Process Control

1

Process System Calls and Relation to Other Algorithms

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					

- Almost all calls use *sleep* and *wakeup*.
- *exec* interacts with the file system algorithms

2

Process Creation

- syntax: pid = fork( );

1. The kernel allocates a slot in the process table for the new process.
2. It assigns a unique ID number to the child process.
3. It makes a logical copy of the context of the parent process. The kernel can sometimes increment a region reference count instead of copying the region to 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 the child to the parent process and a 0 value to the child process.

3

```
algorithm fork
input: none
output: to parent process, child PID number
        to child process, 0
{
   check for available kernel resource;
   get free proc table slot, unique PID
        number;
   check that user not running too many
        processes;
   mark child state "being created;"
   copy data from parent proc table slot to
        new child slot;
   increment counts on current directory
        inode and changed root
        (if applicable);
   increment open file counts in file table;
   make copy of parent context (u area, text,
        data, stack) in memory;
   push dummy system level context
        layer onto child system level
        context;
           dummy context contains data
           allowing child process to
           recognize itself, and start
           running from here when
           scheduled;
   if (executing proc is parent) {
      change child state to "ready to
           run;"
      return (child PID);
   } else {
      initialize u area timing fields;
      return (0);
   }
}
```

4

Process Creation

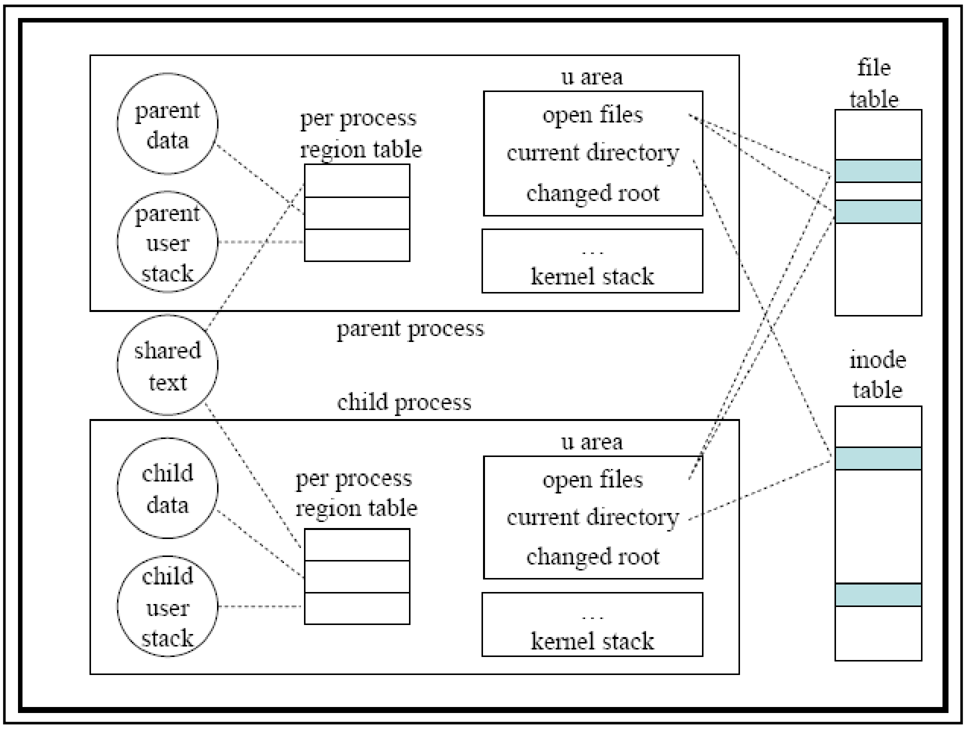
- The algorithm for *fork* varies slightly for demand paging and swapping systems; the ensuing discussion is based on traditional swapping systems.
- The kernel first ascertains that it has available resources to complete the *fork* successfully.
  - on a swapping system, it needs space either in memory or on disk to hold the child process;
  - on a paging system, it has to allocate memory for auxiliary tables such as page tables.

5

Process Creation

- The child process "inherits" various information held in the process table slot.
  - e.g., real/effective UID, process group, *nice* value, etc.
- Not only does the child process inherit access rights to open files, but it also shares access to the files with the parent process because both processes manipulate the same file table entries.

6



7

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

main(int argc, char *argv[])
{
   if (argc != 3)
      exit (1);
   if ((fdrd = open(argv[1], O_RDONLY))
                                        = = -1)
      exit(1);
   if ((fdwt = open(argv[2], O_CREAT))
                                        = = -1)
      exit(1);

   fork();

   rdwrt();
   exit (0);
}
```

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

Sample Program
(parent and child share
file access)

8

```c
#include <string.h>
char string[ ] = "hello world";
main( )
{
  int count, i, to_par[2], to_chil[2];
  char buf[256];
  pipe(to_par); pipe(to_chil);
  if (fork() == 0) {
    close(0);
    dup(to_chil[0]);
    close(1);
    dup(to_par[1]);
    close(to_par[0]); close(to_par[1]);
    close(to_chil[0]); close(to_chil[1]);
    for (; ;) {
      if ((count = read(0, buf, sizeof(buf)))
                                        == 0)
        exit(0);
      write(1, buf, count);
    } /* for */
  } /* if */
  close(1);
  dup(to_chil[1]);
  close(0);
  dup(to_par[0]);
  close(to_par[0]);
  close(to_par[1]);
  close(to_chil[0]);
  close(to_chil[1]);
  for (i = 0; i < 15; i++) {
    write(1, string, stlen(string));
    read(0, buf, sizeof(buf));
  }  /* for */
}
```

Sample Program
(pipe, dup, fork)

9

Signals

- *Signals* inform processes of the occurrence of asynchronous events.
- Processes may send each other *signals* with the *kill* system call, or the kernel may send *signals* internally.
- There are several signals (e.g., 31 signals in FreeBSD) that can be classified as follows:
  - signals having to do with termination of a process, sent when a process *exits* or when a process invokes the *signal* system call with the *death of child* parameter.
  - signals having to do with process induced exception such as when a process accesses an address outside its virtual address space or when it executes privileged instruction.

10

Signals

- Classification of signals (continued)
  - signals having to do with the unrecoverable conditions during a system call, such as running out of system resources during *exec*.
  - signals caused by unexpected error condition during a system call, such as making a nonexistent system call, writing a pipe that has no reader processes, etc.
  - signals originating from a process in user mode, such as when a process wishes to receive an *alarm* signal after a period of time or when processes send arbitrary signals to each other with the *kill* system call.
  - signals related to terminal interaction such as when a user hangs up a terminal, or when a user presses Ctrl-C.
  - signals for tracing execution of a process.

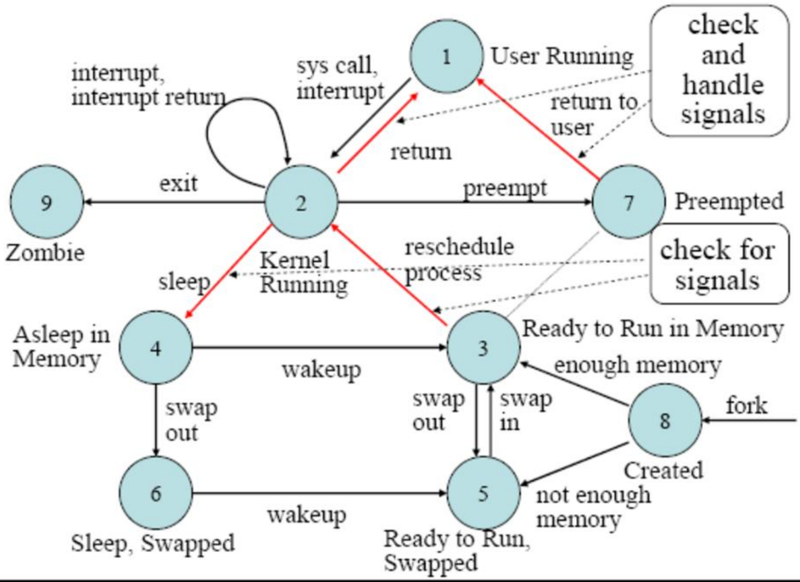
11

Sending Signals

- To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received.
- If the process is asleep at an interruptible priority, the kernel awakens it.
- A process can remember different type of signals, but it has no memory of how many signals it receives of a particular type.
- The kernel checks for receipt of a signal when a process is about to return from kernel mode to user mode and when it enters or leaves the sleep state at a suitable low scheduling priority.

12

Checking and Handling Signals



13

Handling Signals

- The kernel handles signals only when a process returns from kernel mode to user mode. Thus, a signal does not have an instant effect on a process running in kernel mode.
- If a process is running in user mode, and the kernel handles an interrupt that causes a signal to be sent to the process, the kernel will recognize and handle the signal when it returns from the interrupt.
- Thus, a process never executes in user mode before handling outstanding signals.
- A process can choose to ignore signals with the *signal* system call.
  - currently, *signal* is a simplified interface to *sigaction* system call

14

```
algorithm issig                /* test for receipt of signals */
input: none
output: true, if process received signals that it does not ignore
        false otherwise
{
    while (received signal field in process table entry not 0) {
        find a signal number sent to the process;
        if (signal is death of child) {
            if (ignoring death of child signals)
                free process table entries of zombie children;
            else if (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);
}
```

Algorithm for recognizing signals

15

Handling Signals

- The kernel handles signals in the context of the process that receives them so a process must run to handle signals.
- There are three cases for handling signals:
  - the process *exits* on receipt of the signal,
  - it ignores the signal, or
  - it executes a particular (user) function on receipt of the signal.
- The default action is to call *exit* in kernel mode, but a process can specify special action to take on receipt of certain signals with the *signal* system call.

16

Handling Signals

- syntax: old function = signal(signum, function);
  - function = 1: ignore *signum*
  - function = 0: *exit* (default)
  - function = otherwise: address of the signal handler function
- The *u area* contains an array of signal-handler fields, one for each signal defined in the system.
- The kernel stores the address of the user function in the field that corresponds to the signal number.

17

```
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;
      if (user specified function to handle the signal) {
            get user virtual address of signal catcher stored in u area;
            clear u area entry that stored address of signal catcher;
            modify user level context: artificially create user stack frame to
                        mimic call to signal catcher function;
            modify system level context: write address of signal catcher into
                        program counter field of user saved register context;
            return;
      }
      if (signal is type that system should dump core image of process) {
            create file named "core" in current directory;
            write contents of user level context to file "core";
      }
      invoke exit algorithm immediately;
}
```

18

Handling Signals: Core Dump

- If the signal handling function is set to its default value, the kernel will dump a "core" image of the process for certain types of signals before *exiting*.
  - the dump is a convenient to programmers for debugging.
- The kernel dumps core for signals that imply something is wrong with a process.
  - e.g., executing an illegal instruction, accessing an address outside its virtual address space.
- But the kernel does not dump core for signals that do not imply a program error.
  - e.g., receipt of an interrupt signal (ctrl-C).

19

Handling Signals: Catching Signals

- If a process receives a signal that it had previously decided to catch, it executes the user specified signal handling function immediately when it return to user mode, after the kernel does the following steps.

1. The kernel accesses the user saved register context, finding the program counter and stack pointer that it had saved for return to the user process.
2. It clears the signal handler field in the *u area*.
3. The kernel creates a new stack frame on the user stack, writing in the values of the program counter and stack pointer it had retrieved from the user saved register context.

20

Handing Signals: Catching Signals

4. The kernel changes the user saved register context: it resets the value for program counter to the address of the signal catcher function and sets the value for stack pointer to account for the growth of the user stack.

- After returning from the kernel to user mode, the process will thus execute the signal handling function; when it returns from the signal handling function, it returns to the place in the user code where the system call or interrupt originally occurred, mimicking a return from the system call or interrupt.

21

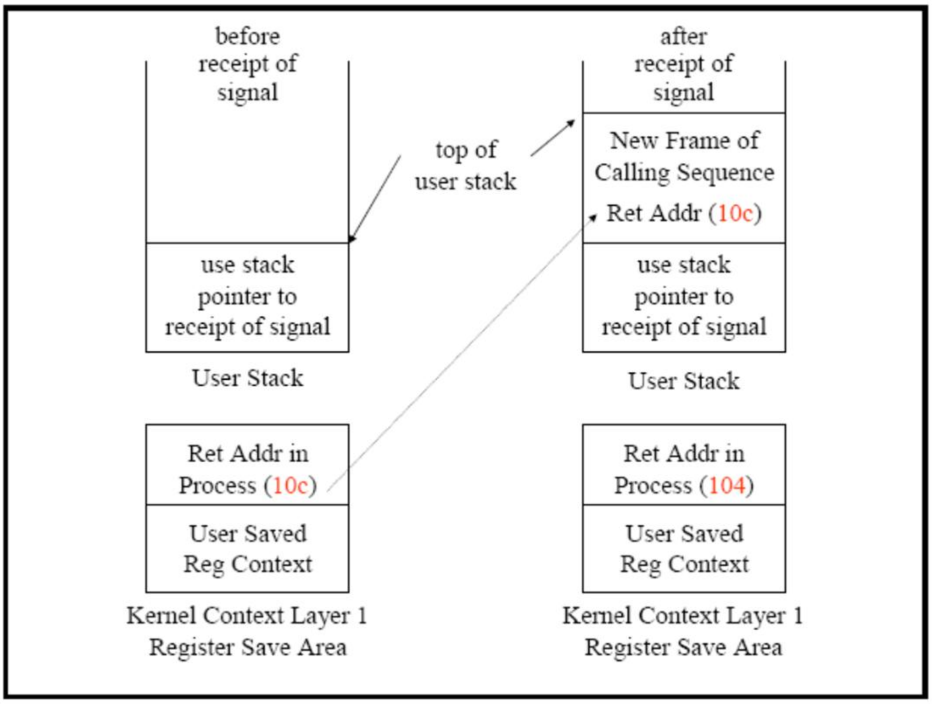
```
#include <signal.h>
main( )
{
   extern catcher( );
   signal(SIGINT, catcher);
   kill(0, SIGINT);
}
```

Source code
that catches signals

```
_main( )
   . . . . . . . .
    f5:  pushl      $0x2
    f7: clrl        -(sp)
# next line calls kill library routine
    f9: calls       $0x2,08(pc)
   100: ret
   . . . . . . . .
_catcher( )
   104:
   106: ret
   . . . . . . . .
_kill( )
   108:
# next line traps into kernel
   10a: chmk        $0x25
   10c: bgequ       0x6 <0x114>
   . . . . . . . .
```

Disassembly of program
that catches signals

22



23

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

main( )
{
        int ppid;
        signal(SIGINT, sigcatcher);
        if (fork( ) == 0) {
                sleep(5);  /* give enough time to set up */
                ppid = getppid( );
                for (; ; )
                        if (kill(ppid, SIGINT) = = −1)
                                exit( );
        }
        nice(10);  /* lower priority */
        for (; ;) ;
}
```

Race condition
in catching signals

24

Signal Handling in FreeBSD

- The BSD system allows a process to block and unblock receipt of signals by *sigprocmask* system call; when a process unblocks signals, the kernel sends pending signals that had been blocked to the process.
- When a process receives a signal, the kernel automatically blocks further receipt of the signal until the signal handler completes.
- This is analogous to how the kernel reacts to hardware interrupts: it blocks report of new interrupts while it handles previous interrupts.

25

Process Group

- the system sometimes identifies processes by "group."
  - e.g., processes with a common ancestor process that is a login shell are generally related, and therefore all such processes receive signals when a user hits Ctrl-C.
- The kernel uses the *process group ID* to identify groups of related processes that should receive a common signal for certain events.
- The *setpgrp* system call initializes the process group number of a process and sets it equal to the value of its process ID.
- A child retains the process group number of its parent during *fork*.

26

Process Termination

- Processes terminate by executing the *exit* system call.
- An *exiting* process enters the zombie state, relinquishes its resources, and dismantles its context except for its slot in the process table.
- syntax: exit(status);
  - the value of *status* is returned to the parent process for its examination.
- Process may call *exit* explicitly or implicitly at the end of a program.
  - the kernel may invoke *exit* internally for a process on receipt of uncaught signals. If so, the value of *status* is the signal number.

27

```
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 (iput);
    release current (changed) root, if exists (iput);
    free regions, memory associated with process (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;
}
```

28

Awaiting Process Termination (1/2)

- A process can synchronize its execution with the termination of a child process by executing the *wait* system call.
- syntax: pid = wait(stat_addr);
- If the process executing *wait* has child processes but none are zombie, it sleeps at an interruptible priority until arrival of a signal.
- The kernel does not contain an explicit wake up call for a process sleeping in *wait*: such processes only wake up on receipt of signals.
  - for any signals except "death of child," the process will react as described previously.

29

Awaiting Process Termination (2/2)

- If the signal is "death of child," the process may respond differently.
  - in the default case, it will wake up from its sleep in *wait*, and *sleep* invokes *issig* to check for signals. *Issig* recognizes the special case of "death of child" signals and returns "false." Consequently, the kernel does not "long jump" from *sleep*, but returns to *wait*. The kernel will restart the *wait* loop, find a zombie child, release the child's process table slot, and return from the *wait* system call.
  - If the process ignores "death of child" signals, the kernel restarts the *wait* loop, frees the process table slots of zombie children, and searches for more children.
  - If the process catches "death of child" signals, it does so.

30

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

31

Invoking Other Programs

- The *exec* system call invokes another program, overlaying the memory space of a process with a copy of an executable file.
- The contents of the user-level context that existed before the *exec* call are no longer accessible afterward except for *exec*'s parameters, which the kernel copies from the old address space to the new address space.
- syntax: execve(finename, argv, envp);

32

```
algorithm exec
input: (1) file name, (2) parameter list, (3) environment variables list
output: none
{
    get file inode (namei);
    verify file executable, use 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 (detachreg);
    for (every region specified in load module) {
        allocate new regions (allocreg);
        attach the regions (attachreg);
        load region into memory if appropriate (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);
}
```

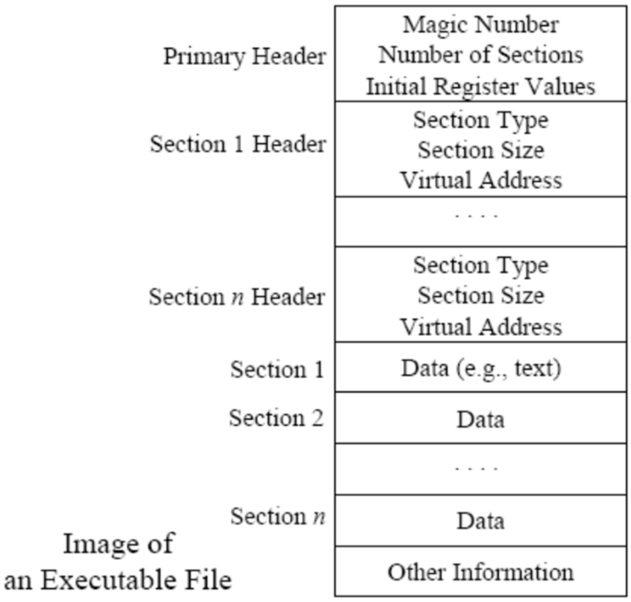
33

Executable File

- An executable file consists of four parts:

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

34



35

Allocating a Text Region

- When the kernel allocates a text region in *exec*, it checks if the executable file allows its text to be shared.
- If so, it follows algorithm *xalloc* to find an existing region for the file text or to assign a new one.
- In *xalloc*, the kernel searches the active region list for the file's text region, identifying it as the one whose inode pointer matches the inode of the executable fie.
- If the kernel locates a region that contains the file text , it makes sure that the region is loaded into memory and attaches it to the process.
- If not, the kernel allocates a new region (*allocreg*), attaches it to the process (*attachreg*), loads it into memory (*loadreg*), and changes its protection to read-only.

36

```
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;
    while (contents of region not ready) {
      increment region reference count;
      unlock region;
      sleep (event contents of region ready);
      lock region;
      decrement region reference count;
    } /* while */
    attach region to process (attachreg);
    unlock region;
    return;
  } /* if */
  /* no such text region exists */
  allocate text region (allocreg);
  if (inode mode has sticky bit set)
    turn on region sticky flag;
  attach region to virtual address
      indicated by inode file header;
      (attachreg);
  if (file specially formatted for
        paging system)
    /* later */
  else
    read file text into region (loadreg);
  change region protection in
      per process region table to
      read-only;
  unlock region;
}
```

37

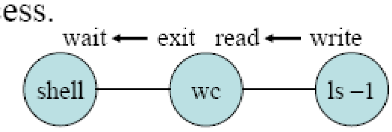
Sticky Bit

- The capability to share text regions allows the kernel to decrease the startup time of an *exec*ed program by using the *sticky-bit*.
- System administrators can set the sticky-bit file mode for frequently used executable files.
- When a process executes a file that has its sticky-bit set, the kernel does not release the memory allocated for text when it later detaches the region during *exit* or *exec*, even if the region reference count drops to 0.
- The FreeBSD VM system totally ignores the sticky bit.

38

Shell

- The shell *reads* a command line from its standard input and interprets it.
- The shell *forks* and creates a child process, which *execs* the program that the user specified on the command line.
  - the parent process *waits* until the child process *exits* from the command and then loops back to *read* the next command.
  - "&" specifies to run a process asynchronously (background).
- The shell supports *pipe* to connect *stdout* of a process to *stdin* of another process.
  - e.g., "ls –1 | wc"



39

```
/* read command line until EOF */
while (read(stdin, buffer, numchars)) {
   /* parse command line */
   if (/* command line contain & */)
      amper = 1;
   else
      amper = 0;
   /* for commands not part of shell */
   if (fork( ) = = 0) {
      /* redirection of I/O? */
      if (/* redirect output */) {
         fd = creat(newfile, fmask);
         close(stdout);
         dup(fd);
         close(fd);
         /* stdout is redirected*/
      } /* if redirect */
```

Main loop of shell

```
      if (/* piping */) {
         pipe(fildes);
         if (fork( ) = = 0) {
            /* 1st command component */
            close(stdout);
            dup(fildes[1]);
            close(fildes[1]);
            close(fildes[0]);
            execlp(command1, …);
         } /* fork */
         /* 2nd command component */
         close(stdin);
         dup(fildes[0]);
         close(fildes[0]);
         close(fildes[1]);
      } /* if (piping) */
      execve(command2, … );
   } /* fork */
   if (amper = = 0)
      retid = wait(&status);
} /* while */
```

40

System Boot

- Bootstrap: to get a copy of the operating system into machine memory and to start executing it.

1. The bootstrap procedure eventually reads the boot block (block 0) of a disk, and loads it into memory.
2. The program contained in the boot block loads the kernel from the file system.
3. The boot program transfers control to the start address of the kernel, and the kernel starts running (algorithm *start*).

41

Start Algorithm (1/2)

- The kernel initializes its internal data structures.
  - linked list of free buffers and inodes, hash queues for buffers and inodes, region structures, page table entries, etc.
- The kernel *mount*s the root file system onto root ("/") and generates the environment for process 0.
  - creating a *u area*, initializing slot 0 in the process table, etc.
- When the environment of process 0 is set up, the system is running as process 0.
- Process 0 *fork*s, and the new process, process 1, creates its user-level context.
  - it allocates data region and attaches it to its address space.
  - it grows the region to its proper size and copies code from the kernel address space to the new region.

42

Start Algorithm (2/2)

- Process 1 sets up the saved user register context, "returns" from kernel to user mode, and executes the code it has just copied from kernel.
  - process 1 is a user-level process while process 0 is a kernel-level process.
- The text for process 1 consists of a call to the *exec* system call to execute "/etc/init".
  - in current BSD, "/sbin/init" is executed.
- Process 1 is commonly called *init* because it is responsible for initialization of new processes.

43

```
algorithm start   /* system startup procedure */
input: none
output: none
{
    initialize all kernel data structures;
    pseudo-mount of root;
    hand-craft environment of process 0;
    fork process 1:
    {   /* process 1 here */
        allocate region;
        attach region to init address space;
        grow region to accommodate code
           about to copy in;
        copy code from kernel space to init
           user space to exec init;
        change mode: return from kernel to
           user mode;
        /* init never gets here – as result of
         * above change mode, init exec's
         * /etc/init and  becomes a normal
         * user process
         */
    }
    /* 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;
```

Booting the system

44

```
algorithm init       /* init process, process 1 of the system */
input none
output none
{
     fd = open("/etc/inittab", O_RDONLY);
     while (line_read(fd, buffer)) {
          if (invoked state != buffer state)
               continue;
          /* state matched */
          if (fork( ) = = 0) {
               execl("process specified in buffer");
               exit( );
          }  /* init process does not wait */
     }
     while ((id = wait(init *)0)) != – 1) {
          /* check here if a spawned child died;
           * consider respawning it.
           * otherwise, just continue;
     }
}
```

45

End

46