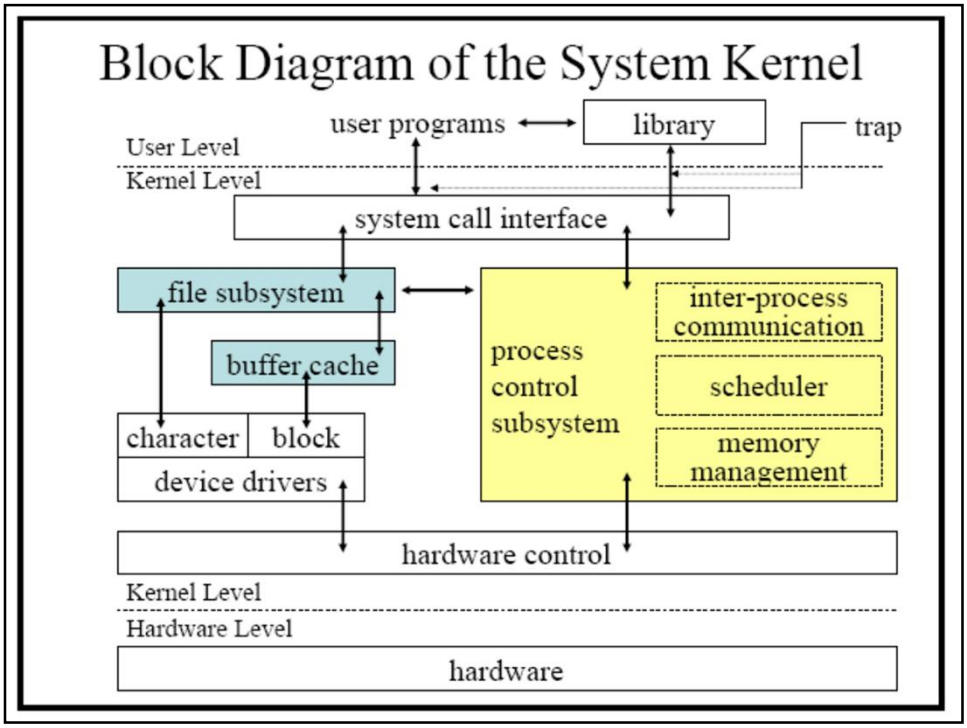
Process Structure

1



2

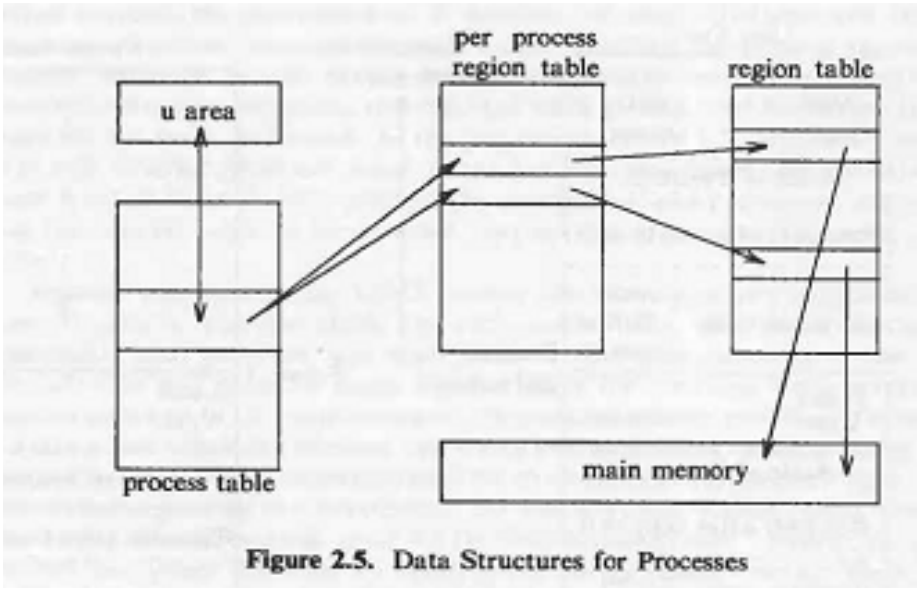
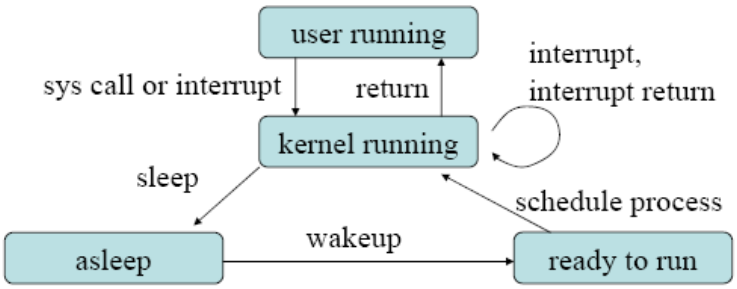


Figure 2.5. Data Structures for Processes

3

State Transitions

- Processes move continuously between the states according to well-defined rules.
- A *state transition* diagram is a directed graph whose *nodes* represent the states and whose *edges* represent the events.



4

Process States (1/2)

1. The process is executing in user mode.
2. The process is executing in kernel mode.
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.

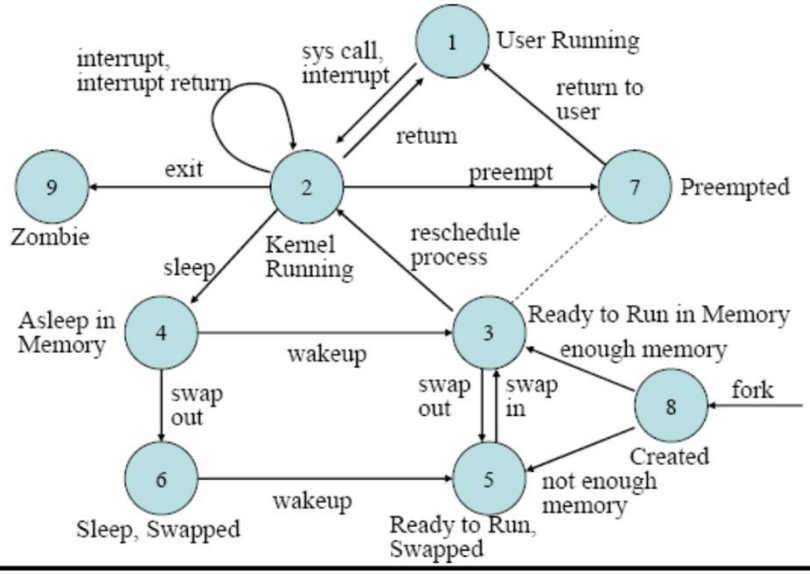
5

Process States (2/2)

7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. The state is the start state for all processes except process 0.
9. The process executed the *exit* system call and is in the *zombie* state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

6

Process State Transition Diagram



7

State Transition (1/3)

- The process enters the state model in the "created" state when the parent process executes *fork* system call and eventually moves into a state where it is ready to run (3 or 5).
- The process scheduler will eventually pick the process to execute, and the process enters the state "kernel running," where it completes its part of the *fork* system call.
- When the process completes the system call, it may move to the state "user running."
- After a period of time, the system clock may interrupt the processor, and the process enters state "kernel running" again.
- When the clock interrupt handler finishes servicing the clock interrupt, the kernel may decide to schedule another process to execute, so the first process enters state "preempted."

8

State Transition (2/3)

- The state "preempted" is really the same as the state "ready to run in memory."
  - A process executing in kernel mode can be preempted only when it is about to return to user mode.
- Eventually, the scheduler will choose the process to execute, and it returns to the state "user running."
- When a process executes a system call, it enters the state "kernel running."
- Suppose the system call requires I/O from the disk.
- It enters the state "asleep in memory."
- When the I/O later completes, the hardware interrupts the CPU, and the interrupt handler awakens the process, causing it to enter the state "ready to run in memory."

9

State Transition (3/3)

- When a process is evicted from main memory, it enters the state "ready to run swapped."
- Eventually, the swapper chooses the process as the most suitable to swap into main memory, and the process reenters the state "ready to run in memory."
- The scheduler will eventually choose to run the process, and it enters the state "kernel running."
- When a process completes, it invokes the *exit* system call, thus entering the states "kernel running" and, finally, the "zombie" state.
- No process can preempt another process executing in the kernel .

10

Process Table and *U Area*

- Two kernel data structures describe the state of a process; the process table entry and *u area*.
- The process table contains fields that must be always be accessible to the kernel, but the *u area* contains fields that need to be accessible only to the running process; therefore, the kernel allocates space for the *u area* only when creating a process.

11

Fields in the Process Table (1/2)

- The *state* field identifies the process state.
- The process table entry contains fields that allow the kernel to locate the process and its *u area* in main memory or in secondary storage.
  - the kernel uses the information to do a *context switch* to the process when the process moves from state "ready to run in memory" to the state "kernel running", from the state "preempted" to the state "user running.", or swapping the process in/out.
- Several user identifiers (UIDs) determine various process privileges.
  - UID fields delineate the set of processes that can send signals to each other.

12

Fields in the Process Table (2/2)

- Process identifier (PID) specify the relationship of processes to each other.
  - These ID fields are set up when the process enters the state "created."
- The process table entry contains an event descriptor when the process is in the *sleep* state.
- Scheduling parameters allow the kernel to determine the order in which processes move to the state "kernel running" and "user running."
- A signal field enumerates the signals sent to a process but not yet handled.
- Various timers give process execution time and kernel resource utilization, used for process accounting and for the calculation of process scheduling priority.

13

Fields in *U Area* (1/2)

- A pointer to the process table identifies the entry that corresponds to the *u area*.
- The real and effective UIDs determine various privileges allowed the process, such as file access rights.
- Timer fields record the time the process (and its descendents) spent executing in user mode and in kernel mode.
- An array indicates how the process wishes to react to signals.
- The control terminal field identifies the "login terminal" associated with the process, if one exists.
- An error field records errors encountered during a system call.

14

Fields in *U Area* (2/2)

- A return value field contains the result of system calls.
- I/O parameters describes the amount of data to transfer, the address of the source (or target) data array in user space, etc.
- The current directory and current root describe the file system environment of the process.
- The user file descriptor table records the files the process has *open*.
- Limit fields restrict the size of a process and the size of a file it can *write*.
- A permission modes field masks mode setting on files the process creates.

15

Layout of System Memory

- A process on the UNIX system consists of three logical sections: text, data, and stack.
- The text section contains the set of instructions the machine executes for the process.
- The compiler generates addresses for a *virtual address space* with a given address range, and the machine's MMU translates the virtual addresses into address locations in physical memory.
- The subsystems of the kernel and the hardware that cooperate to translate virtual to physical addresses comprise the *memory management* subsystem.

16

Regions (1/2)

- The System V kernel divides the virtual address space of a process into logical *regions*.
- A region is a contiguous area of the virtual address space of a process that can be treated as a distinct object to be shared or protected.
- Thus text, data, and stack usually form separate regions of a process.
- Each process contains a private *per process region table*, called a *pregion* for short.
  - for simplicity, assume that pregion entries are part of the process table entry.

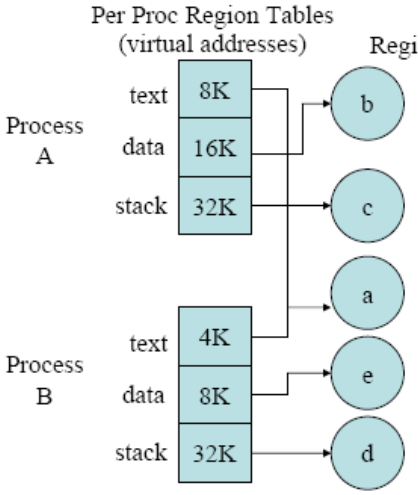
17

Regions (2/2)

- Each pregion entry points to a region table entry and contains the starting virtual address of the region.
- It also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute.
- The pregion and the region structures are analogous to the user file descriptor table and the inode structure in the file system.
  - several processes can share part of their address space via a region, much as they can share access to a file via an inode.
  - each process accesses the region via a private pregion entry, much as it accesses the inode via private entries in its user file descriptor table and the kernel file table.

18

Process and Regions (example)



- two processes, A and B share text region 'a' at virtual addresses 8K and 4K, respectively.
- if process A reads memory location 8K and process B reads memory location 4K, they read the identical memory location in region 'a'.
- the data regions and stack regions of the two processes are private.

19

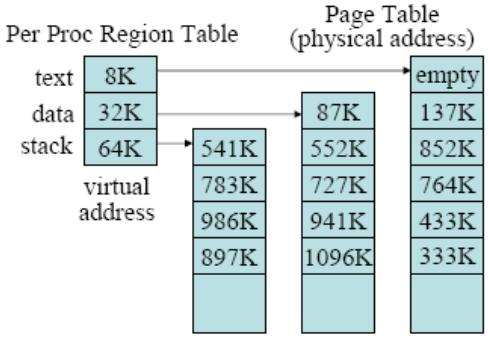
Pages and Page Tables (1/3)

- In a memory management architecture based on *pages*, the memory management hardware divides physical memory into a set of equal-sized blocks called pages.
- Typical page sizes range from 512 bytes to 4K bytes and are defined by the hardware.
- Every memory location can be addressed by a (page number, byte offset in page) pair.
  - if a machine has 2^{32} bytes of physical memory and a page size of 1K bytes, it has 2^{22} pages of physical memory; every 32-bit address can be treated as a pair consisting of a 22-bit page number and a 10-bit offset into the page

20

Pages and Page Tables (2/3)

- The region table entry contains a pointer to a table of physical page numbers called a *page table*.
  - page table entries may also contain machine-dependent information such as permission bits to allow reading or writing of the page.



- assume that the size of a page is 1K bytes, and suppose the process wants to access virtual address 34,432.
- the pregion entries show that the virtual address is in the data region starting at 32K.
- address 34,432 is at byte offset 1,664 in the region.
- the address is contained at byte offset 640 in page 1 (552K).

21

Pages and Page Tables (3/3)

- Let us use the following simple memory model.
- Memory is organized in pages of 1K bytes.
- The system contains a set of memory management register triples:
  - the first register contains the address of a page table in physical memory.
  - the second register contains the first virtual address mapped via the triple.
  - the third register contains control information such as the number of pages in the page table and page access permissions (read-only, read-write).
- When the kernel prepares a process for execution, it loads the set of memory management triples with the corresponding data stored in the pregion entries.
- If a process addresses memory locations outside its virtual address space, the hardware causes an exception condition.

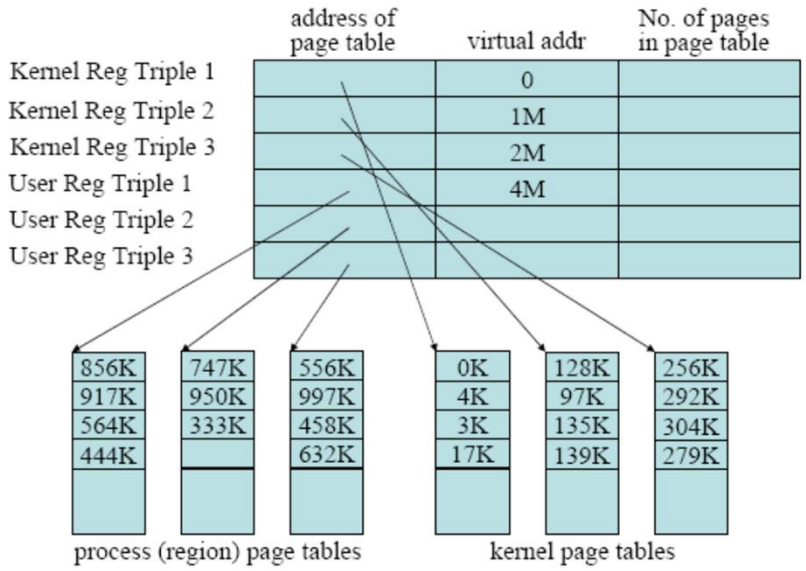
22

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 process share it.
- The kernel page tables are analogous to the page tables associated with a process, and the mechanisms used to map kernel virtual addresses are similar to those used for user addresses.
- 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.

23

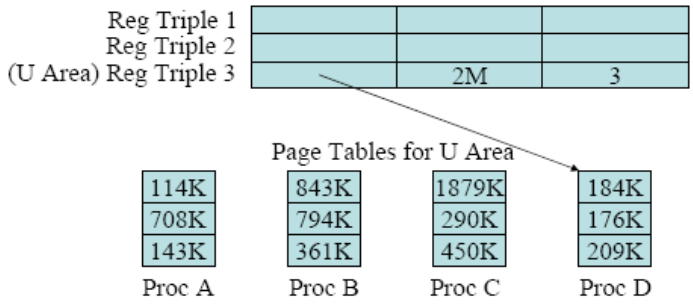
Layout of the Kernel (example)



24

The U Area

- Every process has a private *u area*, yet the kernel accesses it as if there were only one *u area* in the system, that of the running process.
- The kernel changes its virtual address translation map according to the executing process to access the correct *u area*.



25

The Context of a Process (1/6)

- Formally, the context of a process is the union of its *user-level context*, *register context*, and *system-level context*.
- The user-level context consists of the process text, data, user stack and shared memory that occupy the virtual address space of the process.
- The register context consists of the following components:
  - The program counter specifies the address of the next instruction the CPU will execute.
  - The processor status register (PS) specifies the hardware status of the machine as it relates to the process.
  - The stack pointer contains the current address of the next entry in the kernel or user stack, determined by the mode of execution.
  - The general-purpose registers contain data generated by the process during its execution.

26

The Context of a Process (2/6)

- The system-level context has a "static part" and a "dynamic part."
  - a process has one static part throughout its lifetime, but it can have a variable number of dynamic parts.
  - the dynamic part should be viewed as a stack of *context layers*.
- The system-level context consists of the following components (static part):
  - the process table entry of a process defines the state of a process.
  - the *u area* of a process contains process control information.
  - pregion entries, region tables and page tables, define the mapping from virtual to physical addresses and therefore define the text, data, stack, and other regions of a process.

27

The Context of a Process (3/6)

- The system-level context (dynamic part):
  - The kernel stack contains the stack frames of kernel procedures as a process executes in kernel mode. Although all processes execute the identical kernel code, they have a private copy of the kernel stack that specifies their particular invocation of the kernel functions.
  - The dynamic part of the system-level context of a process consists of a set of layers, visualized as a last-in-first-out stack. Each *system-level context layer* contains the necessary information to recover the previous layer, including the register context of the previous level.

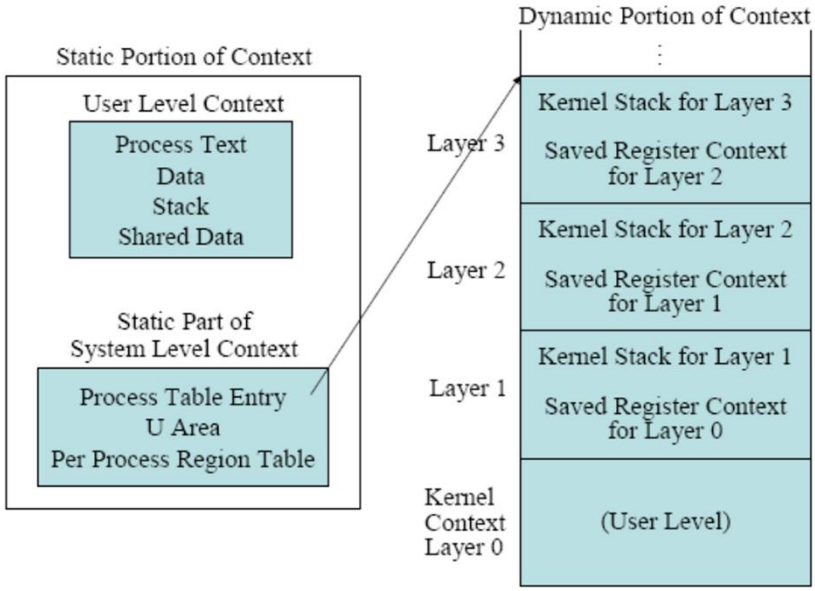
28

The Context of a Process (4/6)

- The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does context switch.
- It pops a context layer when the kernel returns from handling interrupt, when a process returns to user mode, or when a process does a context switch.
- The context switch thus entails a push and pop of a system-level context layer: the kernel pushes the context layer of the old process and pops the context layer of the new process.
- The process table entry stores the necessary information to recover the current context layer.

29

Components of the Context of a Process



30

The Context of a Process (5/6)

- 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.
  - if a machine supports different interrupt levels for software interrupts, terminals, disks, network, and the clock, it supports 5 interrupt levels, and hence, a process can contain at most 7 context layers: 1 for each interrupt level, 1 for system calls, and 1 for user-level.

31

The Context of a Process (6/6)

- Although the kernel always executes in the context of some process, the logical function that it executes does not necessarily pertain to that process.
- If a disk drive interrupts the machine because it has returned data, it interrupts the running process and the kernel executes the interrupt handler in a new system-level context layer of the executing process, even though the data belongs to another process.

32

Interrupts and Exceptions (1/3)

- The system is responsible for handling interrupts, whether they result from hardware, from a programmed interrupt ("software interrupts"), or from exceptions.
- If a CPU is executing at a lower processor execution level than the level of the interrupt, it accepts the interrupt and raises the processor execution level, so that no other interrupts of that level can happen while it handles the current interrupt

33

Interrupts and Exceptions (2/3)

- The kernel handles the interrupt with the following sequence of operations:

1. It saves the current register context of the executing process and creates (pushes) a new context layer.
2. It determines the "source" or cause of the interrupt, identifying the type of interrupt and the unit number of the interrupt, if applicable. When the system receives an interrupt, it gets a number from the machine that it uses as an offset into a table, commonly called an *interrupt vector*.

34

Interrupts and Exceptions (3/3)

3. The kernel invokes the interrupt handler. The kernel stack for the new context layer is logically distinct from the kernel stack of the previous context layer. Some implementations use the kernel stack of the executing process to store the interrupt handler stack frames.
4. The interrupt handler completes its work and returns. The kernel executes a machine-specific sequence of instructions that restores the register context and kernel stack of the previous context layer as they existed at the time of the interrupt and resumes execution of the stored context layer.

35

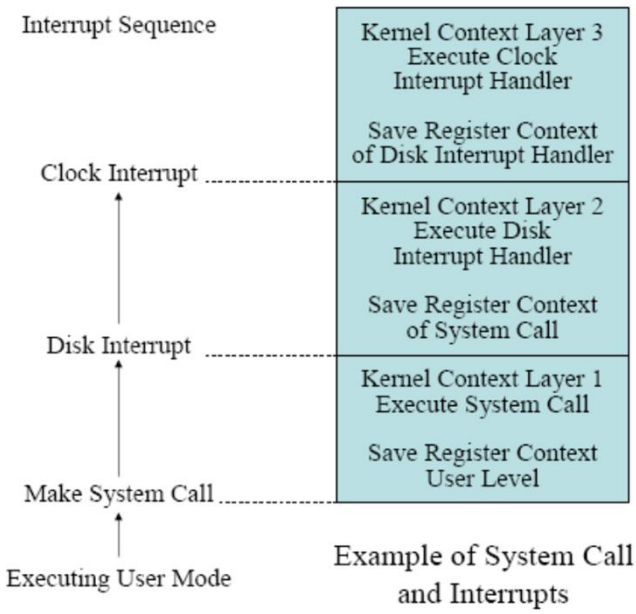
Interrupt Number	Interrupt Handler
0	clockintr
1	diskintr
2	ttyintr
3	netintr
4	softintr
5	otherintr

Sample Interrupt Vector

```
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 for Handling Interrupts

36



Example of System Call and Interrupts

37

System Call Interfaces

- The C compiler uses a predefined library of functions that have the names of the system calls.
- The library functions typically invokes an instruction referred to as an *operating system trap* that changes the process execution mode to kernel mode and causes the kernel to start executing code for system calls; the system call interface is a special case of an interrupt handler.
- The library functions pass the kernel a unique number per system call, e.g., in a particular register, and the kernel determines the specific system call the user is invoking.

38

```
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 numbers;
      determine number of parameters to system call;
      copy parameters from user address space to u area;
      save current context for abortive return; (described later)
      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 register 0, 1 in user saved register context to return values
                  from system call;
}
```

Algorithm for System Calls

39

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

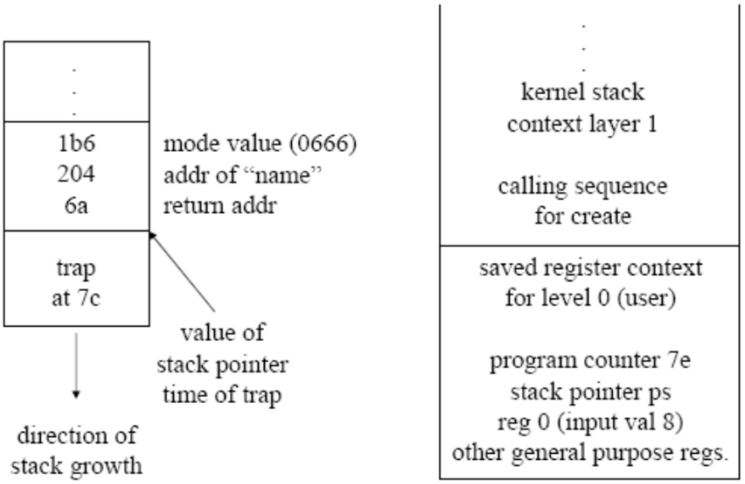
```
# code for main
58:     mov     &0x1b6,(%sp)     # move 0666 onto stack
5e:     mov     &0x204, –(%sp)   # move stack ptr and move variable
                                 # "name" onto stack
64:     jsr     0x7a             # call C library for creat

# 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 reg0 to location 20e (errno)
142:    movq    &–0x1,%d0        # move constant –1 into data register 0
144:    mova    %d0,%a0
146:    rts                      # return from subroutine
```

40

Stack Configuration for Creat System Call



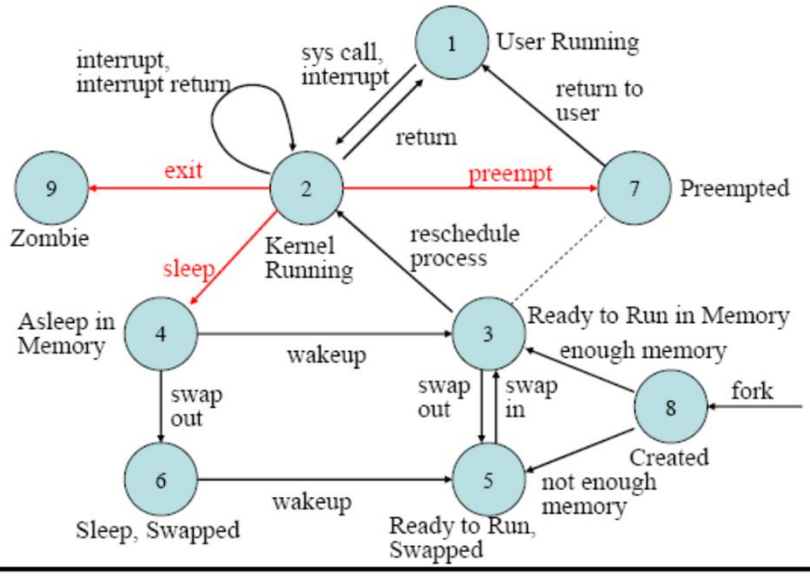
41

Context switch (1/5)

- The kernel permits a context switch under four circumstances:
  - when a process puts itself to sleep
  - when it *exit*s
  - when it returns from a system call to user mode but is not the most eligible process to run
  - when it returns to user mode after the kernel completes handling in interrupt but is not the most eligible process to run
- The kernel makes sure that the state of its data structures is consistent before it does a context switch.
  - e.g., if the kernel allocates a buffer and goes to sleep waiting for I/O, it keeps the buffer locked.

42

Process State Transition Diagram



43

Context Switch (2/5)

- The procedure for a context switch is similar to that 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.

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.
4. Restore its context

Steps for a Context Switch

44

Context Switch (3/5)

- The code that implements the context switch is usually the most difficult to understand in the operating system, because function calls gives the appearance of not returning on some occasions and materializing from nowhere on others.
- This is because the kernel saves the process context at one point in the code but proceeds to execute the context switch and scheduling algorithms in the context of the "old" process.
- When it later restores the context of the process, it resumes execution according to the previously saved context.
- To differentiate between the case where the kernel resumes the context of a new process and the case where it continues to execute in the old context after having saved it, the return value of critical functions may vary.

45

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

Pseudo-Code for Context Switch

46

Context Switch (4/5)

- 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 *save_context*) and the value 0, to be used later as the return value in register 0 from *save_context*.
- The kernel continues to execute in the context of the old process (A), picking another process (B) to run and calling *resume_context* to restore the new context (of B).
- After the new context is restored, the system is executing B; the old process (A) is no longer executing but leaves its saved context behind.
  - hence, the comment in the figure "never gets here!"

47

Context Switch (5/5)

- Later, the kernel will again pick process A to run when another process does a context switch.
- When process A's context is restored, the kernel will set the program counter to the value process A had previously saved in *save_context*, and it will also place the value 0, saved for return value, into register 0.
- The kernel resumes execution of process A inside *save_context* even though it had executed the code up to the call to resume_context before the context switch.
- Finally, process A returns from save_context with the value 0 and resumes execution after the comment line "returning process executes from here."

48

Copying Data between System and User Address Space

- Many system calls move data between kernel and user space such as copying system call parameters from user to kernel space or when copying data from I/O buffers in the *read* system call.
- Many machines allow the kernel to reference addresses in the user space directly. The kernel must ascertain that the address being read or written is accessible as if it had been executing in user mode.
- Therefore, copying data between kernel and user space is an expensive proposition, requiring more than one instruction.

49

Process Address Space Manipulation

- Various system calls manipulate the virtual address space of a process, doing so according to well defined operations on regions.
- The region table entry contains the following information:
  - a pointer to the inode of the file whose contents were originally loaded into the region.
  - the region type (text, data, stack, or shared memory)
  - the size of the region
  - the location of the region in physical memory
  - the status of a region: combination of 1)locked, 2)in demand, 3)in the process of being loaded into memory, or 4)valid.
  - the reference count (number of processes that reference it)

50

Process Address Space Manipulation

- The operations that manipulate regions are
  - lock / unlock a region
  - allocate / free a region
  - attach a region to the memory space of a process / detach a region from the memory space
  - change the size of a region
  - load a region from a file into the memory space of a process
  - duplicate the contents of a region
- E.g., the *exec* system call detaches old regions, frees them if they were not shared, allocates new regions, attaches them, and loads them with the contents of the file.

51

Locking/Unlocking a Region

- The kernel has operations to lock and unlock a region, independent of the operations to allocate and free a region, just as the file system has lock-unlock and allocate-release operations for inodes.
  - *iget, iput* vs. *ialloc, ifree*

52

Allocating a Region

- The kernel allocates a new region (*allocreg*) during *fork*, *exec*, and *shmget* system calls.
- The kernel contains a region table whose entries appear either on a free linked list or on an active linked list.
- When it allocates a region table entry, the kernel removes the first available entry from the free list, places it on the active list, locks the region, and marks its type (shared or private).
- Every process is associated with an executable file as a result of a prior *exec* call, and allocreg sets the inode field in the region table entry to point to the inode of the executable file.
- The inode identifies the region to the kernel so that other processes can share the region if desired.
- The kernel increments the inode reference count to prevent other processes from removing its contents when unlinking it.

53

```
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);
}
```

Algorithm for Allocating a Region

54

Attaching a Region to a Process (1/2)

- The kernel attaches a region during the *fork*, *exec*, and *shmat* system calls to connect it to the address space of a process (*attachreg*).
- The kernel allocates a free pregion entry, sets its type field to text, data, shared memory, or stack, and records the virtual address where the region will exist in the process address space.
  - the kernel checks address limit, address overlap, etc.
- If it is legal to attach a region, the kernel increments the size field in the process table entry according to the region size, and increments the region reference count.

55

Attaching a Region to a Process (2/2)

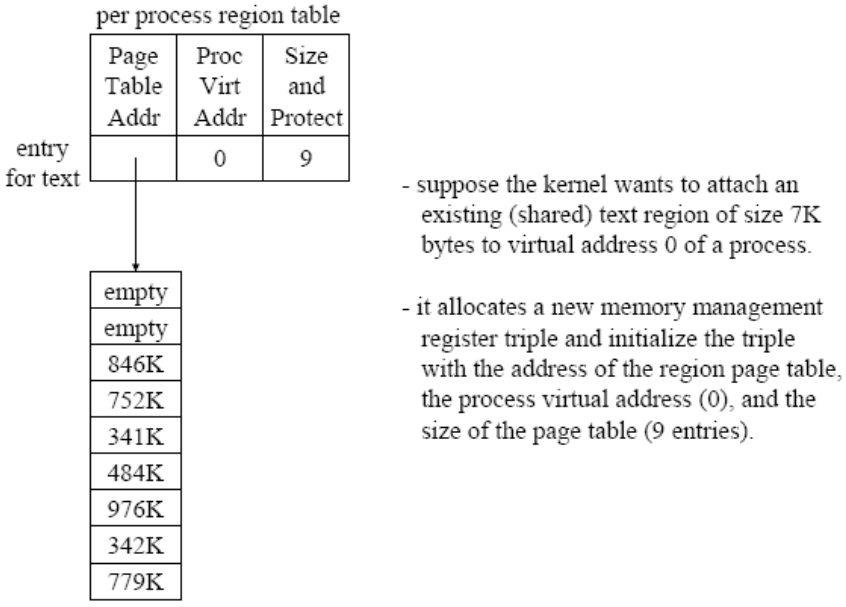
- *Attachreg* then initializes a new set of memory management register triples for the process: If the region is not already attached to another process, the kernel allocates page tables for it in a subsequent call to *growreg*; otherwise, it uses the existing page tables.
- Finally, attachreg returns a pointer to the pregion entry for the newly attached region.

56

```
algorithm attachreg          /* attach a region to a process */
input: (1) pointer to (locked) region being attached
       (2) process to which region is 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);
}
```

Algorithm for Attachreg

57



- suppose the kernel wants to attach an existing (shared) text region of size 7K bytes to virtual address 0 of a process.

- it allocates a new memory management register triple and initialize the triple with the address of the region page table, the process virtual address (0), and the size of the page table (9 entries).

58

Changing the Size of a Region (1/2)

- A process may expand or contract its virtual address space with *sbrk* system call. The kernel invokes *growreg* to change the size of a region.
  - similarly, the stack of a process automatically expands according to the depth of nested procedure calls.
- The kernel never invokes *growreg* to increase the size of a shared region that is already attached to several processes.
- Two cases where the kernel uses *growreg* on an existing region are *sbrk* on the data region of a process and automatic growth of the user stack.

59

Changing the Size of a Region (2/2)

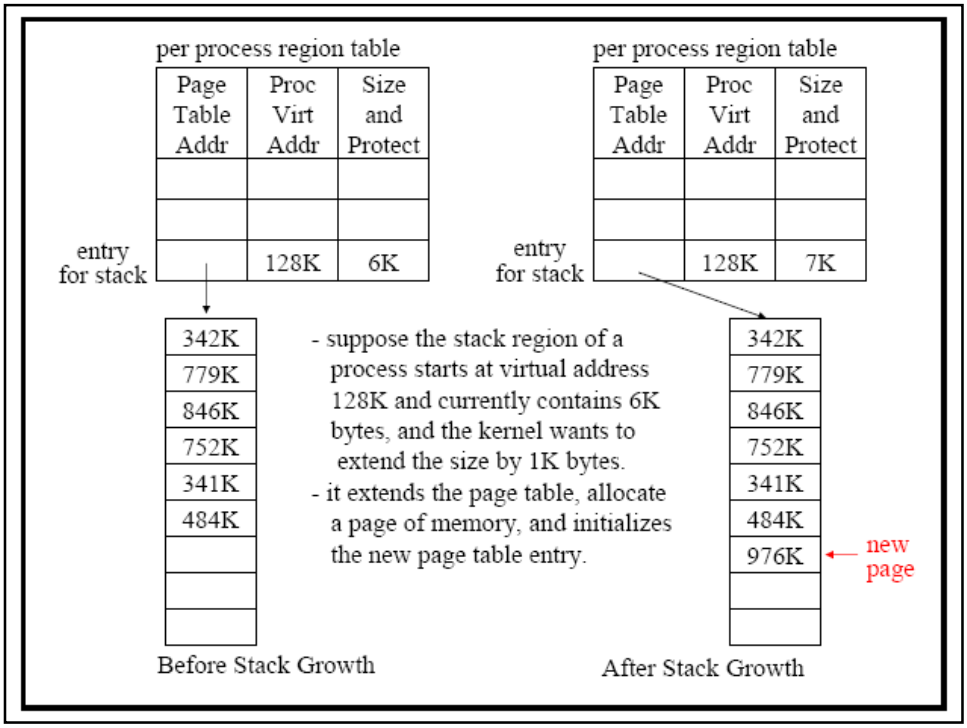
- The kernel allocates page tables (or expands existing page tables) to accommodate the larger region and allocates physical memory on systems that do not support demand paging.
- If the process contracts the region, the kernel simply releases memory assigned to the region.
- In both cases, it adjusts the process size and region size and initializes the pregion entry and memory management register triples to conform to the new mapping.

60

```
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;
}
```

Algorithm Growreg for Changing the Size of a Region

61



62

Loading a Region

- In a system that supports demand paging, the kernel can "map" a file into the process address space during *exec*, arranging to read individual physical pages later on demand.
- 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.

63

```
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 (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;
}
```

64

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 *iput*.
- The kernel releases physical resources associated with the region, such as page tables and memory pages.

65

```
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 (iput);
      free physical memory still associated with region;
      free auxiliary table s associated with region;
      clear region fields;
      place region on region free list;
      unlock region;
}
```

66

Detaching a Region from a Process

- The kernel detaches regions in the *exec*, *exit*, and *shmdt* system calls.
- It updates the pregion entry and severs the connection to physical memory by invalidating the associated memory management register triple.

67

```
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 (freereg);
        else {
                free inode lock, if applicable (inode associated with region);
                free region lock;
        }
}
```

68

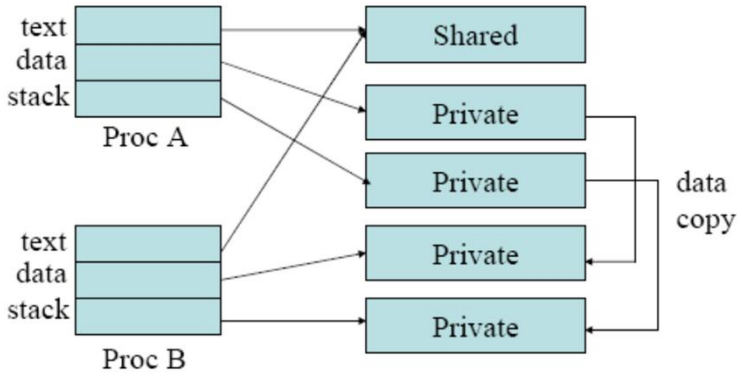
Duplicating a Region

- The *fork* system call requires that the kernel duplicate the regions of a process.
- If a region is shared, however, the kernel need not physically copy the region; instead, it increments the region reference count, allowing the parent and child process to share the region.
- If the region is not shared and the kernel must physically copy the region, it allocates a new region table entry, page table, and physical memory for the region.

69

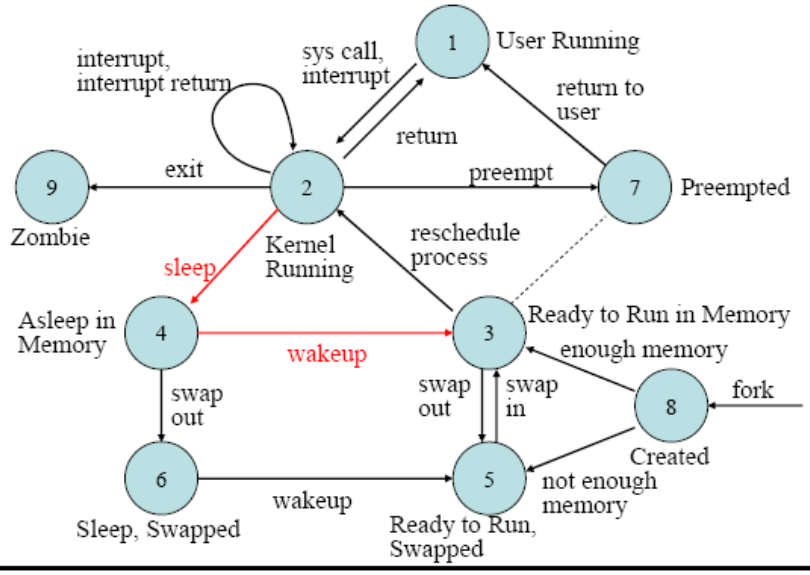
Duplicating a Region

- Process A *fork*ed process B and duplicated its regions.
  - the text region is shared while the data and stack regions are duplicated. (physical copy of data is not always necessary.)



70

Process State Transition Diagram

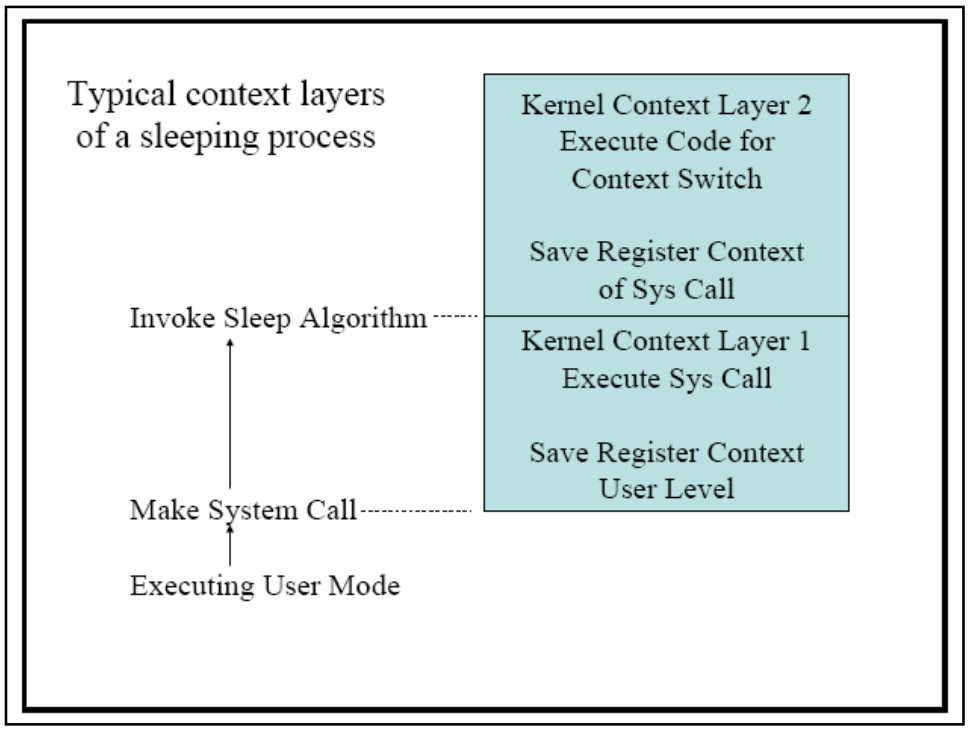


71

Sleep

- The algorithm *sleep* changes the process state from "kernel running" to "asleep in memory," and the algorithm *wakeup* changes the process state from "asleep in memory" to "ready to run" in memory or swapped.
- The process enters the kernel (context layer 1) when it executes an operating system trap and goes to sleep awaiting a resource.
- When a process goes to sleep, it does a context switch, pushing its current context layer and executing in kernel context layer 2.
- Processes also go to sleep when they incur page faults.

72



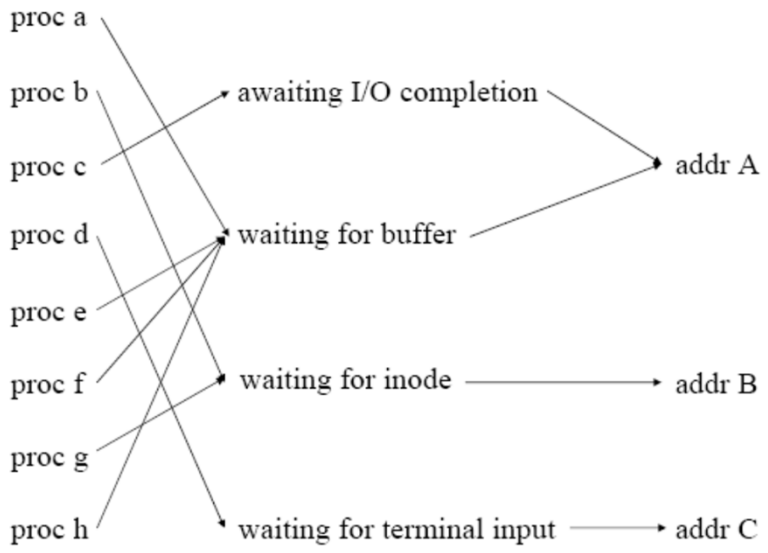
73

Sleep Events and Addresses

- Processes are said to sleep on an event, meaning that they are in the sleep state until the event occurs.
- Although the system uses the abstraction of sleeping on an event, the implementation maps the set of events into a set of (kernel) virtual addresses.
- The abstraction of the event does not distinguish how many processes are awaiting the event, nor does the implementation.
  - when an event occurs and a wakeup call is issued for processes that are sleeping on the event, they all wake up and move from a sleep state to read-to-run state.
  - several events may map into one address.

74

Processes Sleeping on Events



75

Sleep Algorithm

- The kernel first raises the processor execution level to block out all interrupts so that there can be no race conditions for manipulating queues and saving the old processor execution level.
- It marks the process state "asleep", saves the sleep address and priority in the process table, and puts it onto a hashed queue of sleeping processes.
- The process does a context switch and is safely asleep.
- When a sleeping process wakes up, the kernel later schedules it to run: the process returns from its context switch in *sleep*, restores the processor execution level, and returns.

76

```
algorithm sleep
input: (1) sleep address
       (2) priority
output: 1 if process awakened as a result of
          a signal that process catches
        0 otherwise;
{
   raise processor execution level;
   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 */
      reset processor priority level;
      return (0);
   }
   /* here, process sleep is
      interruptible by signals */
   if (no signal pending against proc) {
      do context switch;
      /* process resumes execution here */
      if (no signal pending against proc) {
         reset processor priority level;
         return (0);
      }
   }
   remove process from sleep hash queue,
         if still here;

   reset processor priority level;
   if (process sleep priority set to catch
            signals)
      return (1);
   do longjump algorithm;
}
```

77

Wakeup Algorithm

- The kernel executes *wakeup* either during the usual system calls or when handling an interrupt.
- The kernel raises the processor execution level in *wakeup* to block out interrupts. Then for every process sleeping on the input sleep address, it marks the process state field "read to run," removes the process from the linked list of sleeping processes, places it on a linked list of processes eligible for scheduling, and clears the sleep address field.
- If the awakened process is more eligible to run than the currently executing process, the kernel sets a scheduler flag so that it will go through the process scheduling algorithm when the process returns to user mode.
- *Wakeup* does not cause a process to be scheduled immediately.

78

```
algorithm wakeup            /* wakeup 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;
}
```

79

Sleep and Wakeup

- The kernel invokes *sleep* with a priority value, based on its knowledge that the sleep event is sure to occur or not.
- If the priority is above a threshold value, the process will not wake up prematurely on receipt of a signal.
- But if the priority value is below the threshold value, the process will awaken immediately on receipt of the signal.
  - if a signal is already set against a process when it enters *sleep*, the process does not go to sleep but responds to the signal as if the signal had arrived while it was asleep.

80

Sleep and Wakeup

- When a process is awakened as a result of a signal, the kernel may do a *longjump* to restore a previously save context if it has no way to complete the system call.
  - for instance, a terminal *read* call is interrupted by ctrl-C.
- The kernel saves the process context at the beginning of most system calls using *setjump* in anticipation of the need for a later *longjump*.
- If the kernel must clean up local data structure when a process is awakened by a signal, the kernel invokes *sleep* with a special priority parameter that suppresses execution of *longjump* and causes the *sleep* to return the value 1.

81