

UNIX INTERNALS

Ms. Radha Senthilkumar, Lecturer
Department of IT
MIT, Chromepet
Anna University, Chennai.

Processes

THE DESIGN OF THE UNIX OPERATING SYSTEM

Maurice J. bach Prentice Hall

Table of Contents

◆ Structure of process

- Process States and Transitions
- Layout of System Memory
- The Context of A Process
- Saving the Context of A Process
- Manipulation of the Process Address Space
- Algorithms for Sleep and Wakeup

◆ Process Control

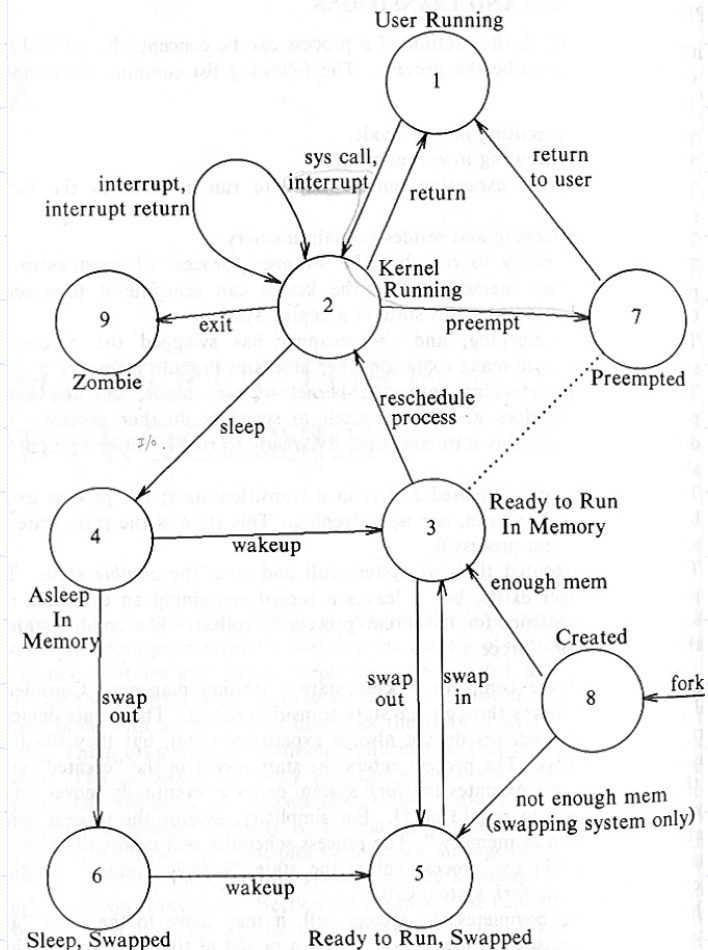
- Process Creation -- fork
- Signals
- Process Termination -- exit
- Awaiting Process Termination -- wait
- Invoking Other Programs -- exec
- The User ID of A Process
- Changing The Size of A Process -- brk
- The SHELL
- System Boot and The Init Process

◆ Process Scheduling

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" to state "kernel running".
- A process can *exit* of its own volition, but external events may dictate that it *exits* 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 allocates 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

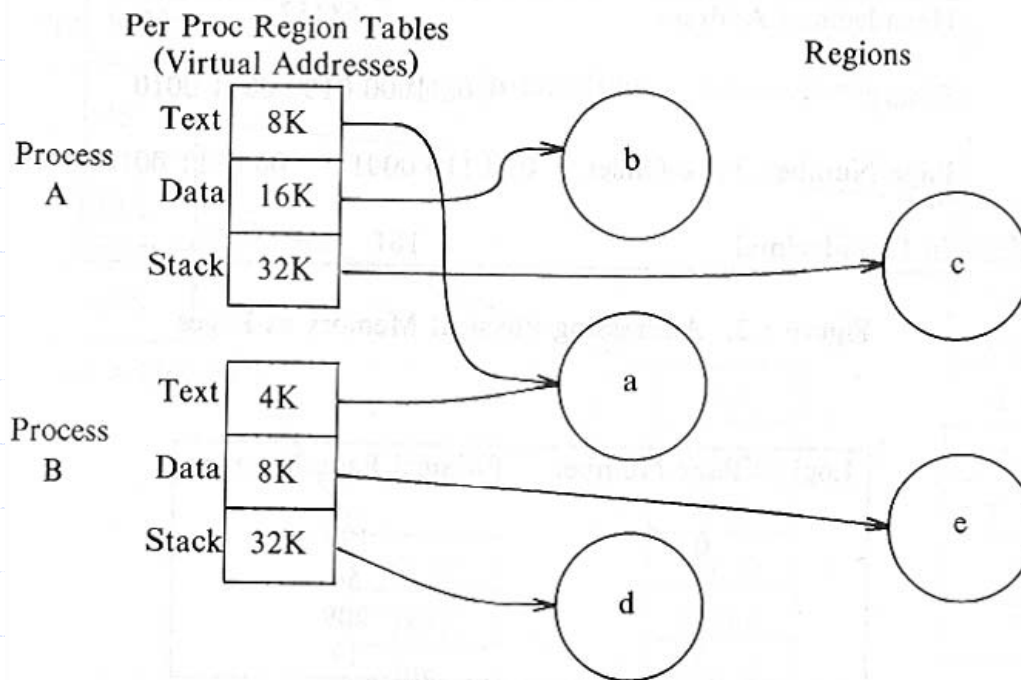
- A contiguous area of the virtual address space of a process
- It can be treated as a distinct object to be **shared/protected**.
- Thus text, data, and stack usually form separate regions of a process. – Several processes can share a region.

◆ Region Table

- It contains the information to determine where its contents are located in physical memory.
- Each pregon(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) (22-bit page no & 10-bit offset)
- 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

Logical Page Number	Physical Page Number
0	177
1	54
2	209
3	17

Pages & Page Tables

◆ Page Table

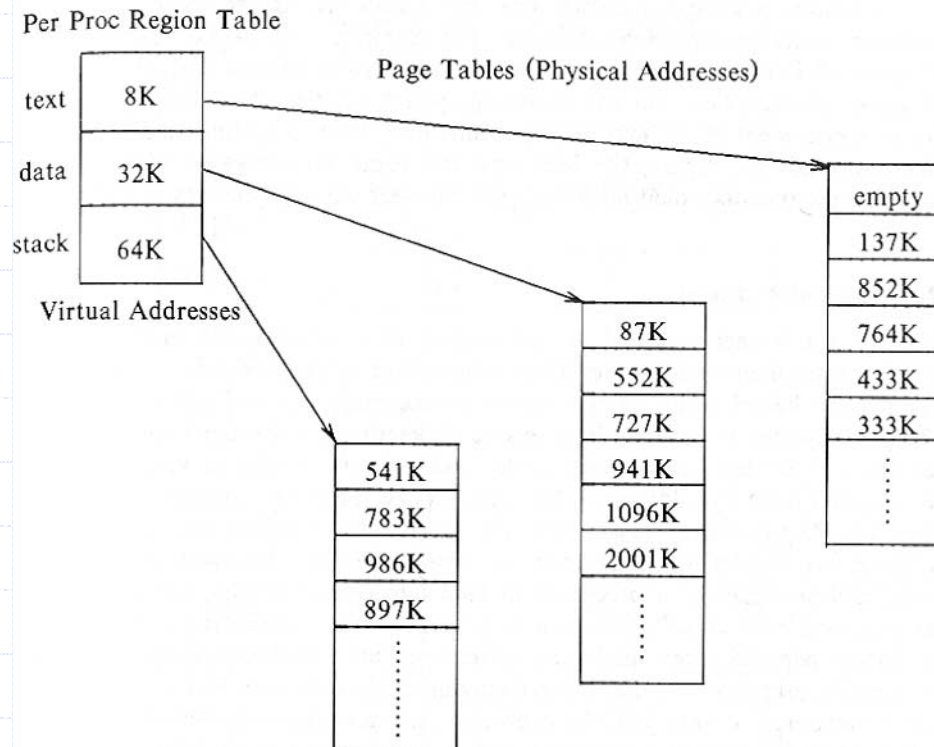
- When the kernel assigns physical pages of memory to a region, it need not assign the pages contiguously or in a particular order.
- The region table entry contains a pointer to a table of physical page numbers called a page table.
- The logical page number is the index into an array of physical page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



Page size = 1k

Virtual memory
address = 68432

Physical address =
?

The virtual address
start at 64k
(65536)

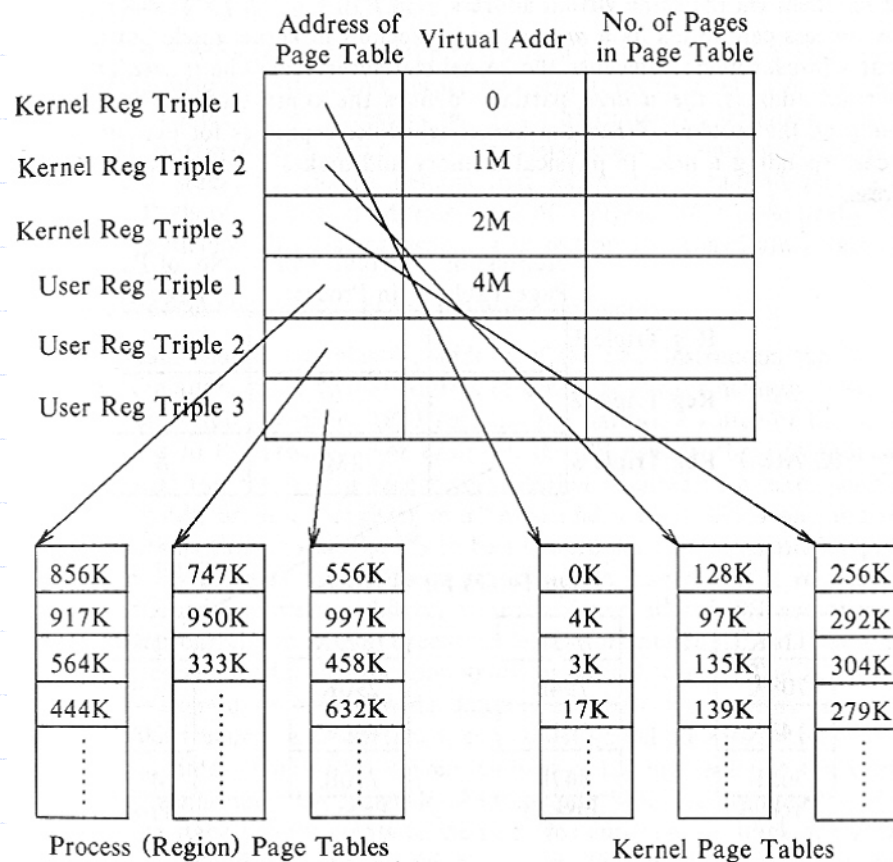
Layout of the Kernel

◆ 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 address.

■ Example:

- ◆ *u area* is 4K bytes long at Virtual Address 2M
- ◆ 1st – kernel text
- ◆ 2nd – kernel data
- ◆ 3rd – *u area* for process D

	Address of Page Table	Virtual Addr in Process	No. of Pages in Page Table
Reg Triple 1			
Reg Triple 2			
(U Area) Reg Triple 3		2M	4

Page Tables for U Areas			
114K	843K	1879K	184K
708K	794K	290K	176K
143K	361K	450K	209K
565K	847K	770K	477K
Proc A	Proc B	Proc C	Proc 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

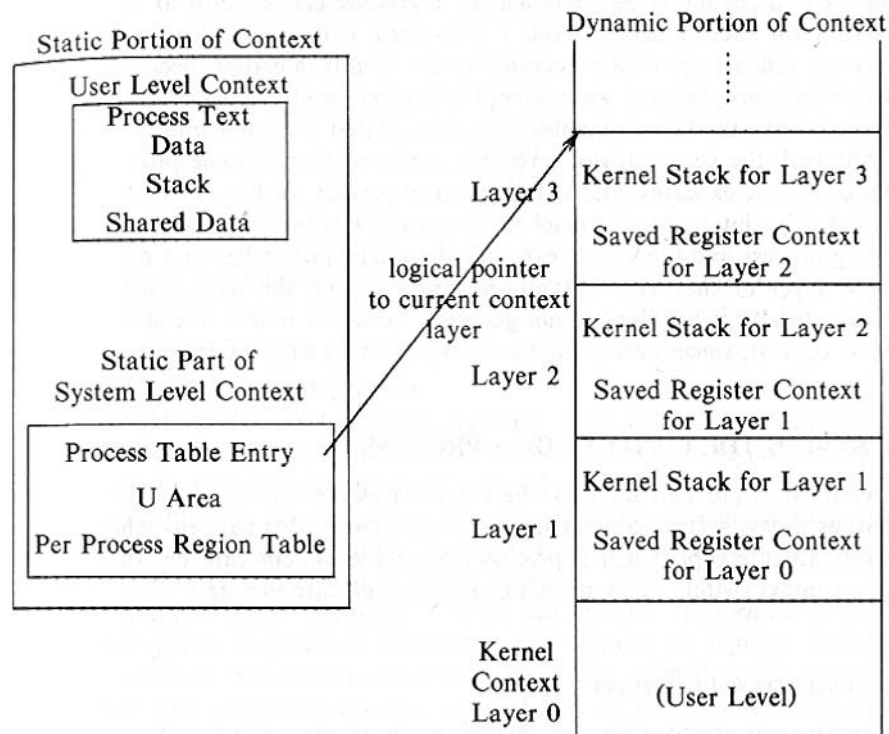
- Static part : process table entry, *u area*, region / page tables
- Dynamic part : kernel stack, system-level context layer (including register context)
- The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does a context switch.

◆ 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



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;  
}
```

Interrupts & Exceptions

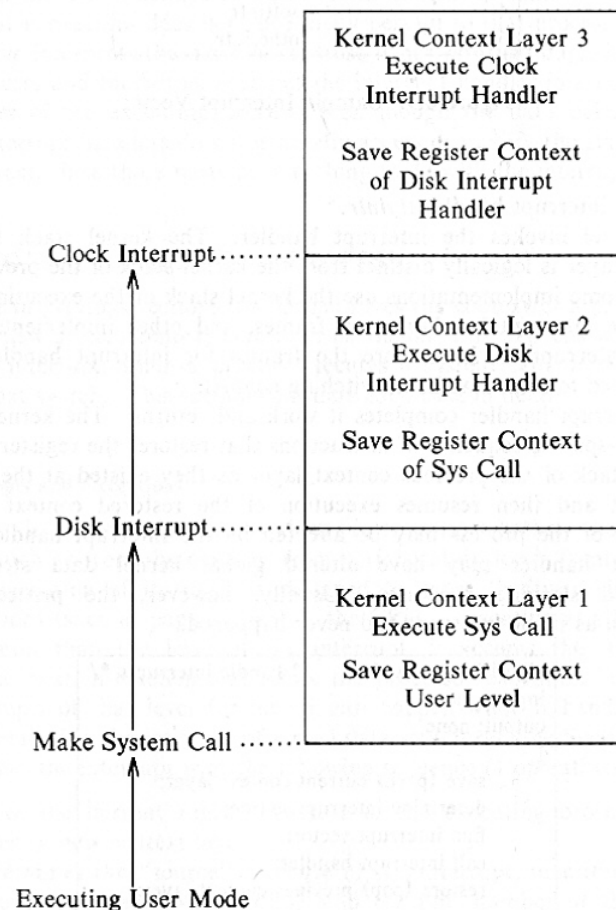
◆ Interrupt Vector

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

◆ Example of Interrupts system call -> disk intr. -> clock intr.

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

Interrupt Sequence



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;
}
```


System Call Interface

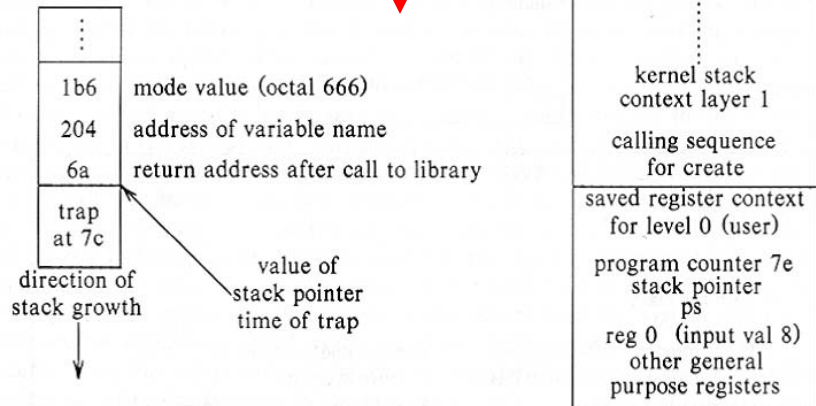
Example: Create System Call

```
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:	trap &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:	movq %d0, %a0	
146:	rts	# return from subroutine

Stack Configuration




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

Region Table

◆ Region Table Entry

- A pointer to the inode of the file whose contents were originally loaded into the region
- The region type (text, share memory, private data or stack)
- The size of the region
- The location of the region in physical memory
- The status of a region, which may be a combination of
 - ◆ Locked
 - ◆ In demand
 - ◆ In the process of being loaded into memory
 - ◆ Valid, loaded into memory
- The reference count, giving the number of processes that reference the region

Region Manipulation Operations

◆ Locking & Unlocking a Region

- The kernel can lock and allocate a region and later unlock it without having to free the region.

◆ Allocating a Region

- The kernel allocates a new region during *fork*, *exec*, and *shmget*(shared memory) system calls.
- With few exceptions, *allocreg* set the inode field in the region table entry to point to the inode of the executable file.

```
algorithm allocreg    /* allocate a region data structure */
input:  (1) inode pointer
        (2) region type
output: locked region
{
    remove region from linked list of free regions;
    assign region type;
    assign region inode pointer;
    if (inode pointer not null)
        increment inode reference count;
    place region on linked list of active regions;
    return(locked region);
}
```

Region Manipulation Operations

◆ Attaching a Region to a Process

- The kernel attaches a region during *fork*, *exec*, and *shmget* system calls to connect it to the address space of a process.
- If the region is not already attached to another process, the kernel allocates page tables for it, otherwise it uses the existing page tables.

```

algorithm attachreg    /* attach a region to a process */
input: (1) pointer to (locked) region being attached
       (2) process to which region is being attached
       (3) virtual address in process where region will be attached
       (4) region type
output: per process region table entry
{
    allocate per process region table entry for process;
    initialize per process region table entry:
        set pointer to region being attached;
        set type field;
        set virtual address field;
    check legality of virtual address, region size;
    increment region reference count;
    increment process size according to attached region;
    initialize new hardware register triple for process;
    return(per process region table entry);
}
    
```

Per Process Region Table

Page Table Addr	Proc Virt Addr	Size and Protect
	0	9
Entry for Text		
↓		
empty		
empty		
846K		
752K		
341K		
484K		
976K		
342K		
779K		

Region Manipulation Operations

◆ Changing the Size of a Region

- The kernel invokes the algorithm *growreg* to change the size of a region(as user stack grows).
- ☞ The kernel never invokes *growreg* to increase the size of a shared region that is already attached to several processes.
- ☞ Text regions and shared memory regions cannot grow after they are initialized.

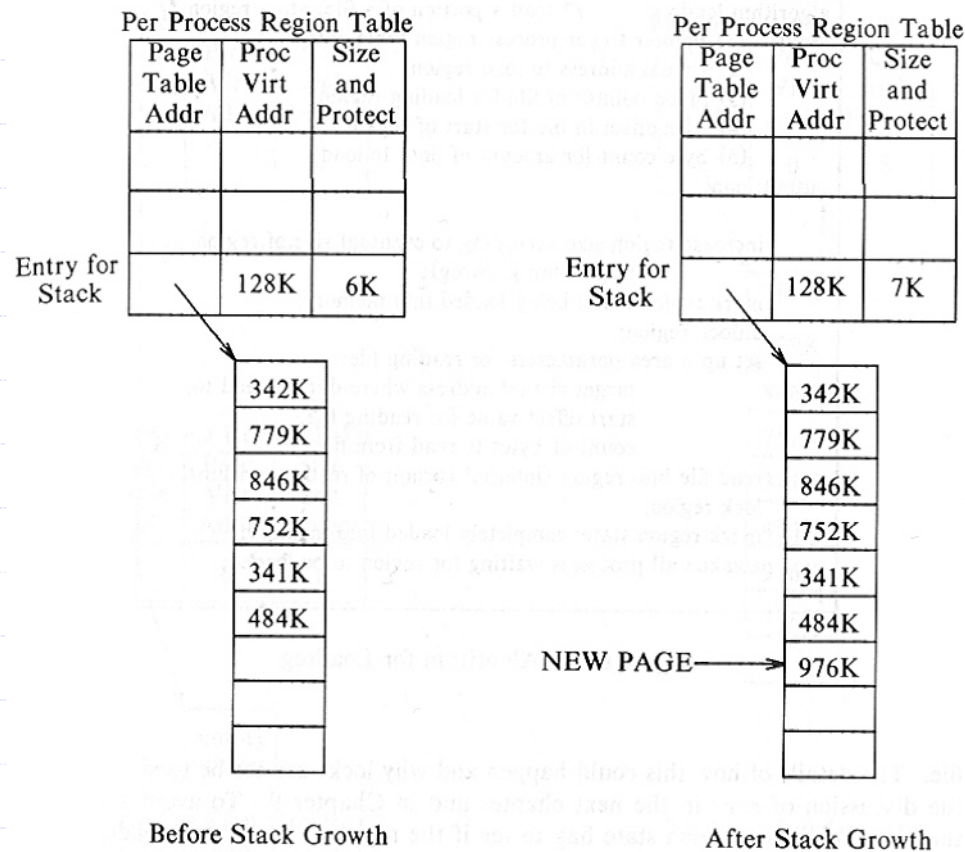
```

algorithm growreg    /* change the size of a region */
input:  (1) pointer to per process region table entry
        (2) change in size of region (may be positive or negative)
output: none
{
    if (region size increasing)
    {
        check legality of new region size;
        allocate auxiliary tables (page tables);
        if (not system supporting demand paging)
        {
            allocate physical memory;
            initialize auxiliary tables, as necessary;
        }
    }
    else    /* region size decreasing */
    {
        free physical memory, as appropriate;
        free auxiliary tables, as appropriate;
    }

    do (other) initialization of auxiliary tables, as necessary;
    set size field in process table;
}
    
```


Region Manipulation Operations

◆ Growing the Stack Region by 1K Bytes



Region Manipulation Operations

Loading a Region

- If the kernel does not support demand paging, it must copy the executable file into memory, loading the process regions at virtual addresses specified in the executable file.
- Creating a gap in the page table is used to cause memory faults when user programs access address 0 illegally.

```

algorithm loadreg      /* load a portion of a file into a region */
input:  (1) pointer to per process region table entry
        (2) virtual address to load region
        (3) inode pointer of file for loading region
        (4) byte offset in file for start of region
        (5) byte count for amount of data to load
output: none
{
    increase region size according to eventual size of region
        (algorithm growreg);
    mark region state: being loaded into memory;
    unlock region;
    set up u area parameters for reading file:
        target virtual address where data is read to,
        start offset value for reading file,
        count of bytes to read from file;
    read file into region (internal variant of read algorithm);
    lock region;
    mark region state: completely loaded into memory;
    awaken all processes waiting for region to be loaded;
}
    
```

Region Manipulation Operations

◆ Loading a Text Region

Per Process Region Table

Page Table Addr	Proc Virt Addr	Size and Protect
Text —		0

(a) Original Region Entry

Per Process Region Table

Page Table Addr	Proc Virt Addr	Size and Protect
	0	1

empty

(b) After First Growreg
Page Table with One Entry
for Gap

Per Process Region Table

Page Table Addr	Proc Virt Addr	Size and Protect
	0	8

empty

779K

846K

752K

341K

484K

976K

794K

(c) After 2nd Growreg

Region Manipulation Operations

◆ Freeing a Region

- When a region is no longer attached to any processes, the kernel can free the region and return it to the list of free regions.
- If the region is associated with an inode, the kernel releases the inode using algorithm *iput*, corresponding to the increment of the inode reference count in *allocreg*.

```
algorithm freereg    /* free an allocated region */
input:  pointer to a (locked) region
output: none
{
    if (region reference count non zero)
    {
        /* some process still using region */
        release region lock;
        if (region has an associated inode)
            release inode lock;
        return;
    }
    if (region has associated inode)
        release inode (algorithm iput);
    free physical memory still associated with region;
    free auxiliary tables associated with region;
    clear region fields;
    place region on region free list;
    unlock region;
}
```

Region Manipulation Operations

◆ Detaching a Region from a Process

- The kernel detaches regions in the *exec*, *exit*, and *shmdt* (detach shared memory) system calls.
- It updates the pregon entry and severs the connection to physical memory by invalidating the associated memory management.
- The address translation mechanisms thus invalidated apply specifically to the **process**, not to the region.

```
algorithm detachreg    /* detach a region from a process */
input:  pointer to per process region table entry
output: none
{
    get auxiliary memory management tables for process,
        release as appropriate;
    decrement process size;
    decrement region reference count;
    if (region reference count is 0 and region not sticky bit)
        free region (algorithm freereg);
    else    /* either reference count non-0 or region sticky bit on */
    {
        free inode lock, if applicable (inode associated with region);
        free region lock;
    }
}
```

Region Manipulation Operations

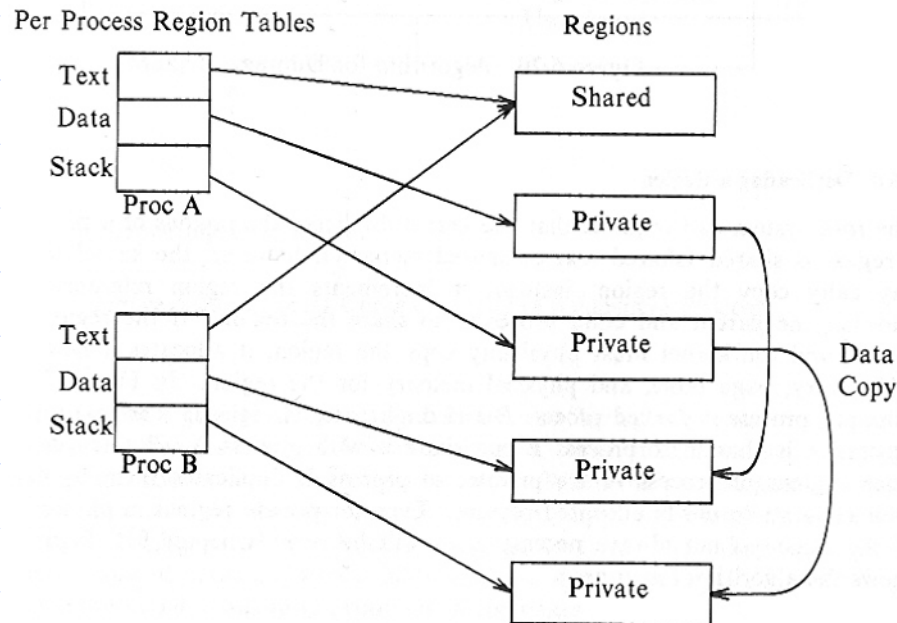
◆ Duplicating a Region

- The fork system call requires that the kernel duplicate the regions of a process.
- Even for private regions, a physical copy of the region is not always necessary, as will be seen Chapter “Demand Paging”.

```
algorithm dupreg    /* duplicate an existing region */
input:  pointer to region table entry
output: pointer to a region that looks identical to input region
{
    if (region type shared)
        /* caller will increment region reference count
         * with subsequent attachreg call
         */
        return(input region pointer);
    allocate new region (algorithm allocreg);
    set up auxiliary memory management structures, as currently
    exists in input region;
    allocate physical memory for region contents;
    "copy" region contents from input region to newly allocated
    region;
    return(pointer to allocated region);
}
```

Region Manipulation Operations

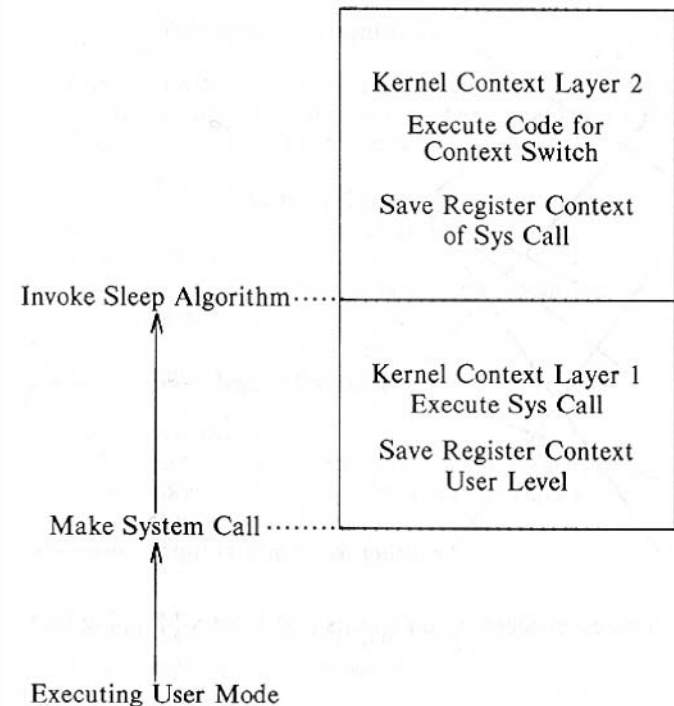
◆ Duplicating a Region (Example)



Sleep

◆ Sleeping Process

- When a process goes to sleep, it typically does so during execution of a system call.
- Processes also go to sleep when they incur page faults as a result of accessing virtual addresses that are not physically loaded; they sleep while the kernel reads in the contents of the pages.

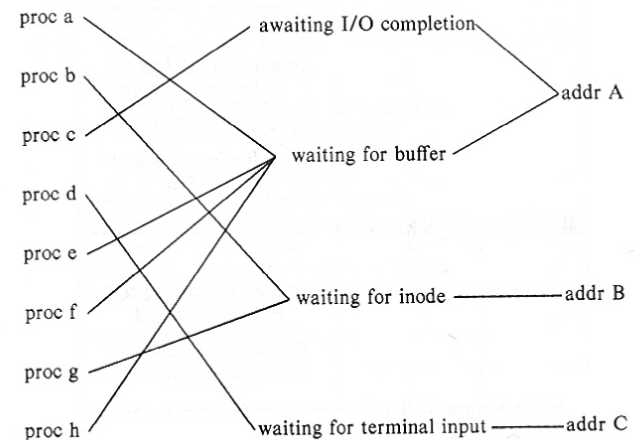


Sleep Events & Addresses

◆ Event Mapping into Address

- Processes are in the sleep state until the event occurs, at which time they wake up and enter a “ready-to-run” state (in memory or swapped out)
- Although the system uses the abstraction of sleeping on an event, the the set of events into a set of addresses.

☞ The mapping of multiple events into one address is rare and the running process usually frees the locked resource before the other processes are scheduled to run.



Algorithms for Sleep & Wakeup

◆ Process Sleep or Wait

- In the simple case (sleep cannot interrupted), the process does a context switch and is safely asleep.
- In the complex case (sleep can interrupted), more consideration is needed.

```

algorithm sleep
input: (1) sleep address
       (2) priority
output: 1 if process awakened as a result of a signal that process catches,
        longjump algorithm if process awakened as a result of a signal
        that it does not catch,
        0 otherwise;
{
    raise processor execution level to block all interrupts;
    set process state to sleep;
    put process on sleep hash queue, based on sleep address;
    save sleep address in process table slot;
    set process priority level to input priority;
    if (process sleep is NOT interruptible)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        reset processor priority level to allow interrupts as when
            process went to sleep;
        return(0);
    }

    /* here, process sleep is interruptible by signals */
    if (no signal pending against process)
    {
        do context switch;
        /* process resumes execution here when it wakes up */
        if (no signal pending against process)
        {
            reset processor priority level to what it was when
                process went to sleep;
            return(0);
        }
    }
    remove process from sleep hash queue, if still there;

    reset processor priority level to what it was when process went to sleep;
    if (process sleep priority set to catch signals)
        return(1)
    do longjmp algorithm;
}
    
```

Algorithms for Sleep & Wakeup

◆ Wakeup Algorithm

- It does not cause a process to be scheduled immediately; it only makes the process eligible for scheduling.

```
algorithm wakeup      /* wake up a sleeping process */
input:  sleep address
output: none
{
    raise processor execution level to block all interrupts;
    find sleep hash queue for sleep address;
    for (every process asleep on sleep address)
    {
        remove process from hash queue;
        mark process state "ready to run";
        put process on scheduler list of processes ready to run;
        clear field in process table entry for sleep address;
        if (process not loaded in memory)
            wake up swapper process (0);
        else if (awakened process is more eligible to run than
                 currently running process)
            set scheduler flag;
    }
    restore processor execution level to original level;
}
```


Algorithms for Sleep & Wakeup

◆ Interruptible Sleep Process

- A process may sometimes sleep on an event that is not sure to happen, and if so, it must have a way to regain control and continue execution.
- For such cases, the kernel “interrupt” the sleeping process immediately by sending it a signal.
- To distinguish the types of sleep states, the kernel sets the scheduling priority of the sleeping process when it enters the sleep state, based on the sleep priority parameter.
- The process should not continue normally after waking up from its sleep, because the sleep event was not satisfied.
- The kernel invokes the sleep algorithm with a special priority parameter that suppresses execution of the longjmp to allow the kernel to clean up local data structures.

Process Control

◆ Control of Process Context

- **fork** : create a new process
- **exit** : terminate process execution
- **wait** : allow parent process to synchronize its execution with the exit of a child process
- **exec** : invoke a new program
- **brk** : allocate more memory dynamically
- **signal** : inform asynchronous event
- major loop of the **shell** and of **init**

System Calls Dealing with Memory Management				System Calls Dealing with Synchronization			Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Process Creation

◆ Process Creation

- the only way to create a new process in UNIX is *fork*
 - ◆ parent process : the process that invokes the *fork*
 - ◆ child process : the newly created process
- Process 0 is the only process not created via *fork*

◆ *fork* system call

- `pid = fork();`
 - ◆ `pid` : to parent, the child process ID
to child process, 0

Sequence of Operations for fork

1. It allocates a slot in 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. Since certain portion of process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory
4. It increments file and inode table counters for files associated with the process.
5. It returns the ID number of child process to the parent process, and a 0 value to the child process.

Algorithm for fork

input : none

output : to parent process, child PID number
to child process, 0

{

check for available kernel resources;

get free process table slot, unique PID number;

check that user not running too many process;

mark child state "being created";

copy data from parent process table 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;

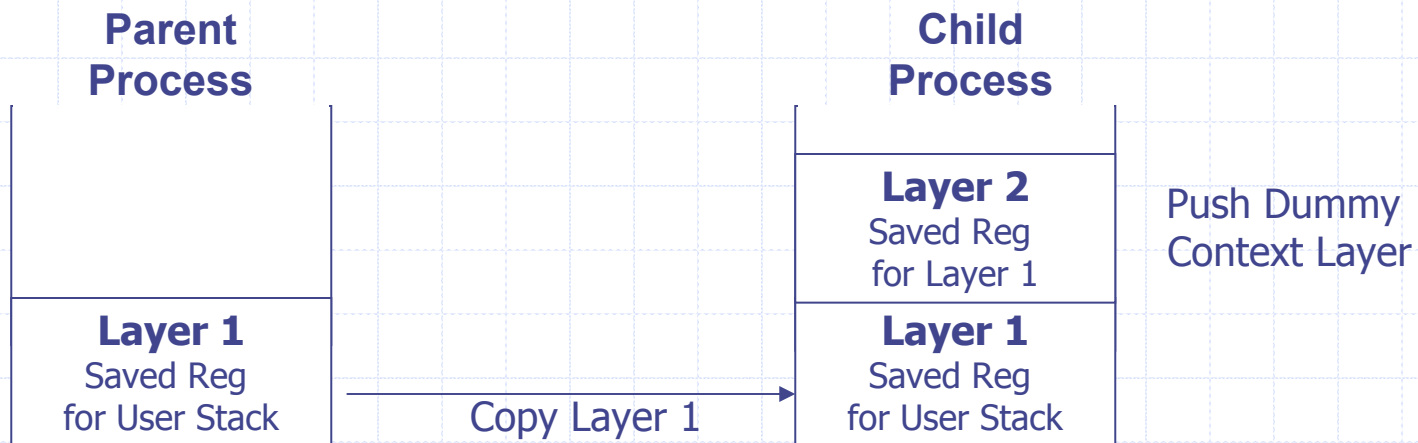
dummy context contains data allowing child process
to recognize itself, and start from here
when scheduled;

Algorithm for fork(Cont.)

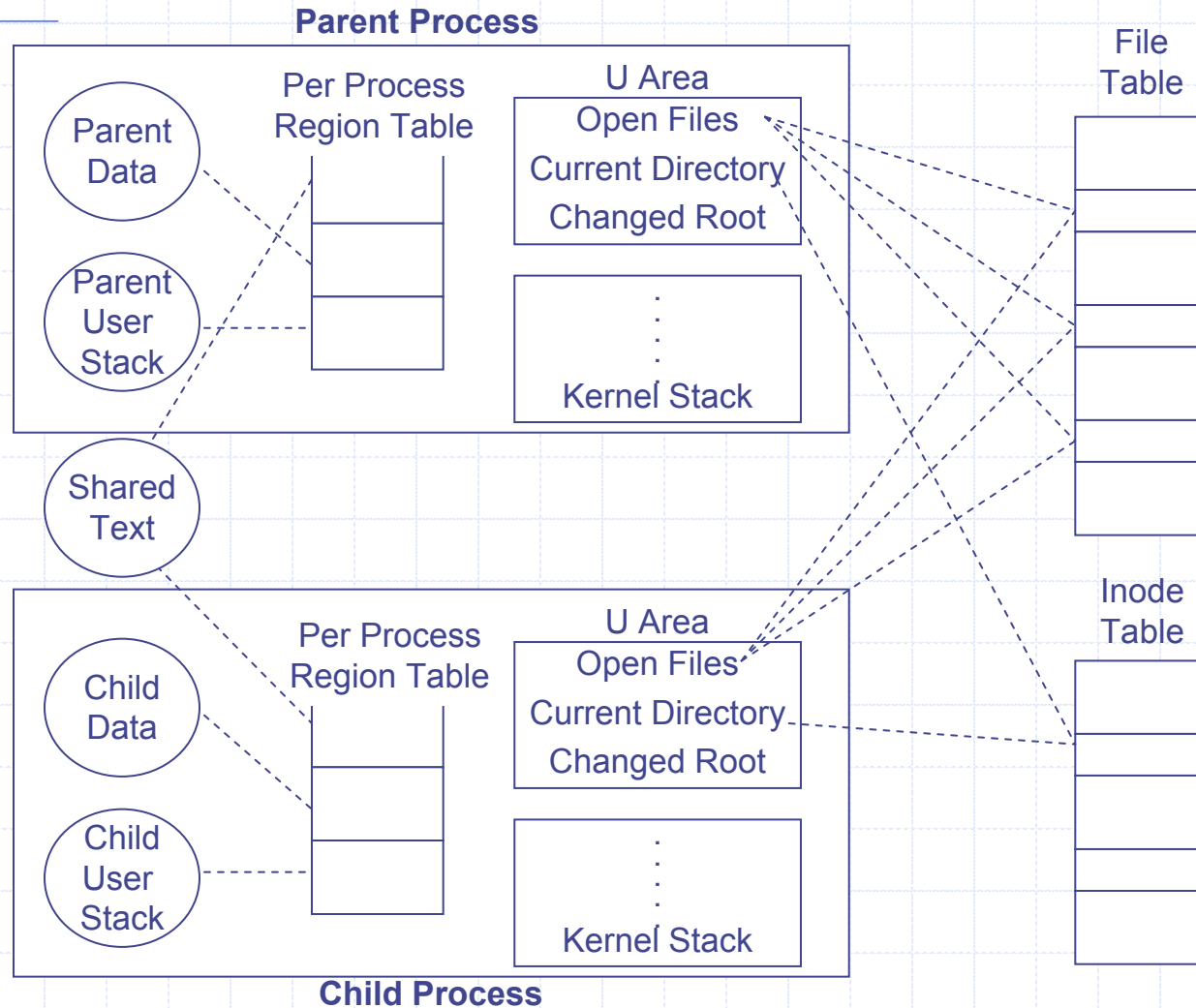
```
if ( executing process is parent )
{
    change child state to "ready to run";
    return(child ID);          /* from system to user */
}
else /* executing process is the child process */
{
    initialize u area timing fields;
    return(0);                 /* to user */
}
}
```


Dummy System Level Context Layer

- ◆ Kernel copies kernel context layer 1 to child process.
- ◆ push dummy system level context layer 2 onto child system level context.
- ◆ containing the saved register context for context layer 1.
- ◆ sets the program counter and other registers in the saved register.
- ◆ the child process can **restore the child context** even though it had never executed before(stored in layer 2).
- ◆ the child process can **recognize itself** as the child(stored in layer 1).



Fork Creating a New Process Context



Example of Sharing File Access

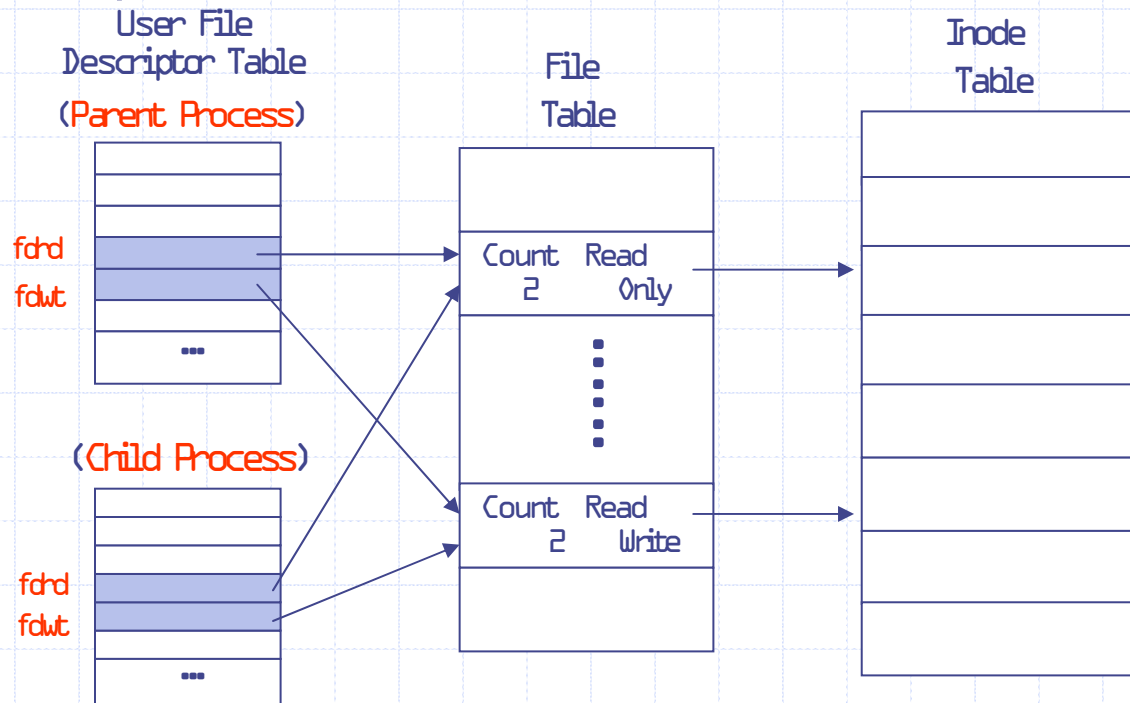
```
#include <fcntl.h>
int fdrd, fdwt;
char c;
```

```
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 process execute same*/
    rdwt();
    exit(0);
}
```

```
rdwt()
{
    for(;;)
    {
        if (read(fdrd,&c,1)!=-1)
            return;
        write(fdwt,&c,1);
    }
}
```

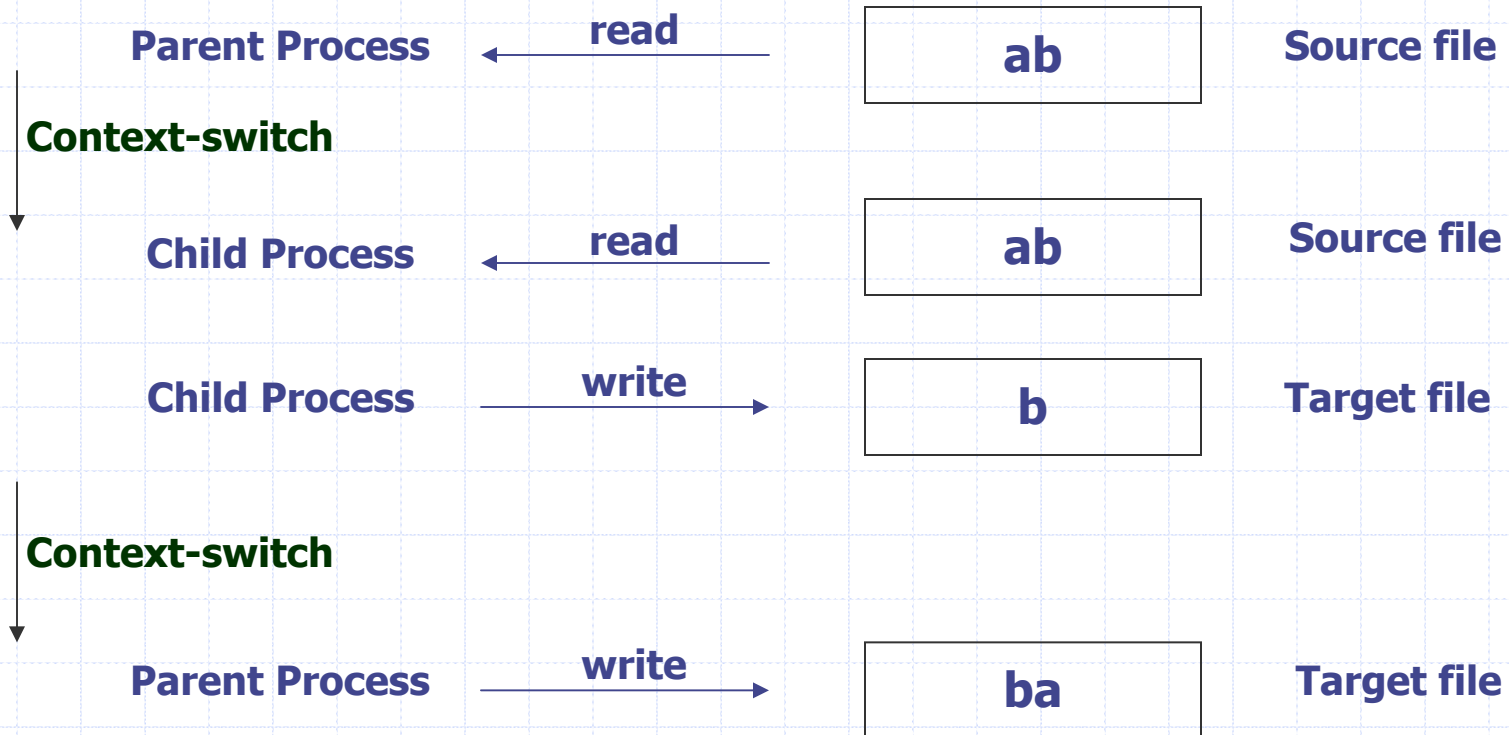
Example of Sharing File Access(Cont.)

- ◆ *fdrd* for both process refer to the file table entry for the source file(*argv[1]*)
- ◆ *fdwt* for both process refer to the file table entry for the target file(*argv[2]*)
- ◆ two processes never read or write the same file offset values.



Example of Sharing File Access(Cont.)

- the contents of target file depends on the order of the kernel scheduled the process.

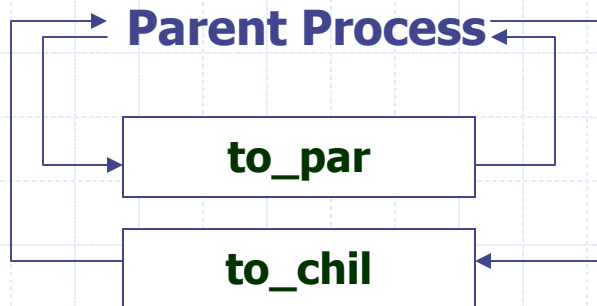


Use of Pipe, Dup, and Fork

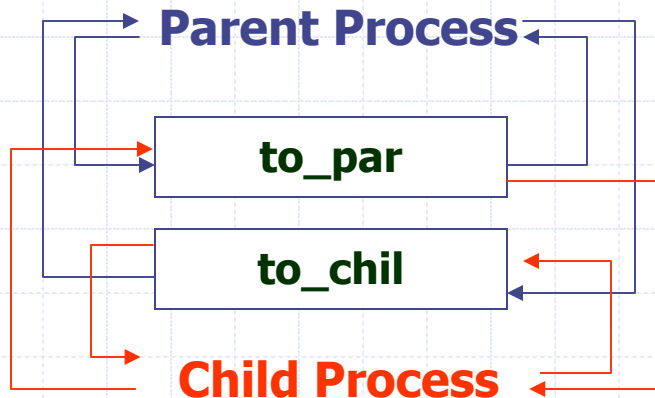
```
#include <string.h>
char string[]="hello-world";
main()
{
    int count, i;
    int to_par[2], to_chil[2];
    char buf[256];
    pipe(to_par);
    pipe(to_chil);
    if (fork()==0)
    {
        /* child process executes here */
        close(0);
        dup(to_chil[0]);
        close(1);
        dup(to_par[1]);
        close(to_par[1]);
        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);
    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));
    }
}
```


Use of Pipe, Dup, and Fork(Cont.)

1. `pipe(to_par); pipe(to_chil)`



2. `fork()`

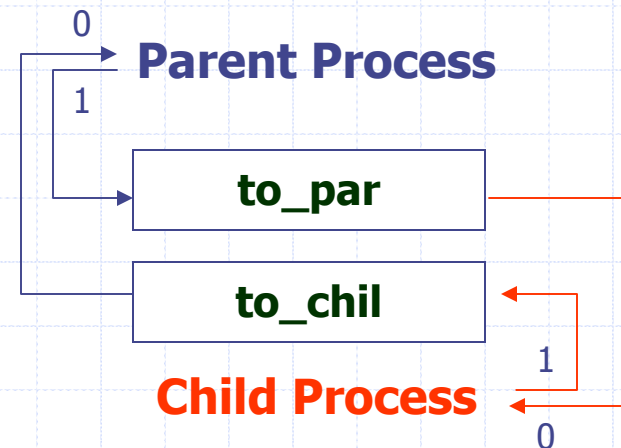


3. Parent Process

```
close(0);
dup(to_chil[0]);
close(1);
dup(to_par[1]);
```

Child Process

```
close(1);
dup(to_chil[1]);
close(0);
dup(to_par[0]);
```



◆ the result is regardless of the order that processes execute

Signals

◆ Informs the occurrence of asynchronous event.

- from process to process : *kill* system call.
- from kernel to process internally.
- 19 Signals in System V.

◆ Treatment of Signals

- How the kernel sends a signal to a process.
- How the process handles a signal
- How a process controls its reaction to the signal

How to Send a Signal to Process

◆ Send a Signal

- sets a bit in the signal field in process table entry.
- process can remember different type of signal.
- process can not remember how many signals it receives of particular type.
- u area contains an array of signal-handler fields.
kernel stores the address of the user-function in the field.

◆ Check for Signal

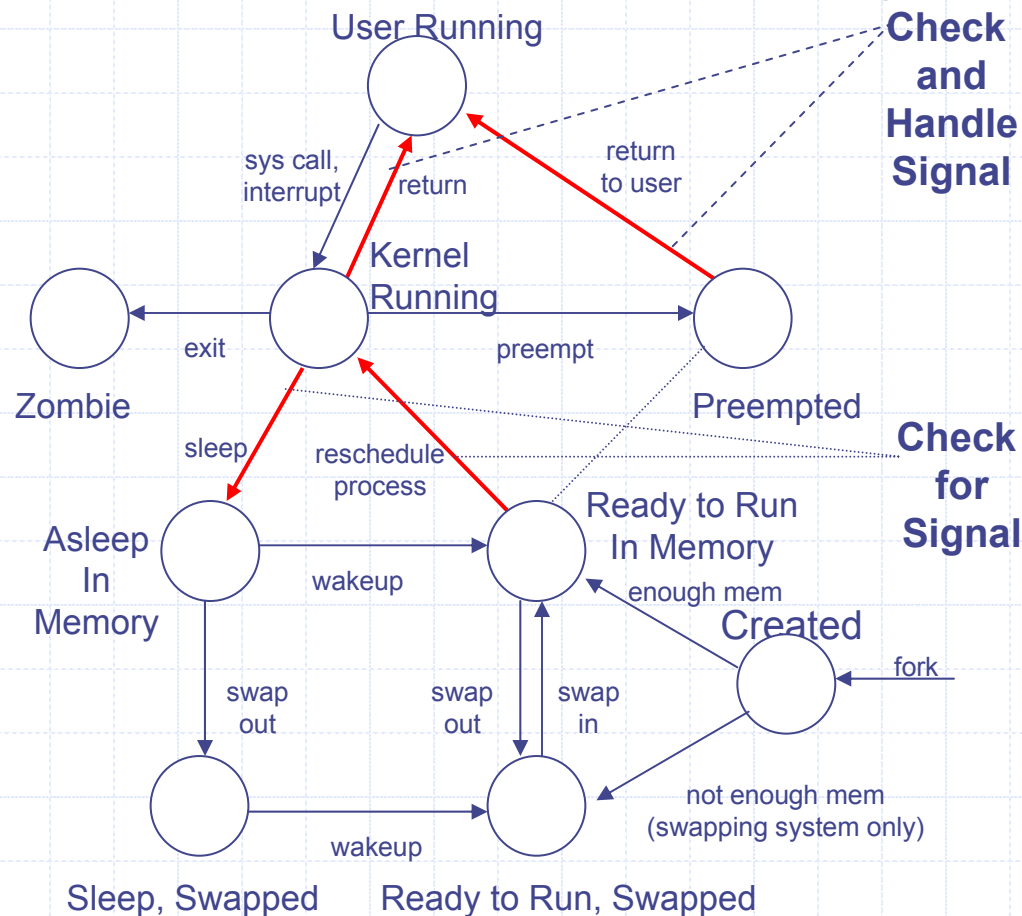
- about to return from kernel mode to user mode
- enters or leaves the sleep state at a suitably low scheduling priority.

◆ Handle Signals

- only when returns from kernel mode to user mode

How to Send a Signal to Process(Cont.)

- ◆ no effect on a process running in the kernel mode
- ◆ a process never executes in user mode before handling outstanding signals



Algorithm for issig

```

algorithm issig                                /* test for receipt of signals */
input : none
output : true, if process receives 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 (catching death of child signals)
                return(true);
        }
        else if (not ignoring signal)
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);
}
    
```

Handling Signals

◆ Handling Signals

- kernel handles signals in the context of the process that received signals.
- process must run to handle signals

◆ Three Cases

- exit(default action)
- ignores signals
- executes a particular user function

◆ *signal* System call

- specify special action to take on receipt of certain signal
- `oldfunction = signal(signum, function);`
 - ◆ `signum` : signal number
 - ◆ `function` : the address of function to invoke on receipt of signal
 - 0 : exit
 - 1 : ignore
 - ◆ `oldfunction` : the most recently specified to call to `signum`

Algorithm for Handling Signal

1. determines **signal type**
2. **turns off signal bit** in the process table entry
3. if receives a signal to **ignore**
 - continues as if the signal has never occurred.
4. If signal handling function is set to its **default** value,
 - kernel dumps core image for signals that imply something is wrong with process and exit
 - kernel does not dump core for signals that do not imply error.
5. If receives **a signal to catch**,
 - accesses the user saved register context, and find the program counter and stack pointer
 - clears the signal handler field in u area(*undesirable side-effects*)
 - kernel creates a new stack frame and writes a program counter and stack pointer from user saved register context
 - kernel changes the user register context: program counter to the address of signal catcher function, and stack pointer to account for the growth of the user stack.

Source Code for a Program that Catches Signals

```
#include <signal.h>
```

```
main()
```

```
{
```

```
    extern catcher();
```

```
    signal(SIGINT, catcher);
```

```
    kill(0,SIGINT);
```

```
}
```

Address of *kill* system call : 10c

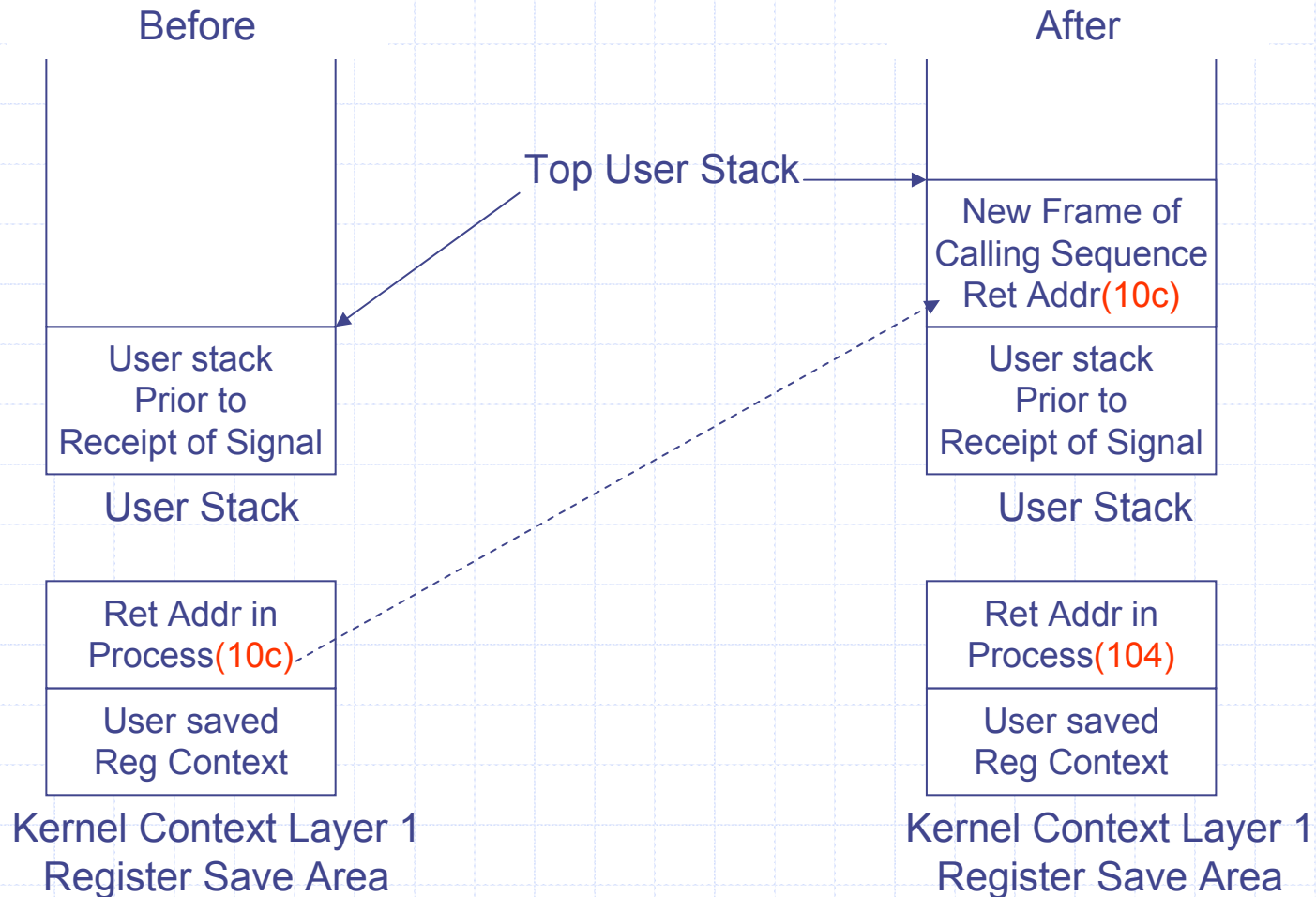
Address of catcher : 104

```
catcher()
```

```
{
```

```
}
```

User Stack and Kernel Stack



Algorithm for psig

```

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 had called signal sys call to ignore this signal)
        return;                                     /*done*/
    if (user specified function to handle this 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 filed of user saved register context;

        return;
    }
}

```

Algorithm for psig(Cont.)

```
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;
}
```

Race Condition in Catching Signal

```

#include <signal.h>
sigcatcher()
{
    /* print poc id */
    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);      /* lib function to delay 5 secs
        */

        ppid = getppid(); /* get parent id */
        for(;;)
    }

    /*lower priority, greater chance of exhibiting
    race*/
    nice(10);
    for(;;)
    ;
}

```


Race Condition in Catching Signal(Cont.)

◆ Race Condition

- child sends a SIGINT to parent
- parent catches signal and call signal catcher, but the kernel preempts before call the signal system call
- child sends SIGINT to parent again
- parent receives second SIGINT but not made arrangements to catch the signal. It exits

◆ Solution

- does not clear signal field
 - ◆ user stack could grow out of bounds because of nested calls to signal catcher
- reset signal handler function to ignore until user specify
 - ◆ loss of information

■ BSD system allows a process to block or unblock receipt of signals with new system call

Process Groups

◆ Process Group ID

- to identify groups that should receive a common signal for certain events
- save in the process table
- child retains process group ID of its parent during *fork*

◆ *setpggrp* system call

- initialize process group number and sets it equal to the value of its process ID
- `grp = setpggrp()`
 - ◆ `grp` : new process group number

Sending Signals from Processes

◆ *kill* system call

■ `kill(pid, signal)`

- ◆ `pid` : set of processes to receive the signals
- ◆ `signal` : signal number being sent

■ `pid` and processes

- ◆ if `pid > 0`, to the process with process ID *pid*.
- ◆ If `pid == 0`, to all processes in the sender's process group.
- ◆ If `pid == -1`, to all processes whose real user ID is equal to effective user ID of senders. If sending process has effective user ID of super user, sends the signal all processes except process 0 and 1.
- ◆ If `pid < 0` and `pid != -1`, to all processes in the process group equal to the absolute value of *pid*.

■ Failure

- ◆ if the sending process does not have effective user ID of super user, or its real user ID or effective user ID do not match the real or effective user ID of the receiving process.

Sample Use of setpgrp

```
#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(), getppid());
```

```
                pause();
```

```
        }
```

```
    }
```

```
    kill(0,SIGINT);
```

◆ Kernel sends the signal to the 5 "even" processes that did not reset their process group

◆ 5 "odd" processes continue to loop

Process Termination

◆ *exit* system call

- process terminate by *exit* system call
 - ◆ enters the zombie status
 - ◆ relinquish resources.
 - ◆ dismantles its context except for its slot in the process table.
- `exit(status);`
 - ◆ `status` : the value returned to parent process
- call `exit` explicitly or implicitly(by startup routine) at the end of program.
- kernel may invoke internally on receipt of uncaught signals.
In this case, the value of `status` is the signal number.

Algorithm for Exit

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 input);

release current(changed) root, if exists (algorithm input);

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 to parent process;

context switch;

}

Awaiting Process Termination

◆ *wait* system call

- synchronize its execution with the termination of a child process
- `pid = wait(stat_addr);`
 - ◆ `pid` : process id of the zombie child process
 - ◆ `stat_addr` : address of an integer to contain the exit status of the child

Algorithm for Awaiting Process Termination

1. searches the zombie child process
2. If no children, return error
3. if finds zombie children, extracts PID number and exit code
4. adds accumulated time the child process executes in the user and kernel mode to the fields in u area
5. Release process table slot

Algorithm for Wait

algorithm wait

input : address of variables to store status of exiting process

output : child ID, child exit code

```
{
    if (waiting process has no child process)
        return(error);
    for(;;)
    {
        if (waiting process has zombie child)
        {
            pick arbitrary zombie child;
            add child CPU usage to parent;
            free child process table entry;
            return(childID,child exit code);
        }
        if (process has no child process)
            return(error);
        sleep at interruptible priority(event child process exits);
    }
}
```

Sleeping at an interruptible priority

- ◆ If executing wait but not zombie process, the process sleeps at an interruptible priority until arrives a signal
- ◆ kernel does not contain an explicit wake up call for a process sleeping in wait
- ◆ sleeping process can only wake up on receipt signal
- ◆ Response to Death of Child Signal
 - default : wake up from its sleep in *wait*, and *sleep* invokes algorithm *issig* check for signals. (*issig* returns false on death of child signal) the kernel does not long jump and returns to wait and release the process table slot.
 - catches : the kernel arranges to call the user signal-handler routine
 - ignores : restarts the wait loop, frees the process table slots of zombie children, and searches for more children

Algorithm for issig

```

algorithm issig                                /* test for receipt of signals */
input : none
output : true, if process receives 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 (catching death of child signals)
                return(true);
        }
        else if (not ignoring signal)
            return(true);
        turn off signal bit in received signal field in process table;
    }
    return(false);                             /* default and ignores */
}

```

Example of Wait and Ignoring Death of Child Signal

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

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


Example of Wait and Ignoring Death of Child Signal(Cont.)

◆ Case 1(argc is 1) : **wait for only one process**

- parent create 15 child processes and wait
- child process exits with return code 1
- kernel searches zombie child process and returns exit code

◆ Case 2(argc > 1) : **wait for all child processes**

- ignores "death of child signal"
- assume parent process sleeps in wait before any child process exit
- when child process exit, sends "death of child signal" to parent process
- the parent process wakes up, but because of ignoring "death of child" signal kernel removes the process table entry and return to wait

Invoking Other Programs

◆ *exec* system call

- invokes another program, overlaying the memory space of a process with a copy of an executable file
- `execve(filename, argv, envp)`
 - ◆ `filename` : the name of executable file being invoked
 - ◆ `argv` : a pointer to an array of character pointers that are parameters to the executable program
 - ◆ `envp` : a pointer array of character pointers that are environment of the executed program
- several library functions that calls *exec* system call
`execl`, `execv`, `execle`...

Algorithm for Invoking Other Programs

1. accesses the file via algorithm **namei**
2. determines if it is an executable, regular file, user has permission
3. **reads the file header** to determine the layout of file
4. **copies the the arguments of exec** from the old memory space to a temporary buffer
5. **detaches the old regions** of the process using algorithm **detachreg**
6. **allocates and attaches regions** for text and data, loading the contents of the executable file into memory (**algorithm allocreg, attachreg, loadreg**)
7. allocates a region for process **stack**, attaches it to the process, and allocates the memory to **store exec parameters**
8. sets the saved register context for user mode, specifically setting the initial **user stack pointer and program counter**
9. **releases inode** that was originally allocated in the namei algorithm (**iput**)

Image of an Executable File

1. The primary Header: Magic Number which gives the type of the executable file, number of sections, initial register values.

2. Section Header : Section Size, Virtual Address the section occupy when running.

3. Data, such as text

4. Miscellaneous sections may contain symbol tables and other useful for debugging.

Primary Header

Section 1 Header

Section 2 Header

⋮

Section n Header

Section 1

Section 2

⋮

Section n

Magic Number
Number of Sections
Initial Register Values
Section Type
Section Size
Virtual Address
Section Type
Section Size
Virtual Address
⋮
Section Type
Section Size
Virtual Address
Data(e.g. text)
Data
⋮
Data
Other Information

Algorithm for Exec

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 programs, tracing;
  initialize user register save area for return to user mode;
  release inode of file(iput);
}
```

Protection

- ◆ two advantages for keeping text and data separate are protection and sharing
- ◆ Protection
 - if data and text were in same region, the system could not prevent a process from overwriting its instruction
 - if text and data are in separate regions, the kernel can set up hardware protection mechanism to prevent processes from overwriting their text space
 - if process attempts to overwrite its text space, it incurs a protection fault that typically results in termination of the process

Example of Program Overwriting its Text

```
#include <signal.h>
main()
{
    int i, *ip
    extern f(), sigcatch();
    ip = (int*)f;
    for( i=0;i<20;i++)
        signal(i, sigcatch);
    *ip=1;
    printf("after assign to ip\n");
    f();
}

f()
{
}

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

Example of Program Overwriting its Text (Cont.)



- ◆ if program compiled so that text and data in separate regions
 - incurs protection fault in $*ip=1$
 - writing its write-protected text region
 - sends SIGBUS signal on an AT&T 3B20 computer
 - process catches the signal and *exits* without executing the print statement in *main*
- ◆ if program compiled so that text and data are in same region
 - kernel overwrites 1 in the address of function $f(*ip=1)$
 - f contains value 1
 - when calls $f()$, send SIGILL signal and the process *exits*.
- ◆ Having instructions and data in separate regions make it easier to protect against addressing errors

Sharing

- ◆ if a process can not write its text region, its text does not change
- ◆ if several processes execute a file, they can share one text region, saving memory
- ◆ when allocates text region in *exec*, it checks if the executable file allows its text to be shared(Magic Number)
- ◆ if sharable, use *xalloc* algorithm

Algorithm for xalloc

- ◆ searches active region list for the file's text region (inode of region == inode of the executable file)
- ◆ if no such region exists
 - allocates a new region (algorithm allocreg)
 - attaches to process (algorithm attachreg)
 - loads into memory (algorithm loadreg)
 - changes its protection to read-only
- ◆ if such region exists
 - makes sure that the region is loaded into memory (otherwise sleep)
 - attach it to process

Algorithm for xalloc

algorithm xalloc

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)
    {
        /* lock 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);
    }
}
```

Algorithm for xalloc(Cont.)

```
unlock region;
return;
}
/* no such text regions exists--create one */
allocate text region(algorithm allocreg); /* region is locked */
if (inode mode has sticky bit set)
    turn on region sticky flags;
attach region to virtual address indicated by inode file header;(algorithm attachreg)
if (file specially formatted for paging system)
    /* Chapter 9 discusses this case */
else
    /* not formatted for paging system */
    read file text into region(algorithm loadreg)
change region protection in per process region table to read only;
unlock region;
}
```

Increment inode reference count in allocreg

algorithm exec

```
{
    get file inode(algorithm namei)
    :
    allocate new regions(algorithm allocreg);
    :
    release inode of file(iput);
}
```

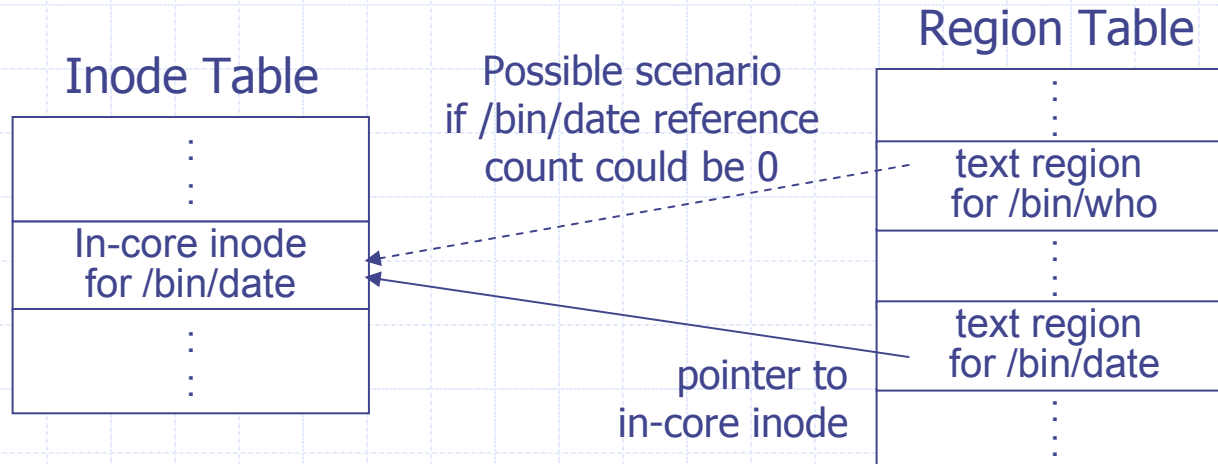
algorithm allocreg

```
{
    :
    assign region inode pointers
    if(inode pointer is not null)
        increment inode reference count;
    :
}
```

- ◆ the inode reference count is at least 1
- ◆ if a process unlinks the file, its contents remain intact
- ◆ kernel no longer needs the file after loading it into memory
- ◆ but it needs the pointer to the in-core inode in the region table
- ◆ If inode reference count is 0, could reallocate to another file
- ◆ if user were to exec the new file, would find the text region of the old file by mistake
- ◆ avoid this problem by incrementing the inode reference count

Increment inode reference count in allocreg

- ◆ executes “bin/date”, the kernel allocates a region table entry
- ◆ if not increment the inode reference count, its reference count would be 0
- ◆ suppose another process execs the file “/bin/who”, and kernel allocates the in-core inode previously used for “bin/date”
- ◆ kernel would search the region table for “bin/who”, but find the inode for “bin/date”
- ◆ would execute wrong program



Sticky-Bit

- ◆ Decrease the start-up time using *stick-bit*
 - system administrator can set the stick-bit with *chmod* system call
 - does not release the memory allocated for text during *exit*, *exec*
 - leaves text region with inode reference count 1
- ◆ remove
 - a process opens for writing
 - a process changes permission modes of the file (*chmod*)
 - a process *unlinks* the file
 - a process unmounts the file system
 - the kernel runs out of space on the swap device

The User ID of Process

- ◆ two user ids of process
 - real user ID
 - ◆ identifies the user who is responsible for the running process
 - effective user ID
 - ◆ assign ownership of newly created files
 - ◆ to check file access permissions
 - ◆ to check permission to send signals via *kill* system call
 - ◆ can change when it *execs* a *setuid* program or invokes *setuid* system call explicitly
 - ◆ call effective user ID in process table *saved user ID*
- inherits its real and effective user IDs from its parent during fork, and maintains their values across exec system calls.

The User ID of Process(Cont.)

◆ *setuid* program

- an executable file that has the *setuid* bit set in its permission mode field
- when *execs* a *setuid* program, the kernel sets the effective user ID fields in the process table and u area to the owner ID of the file.

◆ *setuid* system call

- *setuid*(uid)
 - ◆ uid : the new user ID
 - ◆ result depends on the value of current effective user.
 - Super user : resets the real and effective user ID and u area to *uid*
 - Not Super user, and (*uid* == real user ID or *uid* == saved user ID) : resets the effective user ID in u area to *uid*
 - Otherwise, returns an error

Example of Execution of Setuid Program

```
#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("manuray", 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_RDONLY);
    printf("fdmjb %d fdmaury %d\n", fdmjb, fdmaury);

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

owner of program : maury(UID 8319)
mjb : UID 5088
setuid bit on
all users have permissions to execute
User maury owns file "maury"(Read-Only)
User mjb owns file "mjb"(Read-Only)

Example of Execution of Setuid Program(Cont.)

◆ user mjb's output

```
uid 5088 euid 8319
fdmjb -l fdmaury 3
after setuid(5088) : uid 5088 euid 5088
fdmjb 4 fdmaury -l
after setuid(8319) : uid 5088 euid 8319
```

◆ user maury's output

```
uid 8319 euid 8319
fdmjb -l fdmaury 3
after setuid(5088) : uid 8319 euid 8319
fdmjb -l fdmaury 4
after setuid(8319) : uid 8319 euid 8319
```

Changing the Size of a Process

◆ *brk* system call

- process can increase or decrease the size of its data region
- `brk(endds);`
 - ◆ `endds` : the value of highest virtual address of the data region of the process(called its break value)
- `oldendds = sbrk(increment)`
 - ◆ `oldendds` : break value before the call
 - ◆ `increment` : changes current break value by the specified number of bytes
 - ◆ a C Library routine calls `brk`

Algorithm for Brk

```
algorithm brk
input : new break value
output : old break value
{
    lock process data regions;
    if (region size increasing)
        if (new region size is illegal)
        {
            unlock data regions;
            return(error);
        }
    change region size(algorithm growreg);
    zero out address in new data space;
    unlock process data region;
}
```

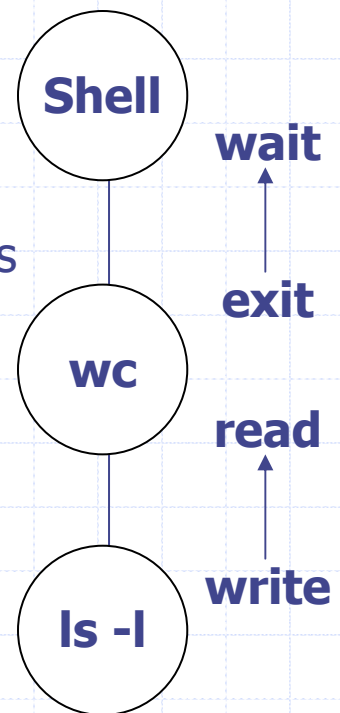
The Shell

1. reads a command line from its standard input and interprets it
2. built in command
 - *cd, for, while ...*
 - executes command internally without creating process
3. simple command line(program and parameters)
 - *who, grep -n include *.c, ls -l*
 - *forks* and creates a child process
 - *execs* the program user specified
 - shell *waits* until the child process *exits*
4. run a process asynchronously(in the background)
 - *nohup -mm bigdocument &*
 - sets an internal variable *amper*
 - shell does not execute *wait*
5. redirect standard output to a file
 - *nohup -mm bigdocument &*
 - the child creates the output file
 - closes its standard output
 - dup its file descriptor to standard output
 - redirects standard input and standard error in similar way

The Shell(Cont.)

6. pipe

- `ls -l | wc`
- parent process *forks* and creates a child process
- child creates a *pipe*
- child process *forks* and creates a grandchild process
- grandchild process handles the first command(*/s*)
 - ◆ close its standard output file descriptor
 - ◆ dups the pipe write descriptor
 - ◆ close original pipe write descriptor
- child process handles the second command(*wc*)
 - ◆ close its standard input file descriptor
 - ◆ dups the pipe read descriptor
 - ◆ close original pipe read descriptor
- output of grandchild(*/s*) goes to the input of child(*wc*)
- parent shell waits its child(*wc*) to exit



Main Loop of the Shell

```
/* read command line until "end of file" */
while(read(stdin,buffer,numchars))
{
    /* 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 directed */
        }
    }
}
```

Main Loop of the Shell(Cont.)

```

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 */
        /* child process does command */
        execlp(command1, command1, 0 );
    }
    /* 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);
}

```

Bootstrap

- ◆ to initialize system from inactive state
- ◆ get a copy of the operating system into machine memory and start executing
- ◆ on UNIX system
 - read the boot block(block 0) of a disk, and loads it into memory
 - the program contained in the boot block loads the kernel from the file system
 - boot program transfers control to the start address of the kernel, and the kernel starts running

Algorithm for Booting the System

```
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 */
        allocates regions;
        attach regions to init address space;
        grow region to accommodate the code about to copy in;
        copy code from kernel space to init user space to exec init;
        change mode: return from kernel mode to user mode;
        /* init nevr 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
        */
    }
}
```


Algorithm for Booting the System (Cont.)

```
/* proc 0 continues here */  
fork kernel processes;  
/* process 0 invokes the swapper to manage the allocation of  
* process address space to main memory and the 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;  
}
```

Init Process

◆ init process

- a process dispatcher, spawning processes that allows users to log in to the system, among others
- reads the file “/etc/inittab” for instructions about which

Format : identifier, state, process specification

Fields separated by colons.

Comment at end of line preceded by `#`

```
co::respawn::/etc/getty console console      # Console in machine room
46:2:respawn:/etc/getty -t 60 tty46 4800H    # comments here
```

Algorithm for Init

```

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)
        {
            exec("process specified in buffer");
            exit();
        }
        /* init process does not wait */
        /* loop back to while */
    }
}

```

Algorithm for Init(Cont.)

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

Processes in UNIX System

◆ User Process

- most processes on typical system
- associated with users at a terminal

◆ Daemon Process

- not associated with any users
- do system-wide functions
- administration and control of networks, execution of time-dependent activities, line printer spooling, and so on
- run at user mode and make system calls to access system service like user process

◆ Kernel Process

- execute only in kernel mode
- process 0 spawns kernel process, such as page-reclaiming process *vhand*, and then becomes the *swapper* process
- extremely powerful, not flexible

Introduction

◆ On a time sharing system..

- The kernel allocates the CPU to a process for a period of time called a time slice or Time quantum
- Every active process has a scheduling priority
- The kernel recalculates the priority of the running process when it returns from kernel mode to user mode
- Some user processes also have a need to know about time
 - ◆ Ex) time, date (user command)
- The system keeps time with a hardware clock that interrupts the CPU at a fixed, hardware-dependent rate, typically between 50 and 100 times a second
- Each occurrence of a clock interrupt is called a clock tick.

8.1 PROCESS SCHEDULING

- ◆ **The scheduler on the UNIX system belongs to the general class of operating system schedulers known as round robin with multilevel feedback**
- ◆ **the kernel**
 - **allocates the CPU to a process for a time quantum**
 - **preempts a process that exceeds its time quantum**
 - **feeds it back into one of several priority queues.**
- ◆ **A process may need many iterations through the "feedback loop" before it finishes.**
- ◆ **When the kernel does a context switch and restores the context of a process, the process resumes execution from the point where it had been suspended**

8.1.1 Algorithm for Process Scheduling

◆ Algorithm `schedule_process`

input: none

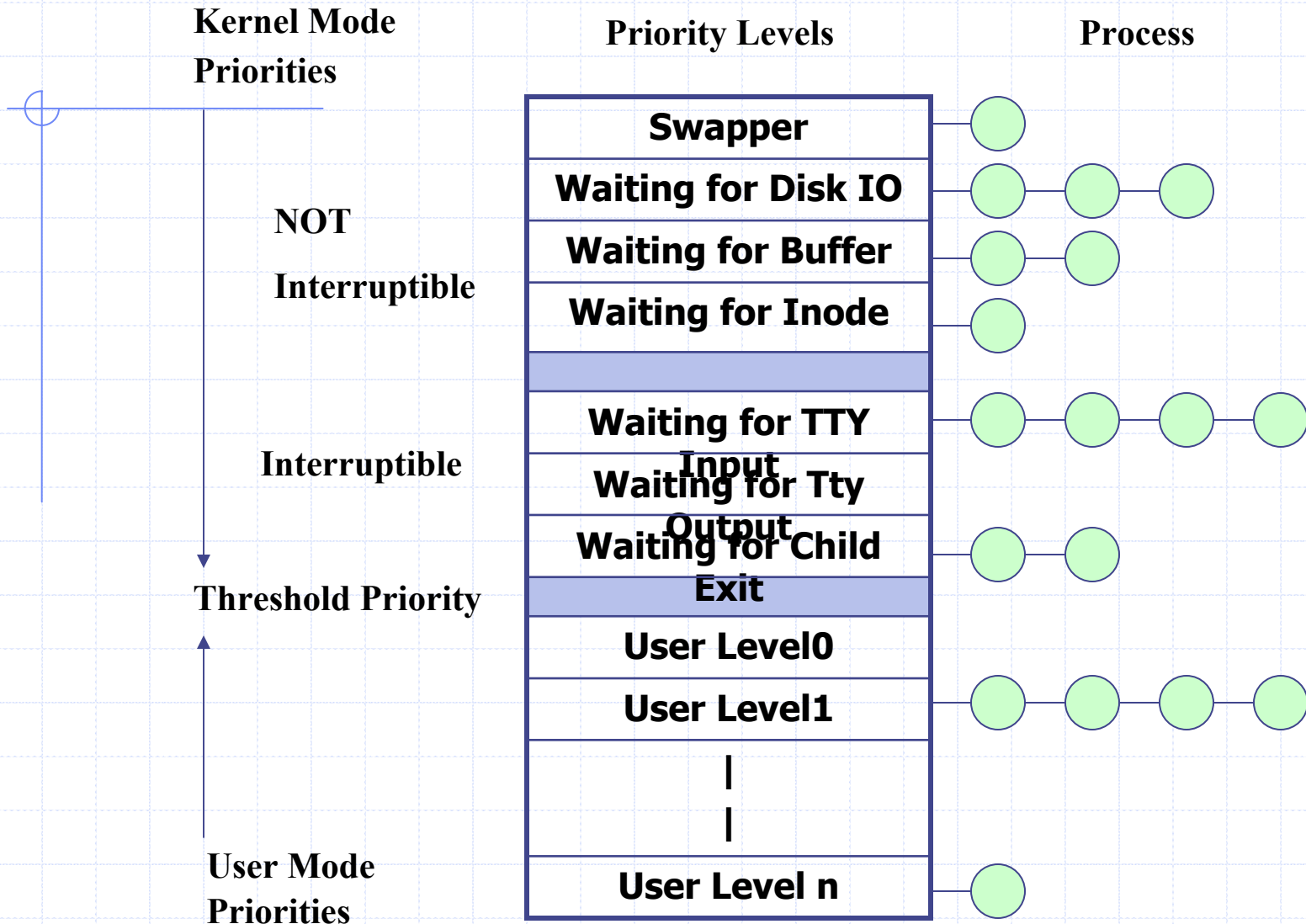
output: none

```
{  
    while (no process picked to execute)  
    {  
        for (every process on run queue)  
            pick highest priority process that is loaded in memory;  
        if (no process eligible to execute)  
            idle the machine;  
        /* interrupt takes machine out of idle state */  
    }  
    remove chosen process from run queue;  
    switch context to that of chosen process, resume its execution;  
}
```

8.1.2 Scheduling Parameters

- ◆ Each process table entry contains a priority field for process scheduling
 - The range of process priorities can be partitioned into two classes (user priorities and kernel priorities)
 - Kernel-level priorities are further subdivided :
 - ◆ (Not interruptible , Interruptible)
- ◆ Process with user-level priorities were preempted on their return from the kernel to user mode
- ◆ Processes with kernel-level priorities achieved them in the sleep algorithm.
- ◆ The kernel calculates the priority of a process in specific process states
 - The clock handler adjusts the priorities of all processes in user mode at 1 second intervals (on System V) and causes the kernel to go through the scheduling algorithm to prevent a process from monopolizing use of the CPU

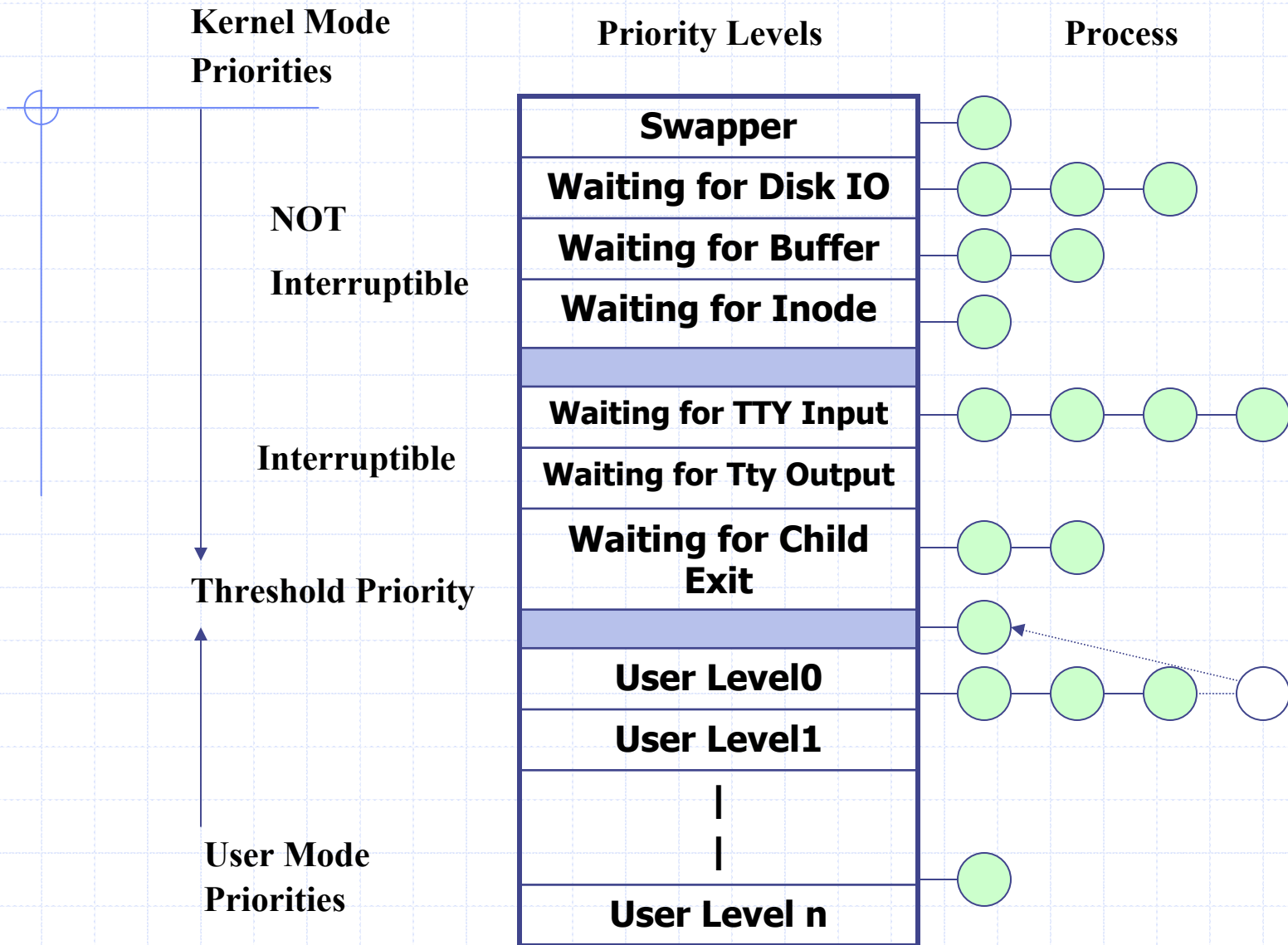
Range of Process Priorities



8.1.2 Scheduling Parameters(Cont.)

- ◆ The clock handler increments a field in the process table that records the recent CPU usage of the process
- ◆ Once a second, the clock handler also adjusts the recent CPU usage of each process according to a decay functions,
 $\text{decay}(\text{CPU}) = \text{CPU} / 2$ (on system V)
- ◆ When it recomputes recent CPU usage, the clock handler also recalculates the priority of every process in the “preempted but ready-to-run” state according to the formula
 - **$\text{Priority} = (\text{“recent CPU usage”} / 2) + (\text{base level user priority})$**
- ◆ Where “base level user priority” is the threshold priority between kernel and user mode described above.

Movement of a Process on Priority Queues



8.1.3 Example of Process Scheduling

- ◆ The processes are created simultaneously with initial priority 60
- ◆ The highest user-level priority is 60
- ◆ The clock interrupts the system 60 times a second
- ◆ The processes make no system calls
- ◆ No other processes are ready to run
 - $CPU = \text{decay}(CPU) = CPU/2$
 - $\text{Priority} = (CPU/2) + 60;$

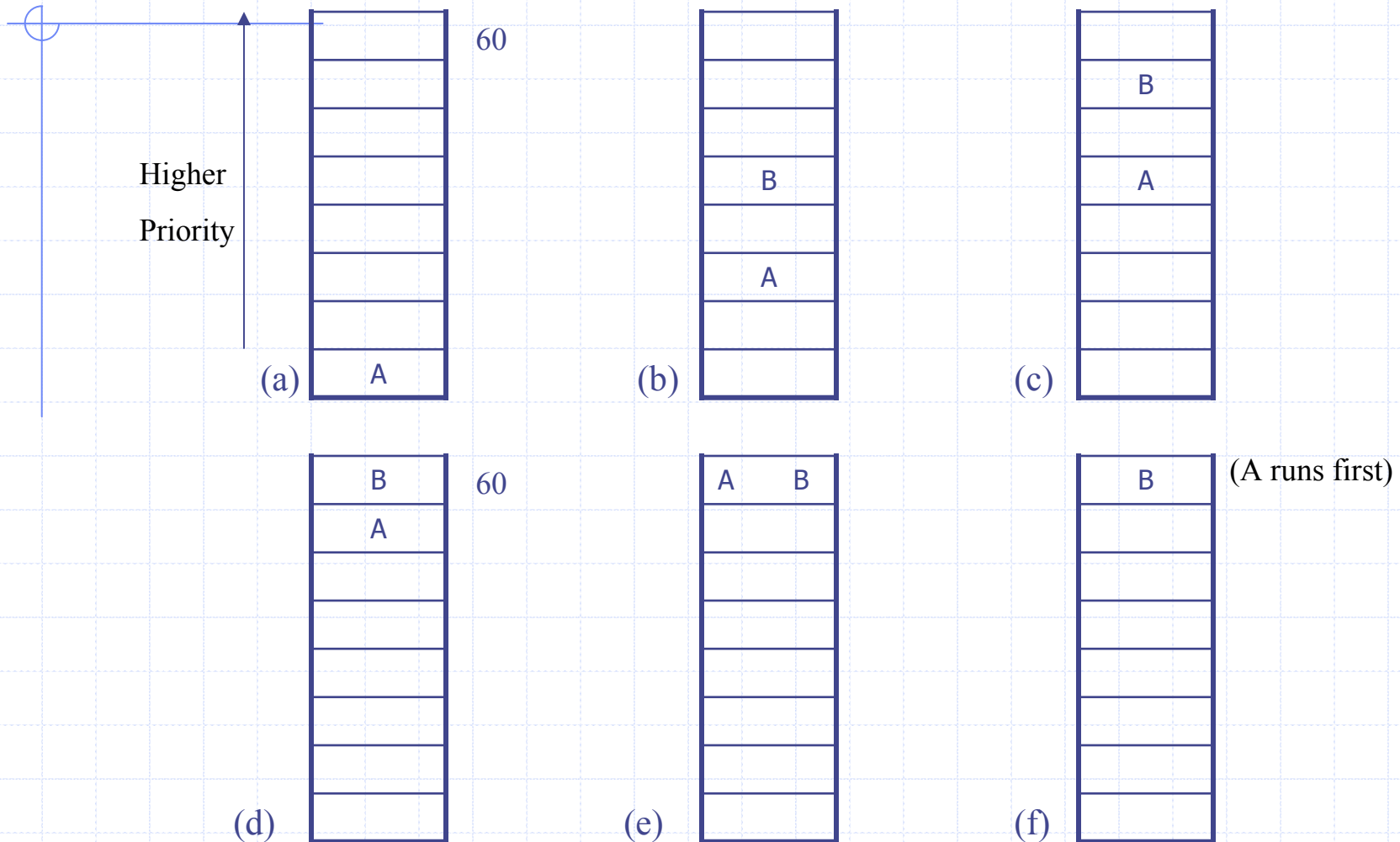
8.1.3 Example of Process Scheduling(Cont.)

Time	Proc A			Proc B			Proc C		
	Priority	Cpu	Count	Priority	Cpu	Count	Priority	Cpu	Count
0	60		0	60		0	60		0
			1						
			2						
1	75		60	60	0		60	0	
			30						
2	67	15		75	30		60	0	
								1	
								2	
3	63	7		67	15		75	60	
			8					30	
			9						
4	76		67	63	7		67	15	
			33						
5	68	16		76	67		63	7	
					33				

8.1.4 Controlling Process Priorities

- ◆ Processes can exercise crude control of their scheduling priority by using the nice system call: `nice(value)`
- ◆ Where value is added in the calculation of process priority:
 - **Priority**=(“recent CPU usage”/constant)+(base priority) + (nice value)
- ◆ The nice system call increments or decrements the nice field in the process table by the value of the parameter.
- ◆ Only the superuser can supply nice values that increase the process priority or that below a particular threshold.
- ◆ Processes inherit the nice value of their parent during the fork system call
- ◆ The nice system call works for the running process only
- ◆ A process cannot reset the nice value of another process
 - Practically, this means that if a system administrator wishes to lower the priority values of various processes because they consume too much time, there is no way to do so short of killing them outright.

Round Robin Scheduling and Process Priorities



8.1.5 Fair Share Scheduler

- ◆ The principle of the fair share scheduler is to divide the user community into a set of fair share groups, such that the members of each group are subject to the constraints of the regular process scheduler relative to other process in the group.
- ◆ The system allocates its CPU time proportionally to each group, regardless of how many processes are in the groups.
- ◆ Each process has a new field in its u area that points to a fair share CPU usage field, shared by all processes in the fair share group.
- ◆ Example.
 - Process A is in one group and processes B and C are in another
 - Assuming the kernel schedules process A first, and so on..

Example of Fair Share Scheduler – Three Processes, Two Groups

Time	Proc A			Proc B			Proc C		
	Priority	Cpu	Group	Priority	Cpu	Group	Priority	Cpu	Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
1	90	60	60	60	0	0	60	0	0
		30	30						
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
3	96	75	75	74	15	15	67	0	15
		37	37						
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
5	98	78	78	70	3	18	76	15	18
		39	39						

8.1.6 Real-Time Processing

- ◆ Real-time processing implies the capability
 - to provide immediate response to specific external events
 - to schedule particular processes to run within a specified time limit after occurrence of an event
- ◆ They cannot guarantee that the kernel can schedule a particular process within a fixed time limit.
- ◆ The impediment to the support of real-time processing is that the kernel is nonpreemptive.
- ◆ A true solution to the problem must allow real-time processes to exist dynamically (that is, not be hard-coded in the kernel), providing them with a mechanism to inform the kernel of their real-time constraints.
- ◆ No standard UNIX system has this capability today.