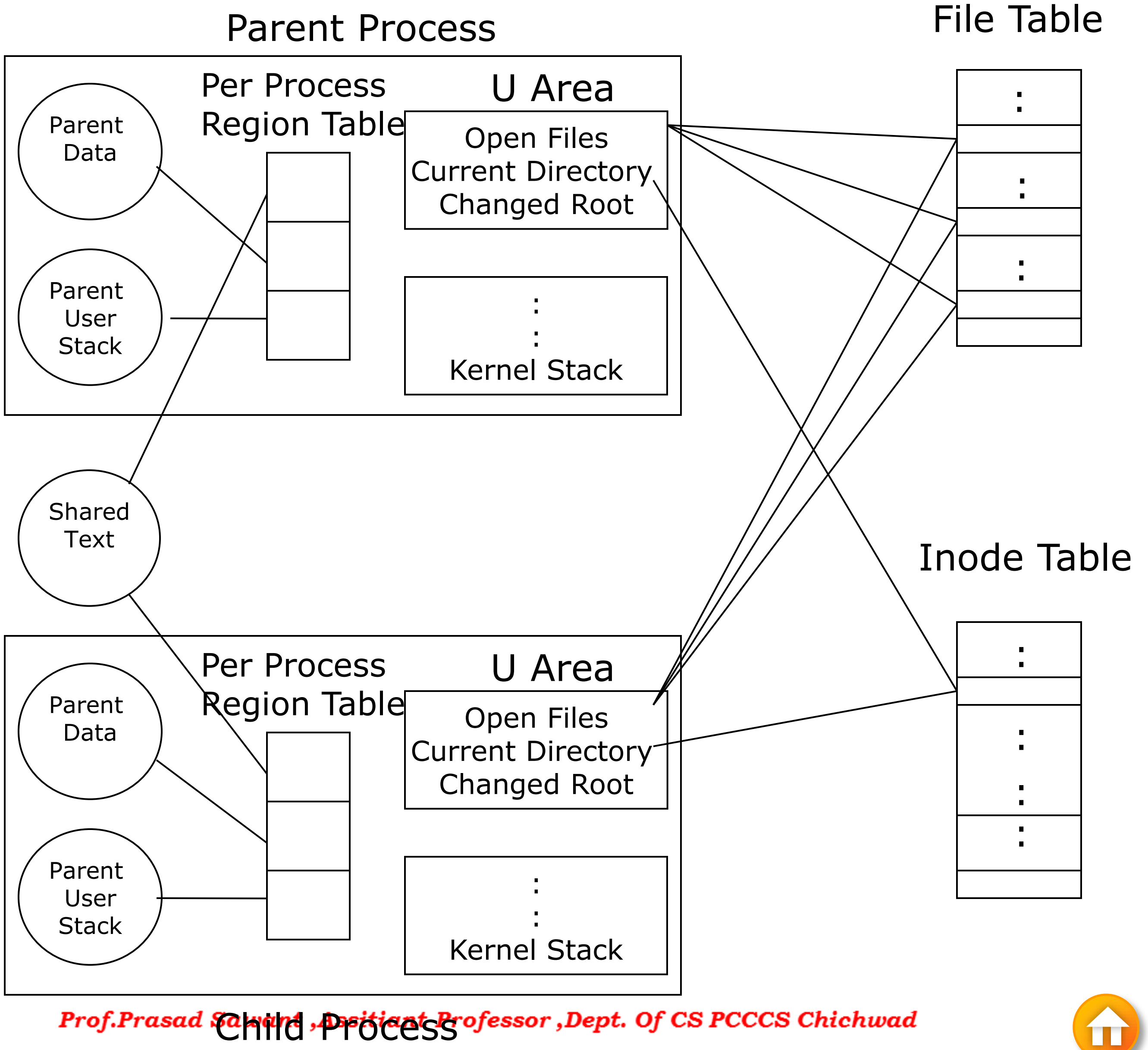
the kernel assign the parent process ID field in the child slot

putting the child in the process tree structure

initializes various scheduling parameters (initial priority value, initial CPU usage, and other timing fields)

the initial state of the process is “being created”

1. Process Creation



1. Process Creation

1. The child process inherits the current directory of the parent process. The number of processes that currently access the directory increases by 1, kernel increments its inode reference count.

2. If the parent process or one of its ancestors had ever executed the chroot system call, the child process inherits the changed root and increments its inode reference count

3. The effect of fork is similar to that of dup vis-à-vis open files

- difference

1. Process Creation

Static portion

- It allocates memory for the child process u area, regions, auxiliary page tables, duplicates every region in the parent process using dupreg, and attaches every region to the child process using attachreg
- Difference -The u area contains a pointer to its process table slot.

Dynamic portion

- The kernel copies the parent context layer 1, containing the user saved register context and the kernel stack frame of the fork system call
- The kernel then creates a dummy context layer 2 for the child process, containing the saved register context for context layer 1. It sets the program counter and other registers in the saved register context so that it can "restore" the child context, even though it had never executed before, and so that the child process can recognize itself as the child when it run

1. Process Creation

Figure 7.4 – example of sharing file access across a fork system call

```
#include<fcntl.h>
Int fdrd, fdwt;
Char c;

Main(argc, argv)
    int argc;
    char *argv[];
{
    if ( argc != 3 )
            exit(1);
    if(( fdrd = open( argv[1], O_RDONLY )) == -1)
            exit(1);
    if(( fdwt = creat( argv[2], 0666 )) == -1)
            exit(1);
    fork();
    /* both procs execute same code */
    rdwrt();
    exit(0);
}
rdwrt()
{
    for(;;)
    {
        if( read( fdrd, &c, 1 ) != 1 )
                return;
        write(fdwt, &c, 1 );
    }
}
```

1. Process Creation

```
#include<string.h>
Char string[] = " hello world";
Main()
{
    int count, I;
    int to_par[2], to_chil[2];
    char buf[256];
    pipe(to_par);
    pipe(to_chil);
    if ( fork() == 0 )
    {
         /* child process executes here */
        close(0); /*close old standard input */
        dup(to_chil[0]); /*dup pipe read to standard input*/
        close(1); /*close old standard output */
        dup(to_par[1]); /*dup pipe write to standard input*/
        close(to_par[1]);/*close unnecessary pipe descriptors*/
        close(to_chil[0]);
        close(to_par[0]);
        close(to_chil[1]);
        for(;;)
        {
                if((count = read( 0, buf, sizeof(buf) ) == 0 )
                    exit();
                write(1, buf,count );
        }
    }
```

```
/* parent process executes here */
                close(1);  /* rearrange standard in, out */
dup(to_chil[1]);
close(0);
                dup(to_par[0]);
close(to_chil[1]);
close(to_par[0]);
close(to_chil[0]);
close(to_par[1]);
for( I = 0; I < 15; I++ )
{
       write(1, string, strlen(string));
                read(0, buf, sizeof(buf));
                }
}
```

SIGNALS

Inform processes of the occurrence of asynchronous events

Kill or the kernel may send signals internally

Classified

- Signals having to do with the termination of a process, sent when a process exits or when a process invokes the signal system call with the death of child parameter;
- Signals having to do with process induced exceptions such as when a process accesses an address outside its virtual address space, when it attempts to write memory that is read-only (such as program text), or when it executes a privileged instruction or for various hardware errors;
- Signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during exec after the original address space has been released;

2. Signal

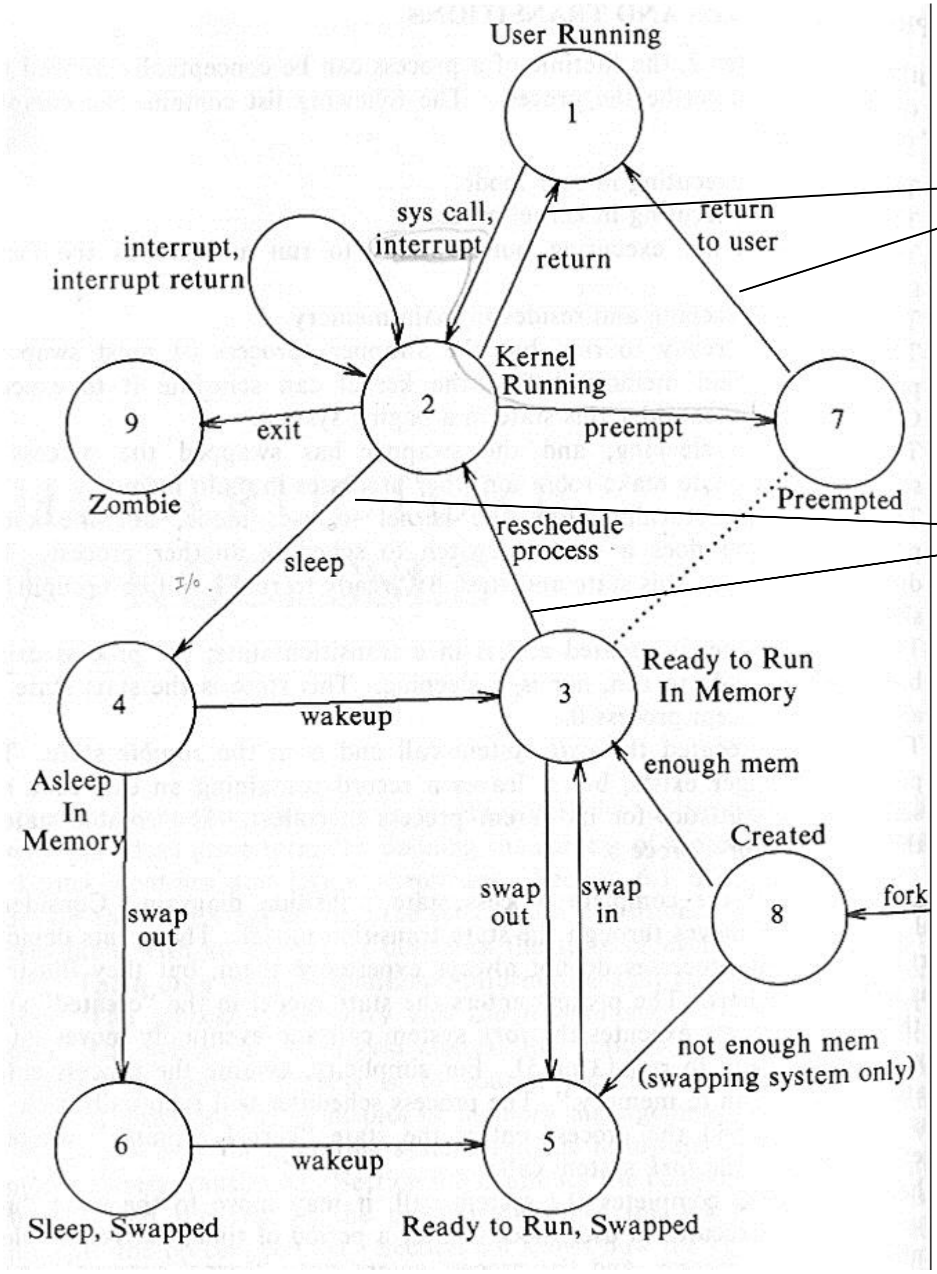
- Signals caused by an unexpected error condition during a system call, such as making a nonexistent system call, writing a pipe that has no reader processes, or using an illegal "reference" value for the lseek system call.
- Signals originating from a process in user mode, such as when a process wishes to receive a alarm signal after a period of time, or when processes send arbitrary signals to each other with the kill system call;
- Signals related to terminal interaction such as when a user hangs up a terminal, or when a user presses the "break" or "delete" keys on a terminal keyboard;
- Signals for tracing execution of a process

2. Signal

* How the kernel sends a signal to a process
* How the process handles a signal
* How a process controls its reaction signals

- To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received.
- If the process is asleep at an interruptible priority, the kernel awakens it.
- The kernel checks for receipt of a signal when a process is about to return from kernel mode to user mode and when it enters or leaves the sleep state at suitably low scheduling priority.
- The kernel handles signals only when a process returns from kernel mode to user mode.

2. Signal



Check And Handle Signals

Check For signals

The algorithm show the kernel executes to determine if a process received a signal

```
Algorithm issig

Input: none
Output: true, if process received signals that it does not ignore
        false otherwise
{
    while( received signal field in process table entry not 0 )
    {
        find a signal number sent to the process;
        if(signal is death of child)
        {
            if(ignoring death of child signals)
                free process table entries of zombie children;
            else if( not ignoring signal )
                return(true);
        }
        else if( not ignoring signal )
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);
}
```

Handling Signals

- The kernel handles signals in the context of the process that receives them so a process must run to handle signals.
    - Exits on receipt of the signal
    - Ignores the signal
    - Executes a particular(user) function on receipt of the signal

- The syntax for the signal system call
    - Oldfunction = signal(signum, function);
        - The process will ignore future occurrence of the signal if the parameter value is 1
        - Exit in the kernel on receipt of the signal if its value is 0

- The u area contains a array of signal-handler fields, one for each signal defined in the system

Algorithm for Handing Signals

```
Algorithm psig      /* handle signals after recognizing their existence */

Input: none
Output : none
{
    get signal number set in process table entry;
    clear signal number in process table entry;
    if( user specified function to handle the signal )
    {
            get user virtual address of signal catcher stored in u area;
            /* the next statement has undesirable side effects */
            clear u area entry that stored address of signal catcher;
            modify user level context:
                    artificially create user stack frame to mimic call to signal catcher function;
            modify system level context:
                    write address of signal catcher into program counter field of
                    user saved register context;
            return;
    }
    if( signal is type that system should dump core image of process )
    {
            create file named "core" in current directory;
            write contents of user level context to file "core";
    }
    invoke exit algorithm immediately
}
```

2. Signal

If a process receives a signal that is had previously decided to catch, it executes the user specified signal handling function immediately when it returns to user mode, after the kernel does the following step

- 1. The kernel accesses the user saved register context, finding the program counter and stack pointer that it had saved for return to the user process.
- 2. It clears the signal handler field in the u area, setting it to the default state.
- 3. The kernel creates a new stack frame, writing In the values of the program counter and stack pointer it had retrieved from the user saved register context and allocating new space, if necessary. The user stack looks as if the process had called a user-level function(the signal catcher) at the point where it had made the system call or where the kernel had interrupted it (before recognition of the signal)
- 4. The kernel changes the user saved register context: It resets the value for the program counter to the address of the signal catcher function and sets the value for the stack pointer to account for the growth of the user stack.

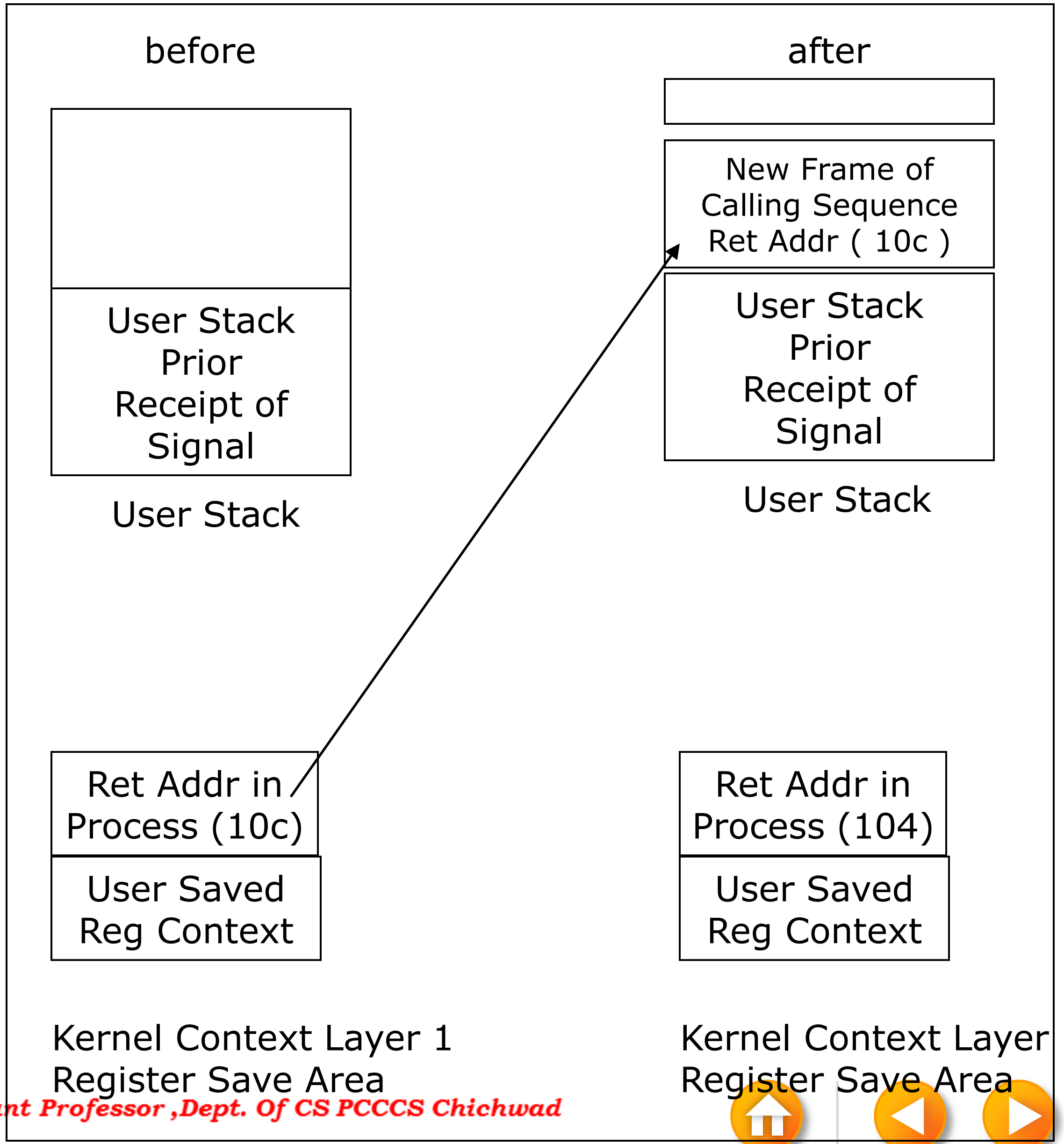
2. Signal

```
#include<signal.h>
Main()
{
     extern catcher();
     signal( SIGINT, catcher );
     kill( 0, SIGINT );
}

Catcher()
{
}
```

```
    *** VAX DISASSEMBLER ***
                 :
                 :
_catcher()
          104:
          106:  ret
          107:  halt
_kill()
          108:
  # next line traps into kernel
          10a: chmk    $0x25
           10c: bgequ
0x6<0x114>
          10e: jmp       0x14(pc)
          114: clrl     r0
          116: ret
```

User Stack and kernel Save Area Before and After Receipt of Signa



before
after
User Stack Prior Receipt of Signal
User Stack
New Frame of Calling Sequence Ret Addr (10c)
User Stack Prior Receipt of Signal
User Stack
Ret Addr in Process (10c)
User Saved Reg Context
Ret Addr in Process (104)
User Saved Reg Context
Kernel Context Layer 1 Register Save Area
Kernel Context Layer 1 Register Save Area

2. Signal

Anomalies

- When a process handles a signal but before it returns to user mode, the kernel clears the field in the u area
- If the process wants to handle the signal, it must call the signal system call again
- A race condition
    - The child process suspends execution for 5 seconds to give the parent process time to execute the nice system call and lower its priority.
    - It is possible for the following sequence of events to occur,
        - 1. The child process sends an interrupt signal to parent process.
        - 2. The parent process catches the signal and calls the signal catcher, but the kernel preempts the process and switches context before it executes the signal system call again.
        - 3. The child process executes again and sends another interrupt signal to the parent process.
        - 4. The parent process receives the second interrupt signal, but it has not made arrangement to catch the signal. When it resumes execution, it exits.

2. Signal

```
#include<signal.h>
Sigcatcher()
{
    printf("PID %d caught one\n", getpid() );
    signal(SIGINT, sigcatcher );
}

Main()
{
    int ppid;
    signal(SIGINT, sigcatcher );
    if( fork() == 0 )
    {
            /* give enough time for both procs to set up */
            sleep(5);
            ppid = getppid();
            for(;;)
                if( kill(ppid, SIGINT ) == -1)
                        exit();
    }
    /* lower priority, greater chance of exhibiting race */
    nice(10);
    for(;;)
        ;
}
```

- The problem would be solved if the signal field were not cleared on receipt of the signal
  - Problem : if signals keep arriving and are caught, the user stack could grow out of bounds because of the nested calls to the signal catcher.
- Ignore
  - Problem : loss of information
- The BSD system allows a process to block and unblock receipt of signal

Process Groups

- The kernel uses the process group ID to identify groups of related processes that should receive a common signal for certain events.
- It saves the group ID in the process table
- grp= setpgrp();

Sending Signals from Processes

- Kill(pid, signum)
    - If pid is a positive integer, the kernel sends the signal to the process with process ID pid.
    - If pid is 0, the kernel sends the signal to all processes in the sender's process group
    - If pid is –1, the kernel sends the signal to all processes whose real user ID equals the effective user ID of the sender. If the sending process has effective user ID of superuser, the kernel sends the signal to all processes except processes 0 and 1.
    - If pid is a negative integer but not –1, the kernel sends the signal to all processes in the process group equal to the absolute value of pid

2. Signal

```
#include<signal.h>
Main()
{
    register int I;

    setpgrp();
    for( I=0; I<10; I++)
    {
            if( fork() == 0 )
            {
                /* child proc */
               if( I & 1 )
                       setpgrp();
                printf("pid = %d pgrp = %d\n", getpid(), getpgrp() );
                pause();            /* sys call to suspend execution */
            }
    }
    kill(0, SIGINT );
}
```

The kernel sends the signal to the 5 "even" process that did not reset their process group, but the 5 "odd" process continue to loop

PROCESS TERMINATION

By executing the exit system call

An exiting process enters the zombie state, relinquishes its resources and dismantles its context except for its slot in the process table

Syntax for the call

- Exit(status);

Startup routine – return from the main function

The kernel may invoke exit internally for a process on receipt of uncaught signals as discussed above

3. Process terminati

Algorithm exit

```
Input: return code for parent process
Output:none
{
    ignore all signals;
    if( process group leader with associated control terminal )
    {
            send hangup signal to all members of process group;
            reset process group for all members to 0;
    }
    close all open files ( internal version of algorithm close );
    release current directory ( algorithm iput );
    release current ( changed ) root, if exists ( algorithm iput );
    free regions, memory associated with process ( algorithm freereg );
    write accounting record;
    make process state zombie
    assign parent process ID of all child processes to be init process (1);
            if any children were zombie, send death of child signal to init;
    send death of child signal signal to parent process;
    context switch;
```

AWAITING PROCESS TERMINATION

A process can synchronize its execution with the termination of a child process by executing the wait system call

The syntax for the system call

- Pid = wait(stat_addr);

Processes only wake up on receipt of signal

For any signal except “death of child”, the process will react as described above. However, if the signal is “ death of child” the process may respond differently

- In the default case, it will wake up from its sleep in wait, and sleep invokes algorithm issig to check for signals.
- If the process catches “death of child” signals, the kernel arranges to call the user signal-handler routine, as it does for other signals
- If the process ignores “death of child” signals, the kernel restarts the wait loop, free the process table slots of zombie children, and searches for more children.

4. Awaiting process termina

Algorithm wait

```
Input: address of variable to store status of exiting process
Output : child ID, child exit code
{
    if( waiting process has no child processes )
            return( error );
    for(;;)     /* loop until return from inside loop */
    {
            if( waiting process has zombie child )
            {
                pick arbitrary zombie child;
                add child CPU usage to parent;
                free child process table entry;
                return ( child ID, child exit code );
            }
            if( process has no children )
                return error;
            sleep at interruptible priority ( event child process exits );
    }
}
```

4. Awaiting process termina

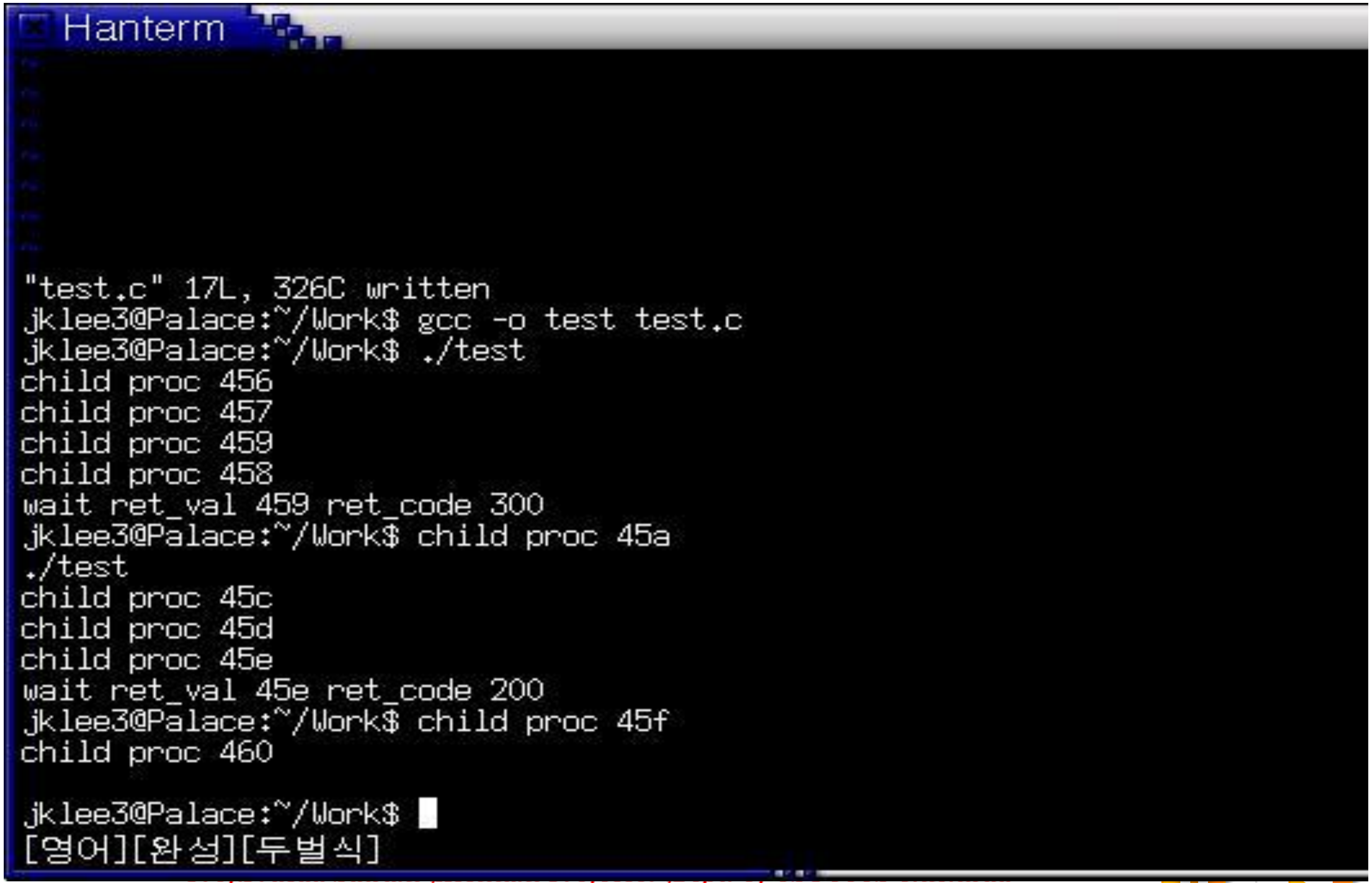
```
#include<signal.h>
Main( argc, argv)
        int argc;
        char *argv[];
{
    int I, ret_val, ret_code;

    if( argc >= 1 )
            signal( SIGCLD, SIG_IGN );    /* ignore death of child */]
    for( I = 0; I < 15; I++ )
            if( fork() == 0 )
            {
                /* child proc here */
               printf("child proc %x\n", getpid() );
               exit(I);
            }
    ret_val = wait( &ret_code );
    printf("wait ret_val %x ret_code %x\n", ret_val, ret_code );
}
```

Exit code in bits 8 to 15 of ret_code and returns the child processID for the wait call

Thus ret_code equals 256*I

4. Awaiting process termina



Hanterm

```
"test.c" 17L, 326C written
jklee3@Palace:~/Work$ gcc -o test test.c
jklee3@Palace:~/Work$ ./test
child proc 456
child proc 457
child proc 459
child proc 458
wait ret_val 459 ret_code 300
jklee3@Palace:~/Work$ child proc 45a
./test
child proc 45c
child proc 45d
child proc 45e
wait ret_val 45e ret_code 200
jklee3@Palace:~/Work$ child proc 45f
child proc 460

jklee3@Palace:~/Work$ 
[영어][완성][두벌식]
```

4. Awaiting process termina

```
#include<signal.h>
Main(argc, argv)
{
    char buf[256];

    if( argc != 1 )
            signal( SIGCLD, SIG_IGN );   /* ignore death of  children  */
    while( read(0, buf, 256 ) )
            if( fork() == 0 )
            {
                /* child proc here typically does something with buf  */
                exit(0);
            }
}
```

The parent process does not wait for the termination of the child process

If the parent makes the signal call to ignore "death of child" signals, Kernel will release the entries for the zombie processes automatically. Otherwise, zombie processes would eventually fill the maximum allowed slots of the process table

INVOKING OTHER PROGRAMS

The exec system call invokes another program, overlaying the memory space of a process with a copy of an executable file.

The syntax for the system call

- Execve(filename, argv, envp)

Process can access their environment via the global variable environ

The logical format of an executable file

- 1. The primary header describes how many sections are in the file, the start address for process execution, and the magic number, which gives the type of the executable file.
- 2. Section headers describe each section in the file, giving the section size, the virtual addresses the section should occupy when running in the system, and other information.
- 3. The sections contain the “data”, such as text, that are initially loaded in the process address space
- 4. Miscellaneous sections may contain symbol tables and other data, useful for debugging.

Algorithm exec

```
Input: (1) file name
       (2) parameter list
       (3) environment variables list
Output: none
{
    get file inode ( algorithm namei );
    verify file executable, user has permission to execute;

    read file headers, check that it is a load module;

    copy exec parameters from old address space to system space;
    for( every region attached to process )
            detach all old regions ( algorithm detach);
    for( every region specified in load module )
   {
            allocate new regions ( algorithm allocreg );
            attach the regions ( algorithm attachreg );
            load region into memory if appropriate ( algorithm loadreg );
    }
    copy exec parameters into new user stack region;
    special processing for setuid program, tracing;
    initialize user register save area for return to user mode;
    release inode of file ( algorithm iput );
}
```

5. Invoking other progr

```
Main()
{
    int status;
    if( fork() == 0 )
            execl( "/bin/date", "date", 0 );
    wait( &status );
}
```

When the child process is about to invoke the exec call,

- Its text region consists of the instructions for the program
- Its data region consists of the string "/bin/date" and "date"
- Its stack contains the stack frames the process pushed to get the exec call

Kernel finds the file "/bin/date"

Kernel then copies the strings "/bin/date" and "date" to an internal holding area and free text, data, stack regions

It allocates new text, data, stack region

Kernel reconstructs the original parameter list and puts it in the stack

5. Invoking other progr

Two advantages for keeping text and data separate:

- Protection
  - Kernel can set up hardware protection mechanisms to prevent processes from overwriting their text space
- Sharing
  - If process cannot write it text region, its text does nor change from the time the kernel loads it from the executable file.
  - If several processes execute a file they can, therefore, share one text region, saving memory

5. Invoking other progr

```
#include<signal.h>
Main()
{
    int I, *ip;
    extern f(), sigcatch();

    ip = (int *)f;      /* assign ip to address of function f */
    for( I = 0; I < 20; I++ )
            signal(I, sigcatch);
    *ip = 1;            /* attempt to overwrite address of f */
    prinf("after assign to ip\n");
    f();
}

F()
{
}

Sigcatch(n)
          int n;
{
    printf("caught sig %d\n", n );
    exit(1);
}
```

5. Invoking other progr

```
Algorithm xalloc            /* allocate and initialize text region */
Input: inode of executable file
Output: none
{
    if( executable file does not have separate text region)
            return;
    if( text region associated with text of inode)
    {
            /* text region already exists…attach to it */
            lock region;
            while( contents of region not ready yet )
            {
                /* manipulation of reference count prevents total removal of the region */
                increment region reference count;
                unlock region;
                sleep( event contents of region ready);
                lock region;
                decrement region reference count;
            }
            attach region to process ( algorithm attachreg );
            unlock region;
            return;
    }
    /* no such text region exists ---create one */
    allocate text region ( algorithm allocreg );    /* region is locked */
    if( inode mode has sticky bit set)
            turn on region sticky flag;
    attach region to virtual address indicated by inode file header ( algorithm attachreg );
    read file text into region ( algorithm loadreg );
    change region protection in per process region table to read only;          unlock
region   }
```

5. Invoking other progr

Traditional implementations of the system contain a text table that the kernel manipulates in the way just described for text regions.

The kernel increments the reference count of the inode associated with the region, because the kernel decrements the reference count once in iput at the end of exec

The capability to share text regions allow the kernel to decrease the startup time of an execd program by using the stick-bit.

5. Invoking other progr

The kernel remove the entries for sticky-bit text region

- 1. If a process opens the file for writing, the write operations will change the contents of the file, invalidating the contents of the region.
- 2. If a process changes the permission modes of the file(chmod) such that the sticky-bit is no longer set, the file should not remain in the region table.
- 3. If a process unlinks the file, no process will able to exec it any more because the file has no entry in the file system; hence no new processes will access the file's region table entry. Because there is no need for the text region, the kernel can remove it to free some resource
- 4. If a process unmounts the file system, the file is no longer accessible and no processes can exec it, so the logic of the previous case applies.
- 5. If the kernel run out of space on the swap device, it attempts to free available space by freeing stick-bit regions that are currently unused. Although other processes may need the text region soon, the kernel has more immediate needs.

7.6 THE USER ID OF A PROCESS

Real user ID identifies the user who is responsible for the running process

The effective use ID is used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes via the kill system call.

Setuid program is an executable file that has the setuid bit set in its permission mode field. – kernel sets the effective user ID fields in the process table and u area to the owner ID of the file.

The syntax for the setuid system call

- Setuid(uid)

6. The user ID of a proce

```
#include<fcntl.h>
Main()
{
    int uid, euid, fdmjb, fdmaury;

    uid = getuid();                /* get real UID */
    euid = geteuid();              /* get effective UID */
    printf( "uid %d euid %d\n", uid, euid );

    fdmjb = open("mjb", O_RDONLY );
    fdmaury = open("maury", O_RDONLY );
    printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury );

    setuid(uid);
    printf("after setuid(%d): uid %d euid %d\n", uid, getuid(), geteuid() );

    fdmjb = open("mjb", O_RDONLY );
    fdmaury = open("maury", O_RONLY);
    printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury );

    setuid(euid);
    printf("after setuid(%d): uid %d euid %d\n", uid, getuid(), geteuid() );
}
```

6. The user ID of a proce

Suppose the executable file produced by compiling the program has owner “maury”(usr ID 8319), it setuid bit is on, and all users have permission to execute it.

Assume the users “mjb”(user ID 5088)

```
User “mjb”

Uid 5088 euid 8319
Fdmjb –1 fdmaury 3
Afer setuid(5088): uid 5088 euid 5088
Fdmjb 4 fdmaury –1
After setuid(8319): uid 5088 euid 8319
```

```
User “maury”

Uid 8319 euid 8319
Fdmjb –1 fdmaury 3
Afer setuid(8319): uid 8319 euid 8319
Fdmjb -1 fdmaury 4
After setuid(8319): uid 8319 euid 8319
```

Login

mkdir

7.7 CHANGING THE SIZE OF A PROCESS

A process may increase or decrease the size of its data region by using the brk system call.

The syntax for the brk system call

- Brk(endds);
- Oldendds = sbrk(increment);

```
Algorithm brk
Input: new break value
Output: old break value
{
    lock process data region;
    if( region size increase )
            if( new region size is illegal )
            {
                unlock data region;
                return( error );
            }
    change region size ( algorithm growreg );
    zero out addresses in new data space;
    unlock process data region;   }
```

7. Changing the size of a proc

```
#include<signal.h>
Char *cp;
Int callno;

Main()
{
    char *sbrk ();
    extern catcher();

addr %u
    signal( SIGSEGV, catcher );
    cp = sbrk( 0 );
    printf("original brk value %u\n", cp );
    for(;;)
            *cp++ = 1;
}
```

```
catcher( signo )
        int signo;
 {
         callno++;
         printf("caught sig %d %dth call at

                       \n", signo, callno, cp );
          sbrk( 256 );
          signal( SIGSEGV, catcher );
  }
```

```
Original brk value 140924
Caught sig 11 1th call at address 141312
Caught sig 11 2th call at address 141312
Caught sig 11 3th call at address 143360
    … ( same address printed out to 10th call )
Caught sig 11 10th call at address 143360
Caught sig 11 11th call at address 145408
    … ( same address printed out to 18th call )
Caught sig 11 1th call at address 145408
Caught sig 11 1th call at address 145408
                        .
                        .
```

7.8 THE SHELL

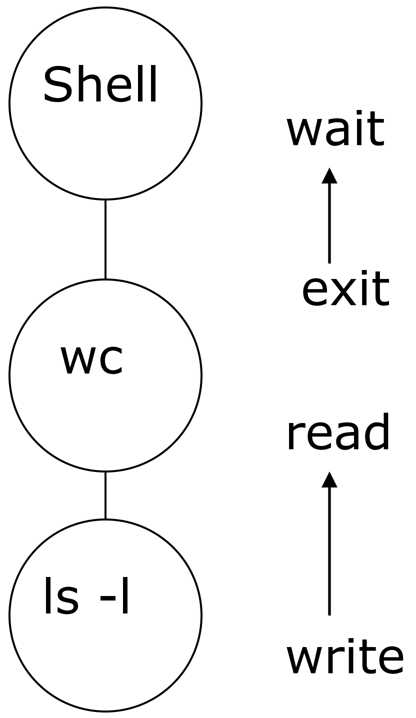
If the shell recognizes the input string as a built-in command, it executes the command internally without creating new processes; otherwise, it assumes the command is name of an executable file.

To run a process asynchronously(in the background), sets an internal variable amper

Nroff –mm bigdocument &

Nroff –mm bigdocment > output

Ls –l | wc



8. The shel

```
/* read command line until “end of file” */
While( read( stdin, buffer, numberchars ) )
{
    /* parse command line */
    if( /* command line contains & */)
            amper = 1;
    else
            amper = 0;
    /* for commands not part of the shell command language */
    if( fork() == 0 )
    {
            /* redirection of IO ? */
            if( /* redirect output */ )
            {
                fd = creat( newfile, fmask );
                close(stdout);
                dup(fd);
                close(fd);
                /* stdout is now redirected */
            }
            if( /* piping */ )
            {
                pipe( fildes);
                if( fork() == 0 )
                {
                        /* first component of command line */
                        close( stdout );
                        dup( fildes[1]);
                        close( fildes[1]);
                        close( fildes[0]);   /* stdout now goes to pipe */
                        execlp( command1, command1, 0 );   /* child process does command line
```

8. The shel

```
                    /* 2nd command component of command line */
                    close( stdin );
                    dup( fildes[0]);
                    close( fildes[0]);
                    close( fildes[1]);
                     /* standard input now comes from pipe  */
               }
               execve( command2, command2, 0 );

     }
     /* parent continues over here...
      * waits for child to exit if required
     */
     if( amper == 0 )
               retid = wait( &status );
}
```

7.9 SYSTEM BOOT AND THE INIT PROCESS

To initialize a system from an inactive state, an administrator goes through a "bootstrap" sequence:

Goal -To get a copy of the operating system into machine memory and to start executing it

On UNIX systems, the bootstrap procedure eventually reads the boot block(block 0) of disk, and loads it into memory

After the kernel is loaded in memory, the boot program transfers control to the start address of the kernel, the kernel starts running(algorithm start)

9. System boot and the init pro

```
Algorithm start
Input:none
Output:none
{
    initialize all kernel data structures;
    pseudo-mount of root;
    hand-craft environment of process 0;
    fork process 1:
    {
            /* process 1 in here */
            allocate region;
            attach region to init address space;
            grow region to accommodate code about to copy in;
            copy code from kernel space to init user space to exec init;
            change mode: return from kernel to user mode;
            /* init never gets here – as result of above change mode,
             * init exec's /etc/init and becomes a "normal" user process
             * with respect to invocation of system calls
             */
    }
    /* proc 0 continues here */
    fork kernel processes;
    /* process 0 invokes the swapper to manage the allocation of
     * process address space to main memory and swap devices.
     * This is an infinite loop; process 0 usually sleeps in the
     * loop unless there is work for it to do.
     */
     execute code for swapper algorithm;
}
```

9. System boot and the init pro

It constructs the linked lists of free buffers and inodes, constructs hash queues for buffers and inodes, initializes region structures, page table entries

The new process, process 1, running in kernel mode, creates its user-level context by allocating a data region and attaching it to its address space. It grows the region to its proper size and copies code from kernel address space to the new region.

Process 1 is commonly called init because it is responsible for initialization of new processes.

9. System boot and the init pro

```
Algorithm init          /* init process, process 1 of the system */
Input: none
Output:none
{
    fd = open("/etc/inittab", O_RDONLY );
    while( line_read( fd, buffer ) )
    {
            /*  read every line of file  */
            if( invoked state != buffer state )
                continue;      /* loop back to while */
            /* state matched */
            if( fork() == 0 )
            {
                execl("process specified in buffer");
                exit();
            }
            /* init process does not wait */
            /* loop back to while */
    }

    while( ( id = wait( (int *)0 ) != -1 )
    {
            /* check here if a spawned child died;
             * consider respawning it */
            /* otherwise, just continue */
    }
}
```

9. System boot and the init pro

The init process is a process dispatcher, spawning processes that allow users to log in to the system

Init reads the file and, if the state in which it was invoked matches the state identifier of a line, creates a process that executes the given program specification.

Init executes the wait system call, monitoring the death of its child processes and the death of processes "orphaned" by exiting parents.

User process, deamon process, kernel process