

OPERATING SYSTEMS (1)

- An **operating system (OS)** is a program that performs system startup and controls the execution of all other programs
 - ▷ Provides a higher level of abstraction than the instruction set architecture
 - ▷ User programs don't have to know any details of the hardware or the instruction set
 - ▷ The **kernel** is the part of an OS that:
 - Runs at the highest **privilege level**
 - ◇ The kernel's private address space ($\geq 0x80000000$ in MIPS R2000) is processor-locked against access by less privileged programs
 - Mediates access by all user programs to the hardware
 - ◇ CPU, memory, I/O devices
 - Mediates access by all user programs to basic software routines
 - ◇ File operations, communication with other processes, network

EXAMPLES OF OPERATING SYSTEMS (1)

- **MVS** (aka OS/390 or z/OS)
 - ▷ Used on IBM mainframe computers
 - ▷ Most stable and secure of all OS's
- **UNIX**
 - ▷ Used on many servers and special-purpose desktops (IC design, etc.)
 - ▷ Highly stable and secure
 - ▷ Originated at AT&T
 - ▷ Many variants
 - Open-source
 - ◇ FreeBSD, Mac OS X (Darwin kernel only)
 - Proprietary
 - ◇ Solaris (Sun Microsystems), HP-UX (Hewlett-Packard)
 - ▷ Open-API (application programming interface) standard: **POSIX**

EXAMPLES OF OPERATING SYSTEMS (2)

- **Linux**

- ▷ Kernel created by a student, Linus Torvalds, to provide a free, open-source, UNIX-like OS
- ▷ Supported by thousands of programmers world-wide
- ▷ Favored by industry for its customizability

- **Microsoft Windows**

- ▷ Proprietary
- ▷ Most widely-installed OS
- ▷ Some applications created for Windows are default standards (Microsoft Office suite)
- ▷ Widely criticized by many IT professionals for bugs, security problems, and instability

OPERATING SYSTEMS (2)

- A **microkernel** is a kernel that provides only the most essential services
 - ▷ Memory management, interprocess communication, basic scheduling
 - ▷ Examples: `vmunix`, `vmlinux`, `System`, `kernel32.dll`
- Programs issue **system calls** to request execution of a kernel routine (file access, I/O, interprocess communication, network protocols)
 - ▷ The kernel's **call handler** uses information passed as arguments in the system call to decide what action to take
 - ▷ After servicing a system call, the kernel can resume execution of, suspend, or terminate the process that issued the system call
 - ▷ The kernel can schedule other tasks while a process awaits the return from a system call (if the system call returns to the process)

OPERATING SYSTEMS (3)

- Services provided by modern operating systems:
 - ▷ Memory management (virtual memory)
 - ▷ Priority-controlled execution of applications and utility programs
 - The OS calls other programs as if they were procedures, and allows them to run until their time is up or an exception occurs
 - ▷ Interprocess communication
 - ▷ Access to I/O devices (screen, mouse, keyboard, printer, ...)
 - A **driver** is a routine that issues commands to an I/O device
 - ▷ Access to files
 - Actions include open, close, rewind, seek
 - Permissions (stored with each file) control access
 - ▷ Detection and handling of events that interrupt program execution
 - Hardware interrupts, processor exceptions, protection faults
 - ▷ Accounting

REFERENCES ON OPERATING SYSTEMS

Nothing can take the place of a good course, but these books may help:

1. *Operating Systems: Internals and Design Principles*, 3rd Edition, by William Stallings (Prentice Hall, 1998; ISBN 0-13-887407-7).
2. *Operating Systems: Design And Implementation*, 2nd Edition, by Andrew S. Tanenbaum (Prentice Hall, 1997; ISBN 0-13-638677-6).
3. *The Design and Implementation of the 4.4BSD Operating System*, by Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman (Addison-Wesley, 1996; ISBN 0-201-54979-4).
4. *Design of the UNIX Operating System*, by Maurice Bach (Prentice Hall, 1987; ISBN 0-13-201799-7).

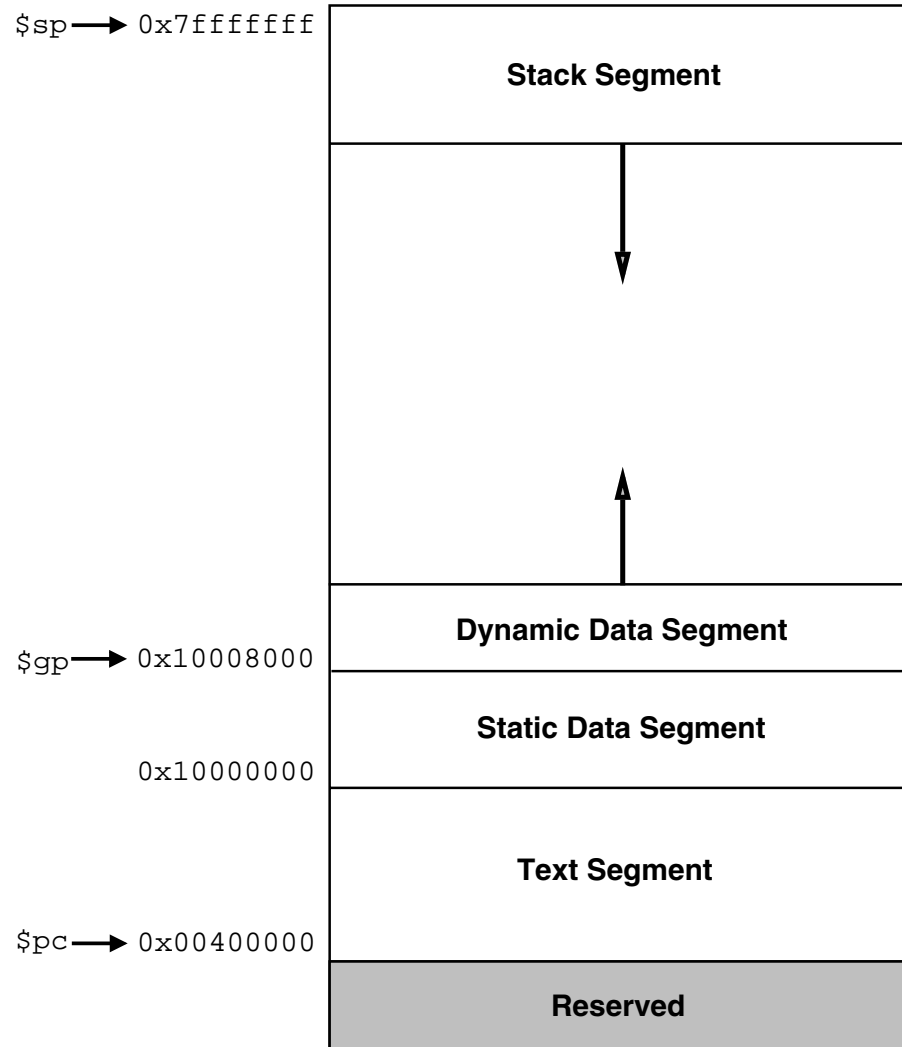
HOW A COMPUTER RUNS PROGRAMS (1)

- Single-tasking operating system (PC-DOS, 1950's mainframe systems)
 - ▷ Only one program can run at a time
 - ▷ User programs can call OS functions directly (BIOS routines in PC)
 - ▷ User programs can issue commands directly to the hardware
 - Affords easy entry for destructive viruses (which can execute `FORMAT C:`, for example)
 - ▷ Division between OS memory segment and user memory segment is voluntary (not enforced by hardware)
 - ⇒ User programs can clobber the OS memory segment

HOW A COMPUTER RUNS PROGRAMS (2)

- Multi-tasking operating system (UNIX, MVS, Windows NT, MacOS)
 - ▷ Many programs can run simultaneously (though only one at a time can change the program counter)
 - ▷ Each user program “sees” a **virtual machine**
 - Address space in memory
 - Registers
 - Program counter
 - ▷ The operating system has a memory-resident **kernel**, which provides low-level I/O, hardware access and security
 - In the MIPS R2000 ISA, kernel addresses have the most significant bit set (kernel addresses are $\geq 0x8000\ 0000$)
 - Kernel functions and the kernel’s memory space can be accessed only in a special CPU mode (**kernel mode**)
 - User programs obtain services from the OS through **system calls**

MIPS USER VIRTUAL ADDRESS SPACE



PROCESSES (1)

- A **program** is an executable file
 - ▷ All C programs that run under Solaris have the interface `main(int argc, char *argv[], char *envp[])`
- A **process** is an instance of a program in execution
 - ▷ In the UNIX OS, a process is created by a system call (`fork`)
 - ▷ In a multitasking OS, one user can execute many processes simultaneously
 - ▷ Many instances of the same program can execute simultaneously (e.g., UNIX shell commands such as `cd`, `cp`, etc.)
 - ▷ A process terminates:
 - When it reaches the end of its `main()` function
 - When the kernel kills it because of timeout, an exception, termination of its parent process, or user action

HOW A COMPUTER LAUNCHES A PROCESS

- All C programs that run under Solaris have the interface `main(int argc, char *argv[], char *envp[])`
- To launch a user process, the kernel
 - ▷ Creates an address space for the new process
 - ▷ Copies the executable text module into the address space
 - ▷ Pushes the command-line arguments and the environment arguments onto the stack
 - ▷ Calls `main` as a procedure

ARGUMENTS AND ENVIRONMENTAL VARIABLES

- C program `prarg.c` to obtain values of arguments:

```
#include <stdio.h>
main(int argc, char *argv[], char *envp[])
{
    int index;
    for (index=0; argv[index] != NULL; index++)
        printf("argv[%d] = %s\n", index, argv[index]);
}
```

- C program `prenv.c` to obtain values of environmental variables:

```
#include <stdio.h>
main(int argc, char *argv[], char *envp[])
{
    int index;
    for (index=0; envp[index] != NULL; index++)
        printf("envp[%d] = %s\n", index, envp[index]);
}
```

VALUES OF ARGUMENTS

```
thor% cc prarg.c
thor% mv a.out prarg
thor% prarg arg1 arg2 arg3 arg4
argv[0] = prarg
argv[1] = arg1
argv[2] = arg2
argv[3] = arg3
argv[4] = arg4
```

VALUES OF ENVIRONMENTAL VARIABLES

```
thor% cc prenv.c
thor% mv a.out prenv
thor% prenv
envp[0] = HOME=/home/thor/cantrell
envp[1] = PATH=/bin:/home/thor/cantrell/bin:/usr/bin:<snip>
envp[2] = LOGNAME=cantrell
envp[3] = HZ=100
envp[4] = TERM=xterm
envp[5] = TZ=US/Central
envp[6] = SHELL=/bin/csh
envp[7] = MAIL=/var/mail/cantrell
envp[8] = PWD=/home/thor/cantrell/codes/c.progs
envp[9] = USER=cantrell
envp[10] = HOST=thor
.
.
.
```

XSPIM STARTUP CODE BEFORE LOADING A PROGRAM

```
[0x00400000] 0x8fa40000 lw $4, 0($29)    #a0 points to argc (top of stack)
[0x00400004] 0x27a50004 addiu $5, $29, 4 #a1 points to argv (next)
[0x00400008] 0x24a60004 addiu $6, $5, 4  #a2 would point to envp if there
                                     # were no command-line arguments
[0x0040000c] 0x00041080 sll $2, $4, 2    #multiply argc by 4 (gives number
                                     # of bytes needed to hold pointers
                                     # to arguments)
[0x00400010] 0x00c23021 addu $6, $6, $2  #a2 now points to envp, with
                                     # space needed for arguments
                                     # taken into account
[0x00400014] 0x0c000000 jal 0x00000000 [main] #main called as a procedure
[0x00400018] 0x3402000a ori $2, $0, 10   #$v0 holds number of system call
                                     # (exit routine, in this case)
[0x0040001c] 0x0000000c syscall    #system call
```

XSPIM STARTUP CODE AFTER LOADING A PROGRAM

```
[0x00400000] 0x8fa40000 lw $4, 0($29)    # $a0 points to argc (top of stack)
[0x00400004] 0x27a50004 addiu $5, $29, 4 # $a1 points to argv (next)
[0x00400008] 0x24a60004 addiu $6, $5, 4  # $a2 would point to envp if there
                                     # were no command-line arguments
[0x0040000c] 0x00041080 sll $2, $4, 2    # multiply argc by 4 (gives number
                                     # of bytes needed to hold pointers
                                     # to arguments)
[0x00400010] 0x00c23021 addu $6, $6, $2  # $a2 now points to envp, with
                                     # space needed for arguments
                                     # taken into account
[0x00400014] 0x0c100008 jal 0x00400020  # main called as a procedure
                                     # (note that the loader has
                                     # inserted the correct address)
[0x00400018] 0x3402000a ori $2, $0, 10  # $v0 holds number of system call
                                     # (exit routine, in this case)
[0x0040001c] 0x0000000c syscall      # system call
[0x00400020] 0x3c071001 lui $7, 4097 [ar1] # start of "main"
```

...

XSPIM STACK BEFORE jal main (1)

STACK

| | | | | |
|---------------|------------|------------|------------|------------|
| [0x7fffebe0] | 0x00000001 | 0x7fffec7c | 0x00000000 | 0x7fffe4e4 |
| [0x7fffebfb0] | 0x7fffef30 | 0x7fffef1f | 0x7fffef18 | 0x7fffef0d |
| [0x7fffec00] | 0x7fffeeff | 0x7fffeef0 | 0x7fffeed8 | 0x7fffeec0 |
| [0x7fffec10] | 0x7fffeeb2 | 0x7fffeea8 | 0x7fffee99 | 0x7fffee8b |
| [0x7fffec20] | 0x7fffee72 | 0x7fffee4b | 0x7fffee22 | 0x7fffedb6 |
| [0x7fffec30] | 0x7fffed9b | 0x7fffed7e | 0x7fffed60 | 0x7fffed55 |
| [0x7fffec40] | 0x7fffed49 | 0x7fffed3a | 0x7fffed2e | 0x7fffed22 |
| [0x7fffec50] | 0x7fffed18 | 0x7fffed10 | 0x7fffed05 | 0x7fffecf5 |
| [0x7fffec60] | 0x7fffecdd | 0x7fffecb9 | 0x7fffec9f | 0x00000000 |
| [0x7fffec70] | 0x00000000 | 0x00000000 | 0x00000000 | 0x2f686f6d |
| [0x7fffec80] | 0x652f7468 | 0x6f722f63 | 0x616e7472 | 0x656c6c2f |
| [0x7fffec90] | 0x696e6974 | 0x31642d78 | 0x7370696d | 0x2e730044 |
| [0x7fffecca0] | 0x4953504c | 0x41593d31 | 0x32392e31 | 0x31302e38 |
| ... | | | | |
| [0x7fffe4e0] | 0x62696e00 | 0x484f4d45 | 0x3d2f686f | 0x6d652f74 |
| [0x7fffe4f0] | 0x686f722f | 0x63616e74 | 0x72656c6c | 0x00000000 |

XSPIM STACK BEFORE jal main (2)

| STACK | | | | |
|--------------|------------|------------------------|--------------------------|---------------------------|
| [0x7fffebe0] | 0x00000001 | 0x7fffec7c | 0x00000000 | 0x7fffe4 |
| | ~~~~~ | ~~~~~ | ~~~~~ | ~~~~~ |
| | argc = 1 | pointer to argument | terminating null word | pointer to 1st env var |
| ... | | | | |
| [0x7fffec70] | 0x00000000 | 0x00000000 | 0x00000000 | 0x2f686f6d |
| | ~~~~~ | ~~~~~ | ~~~~~ | ~~~~~ |
| | / h o m | | | |
| [0x7fffec80] | 0x652f7468 | 0x6f722f63 | 0x616e7472 | 0x656c6c2f |
| | ~~~~~ | ~~~~~ | ~~~~~ | ~~~~~ |
| | e / t h | o r / c | a n t r | e l l / |
| [0x7fffec90] | 0x696e6974 | 0x31642d78 | 0x7370696d | 0x2e730044 |
| | ~~~~~ | ~~~~~ | ~~~~~ | ~~~~~ |
| | i n i t | 1 d - x | s p i m | . s |

- Recall that the first argument of every procedure is the procedure's filename

PROCESSES (2)

- When the UNIX kernel creates a process, it creates a **process structure** that contains:
 - ▷ Process number, user & group IDs, privileges
 - ▷ Processor state information
 - User-visible registers, program counter, status register, etc.
 - Stack pointers to system stack(s) employed by the process for procedure calls
 - ▷ Process control information
 - Scheduling information (process ready for execution, blocked, waiting for I/O, waiting for a specific event, etc.)
 - Files opened or used by the process
 - Data structuring (pointers to other processes in a queue, etc.)
 - Interprocess communication

INSPECTION OF RUNNING PROCESSES USING ps

apache% ps -ef | more

| UID | PID | PPID | C | STIME | TTY | TIME | CMD |
|-------|-------|-------|---|----------|--------|-------|--------------------------------|
| root | 0 | 0 | 0 | Apr 20 | ? | 0:15 | sched |
| root | 1 | 0 | 0 | Apr 20 | ? | 2:05 | /etc/init - |
| root | 2 | 0 | 0 | Apr 20 | ? | 0:22 | pageout |
| root | 3 | 0 | 1 | Apr 20 | ? | 83:17 | fsflush |
| ggb | 27691 | 27676 | 0 | 07:58:22 | pts/21 | 0:01 | telnet alanthia.gator.net 1536 |
| root | 26125 | 1 | 0 | Apr 25 | ? | 0:37 | /usr/sbin/syslogd |
| root | 110 | 1 | 0 | Apr 20 | ? | 0:45 | /usr/sbin/rpcbind |
| root | 518 | 1 | 0 | Apr 20 | ? | 0:00 | /usr/lib/saf/sac -t 300 |
| root | 140 | 1 | 0 | Apr 20 | ? | 0:57 | /usr/sbin/inetd -s -t |
| dkw | 22751 | 20173 | 0 | Apr 25 | pts/51 | 0:00 | vi project.pl |
| root | 118 | 1 | 0 | Apr 20 | ? | 0:02 | /usr/sbin/nis_cachemgr |
| root | 147 | 1 | 0 | Apr 20 | ? | 0:01 | /usr/lib/nfs/lockd |
| root | 145 | 1 | 0 | Apr 20 | ? | 0:06 | /usr/lib/nfs/statd |
| root | 195 | 1 | 0 | Apr 20 | ? | 21:47 | /usr/lib/autofs/automountd |
| liux | 29777 | 29771 | 0 | 08:48:40 | pts/65 | 0:00 | telnet titan |
| root | 19238 | 140 | 0 | 00:54:19 | ? | 0:00 | in.telnetd |
| nigel | 29379 | 29377 | 0 | Apr 21 | pts/59 | 0:00 | -bash |

PROCESSES (3)

- When the UNIX kernel creates a process, it allocates a standard **process address space** that includes:
 - ▷ User data segment(s)
 - ▷ User text segment(s)
 - ▷ System stack(s) to support procedure calls
 - ▷ An initially unallocated **protected region** from which the process can allocate space using `malloc()`

INSPECTION OF PROCESS ADDRESS SPACE

```
thor% /usr/proc/bin/pmap 373
373:   -csh -c unsetenv _ PWD; unsetenv DT; setenv DISPLAY :
00010000   144K read/exec      /usr/bin/csh
00042000    24K read/write/exec /usr/bin/csh
00048000   192K read/write/exec [ heap ]
EF680000   592K read/exec      /usr/lib/libc.so.1
EF722000    32K read/write/exec /usr/lib/libc.so.1
EF72A000    8K read/write/exec   [ anon ]
EF770000   16K read/exec      /usr/platform/sun4u/lib/libc_psr.so.1
EF790000    8K read/exec      /usr/lib/libmapmalloc.so.1
EF7A0000    8K read/write/exec /usr/lib/libmapmalloc.so.1
EF7B0000    8K read/exec      /usr/lib/libdl.so.1
EF7C0000    8K read/write/exec [ anon ]
EF7D0000   112K read/exec      /usr/lib/ld.so.1
EF7FA000    8K read/write/exec /usr/lib/ld.so.1
EFFF6000   40K read/write/exec [ stack ]
total    1200K
```