

PROCEDURES (1)

- Why use procedures or function calls?
 - ▷ To be able to re-use the same code at many points in a program
 - ▷ Modularity simplifies program writing and debugging
 - Hierarchical design — a good practice when designing any complex system, software or hardware
 - Each module is defined by its inputs and outputs
 - Different programmers can work on the same program simultaneously
 - Each module can be debugged independently of the others using simulated inputs

PROCEDURES (2)

- Steps in transferring to and from a subprogram (procedure or function)
 - ▷ Save the return address
 - The **return address** is the address of the instruction (in the program that calls the procedure) which is to be executed after the subprogram has finished running
 - ▷ Issue the procedure call instruction
 - ▷ Execute the instructions in the subprogram
 - ▷ Return to the calling program
- Passing data to and from a procedure
 - ▷ Data can be passed either in registers or on a stack (or both)
 - ▷ Pass by value: The actual value of the data passed to, or returned from, the subprogram, is put into a register or pushed onto the stack
 - ▷ Pass by name: A pointer to the data is put into a register or pushed onto the stack

PROCEDURES (3)

- Structure of a C program including function (subprogram) calls:

```
void main(int argc, char *argv[], char *envp[]){
    int p, s;
    s = some_library_function(argv[2], envp[3]);
    p = func1(s, argv[1]);
}
int func1(int j, char *george){
    /* executable statements that use j and george */
return henry;}
```

- ▷ The first statement defines **main** as a function that returns no value
- ▷ The operating system passes 3 arguments to **main**: The argument count (**argc**), a pointer to an array of arguments (**argv[]**) and a pointer to an array of environmental parameters (**envp[]**)
- ▷ A subprogram can be defined in the same source file as **main**, or in a different source file, or in a library of pre-compiled functions

SUBPROGRAM CALLS IN THE MIPS ISA

- Up to 4 arguments can be passed in the registers `$a0` through `$a3` (`$4` through `$7`)
- The stack may be used for arguments, return addresses and register contents
 - ▷ For a subprogram that is nested to a depth of only 1, no stack is needed unless a large number of variables or pointers to arrays need to be passed
 - ▷ See [later slides](#) for stack usage
- The C statement

`return a`

causes the value of `a` to be passed to the calling program in register `$v0` (`$2`)

- ▷ A second value can be returned in register `$v1` (`$3`) (not done by the `return` statement in C)

PROCEDURES (4)

- Instructions for saving a return address and calling a procedure:

- ▷ SAL: The calling program issues the instructions

```
    la proc1_ret, ret_addr
    b  proc1
```

`ret_addr:` (next instruction)

- Saves address of label `ret_addr` in location labeled `proc1_ret`
- Jumps to instruction labeled `proc1`

- ▷ MIPS assembly language: The calling program issues the instructions

```
jal proc1
(next instruction)
```

- Saves return address in register `$31`
- Jumps to instruction labeled `proc1`
- The instruction `jalr rd rs` saves the return address in register `rd` and jumps to the instruction whose address is in register `rs`

PROCEDURES (5)

- Instructions for returning from a subprogram (function or procedure):
 - ▷ MIPS assembly language: The subprogram issues the instruction
`jr $31`
 - Jumps to instruction whose address is in register `$31`
 - In principle, any other register can be used to save the return address
 - If subprograms are deeply nested, each return address must be saved on the stack
 - ▷ SAL: The subprogram issues the instruction
`done`
 - Equivalent to `jr $31`

PROCEDURES (6)

- Support for procedure calls in MIPS assembler:
 - ▷ In general, assemblers for RISC architectures give the programmer the bare minimum of support for procedures
 - ▷ Special branch instructions:
 - `jal label` – Loads the value of the program counter (the address of the instruction that follows the `jal` instruction) into register `$31 ($ra)` and loads the address designated by `label` into the program counter
 - `jalr [rd] rs` – Saves the program counter in register `rd` and jumps to the instruction whose address is in register `rs`
 - ◇ If `rd` is omitted, it defaults to `$31`
 - `jr $31` – Loads the word at the address pointed to by `$31 ($ra)` into the program counter
 - ▷ The programmer must pass arguments, get the result(s) returned by the procedure, and manage the registers
 - ▷ Many levels of procedure calls \Rightarrow a stack is the ideal data structure

```

# init1d-proc.s
# MIPS assembler code to initialize the elements of a 1-d array
# with the value stored in location val, using a procedure call
#
# Please load this program into SPIM and single-step through it,
# while watching the register, text and data windows, and the console
#
# Data segment
#
.data
arl:      .word    0:20      # array of 20 integers (4 bytes each)
val:      .float   1.e-3     # value to store in array elements
veclen:   .word    20       # vector length (number of elements)
size:     .word    4        # size of an array element, in bytes

.text
__start:
    la      $a0,arl          # First argument: a pointer to arl[0]
    lw      $a1,size        # Second argument: value of size
    lw      $a2,veclen      # Third argument: value of vector length
    lw      $a3,val         # Fourth argument: value to be stored
    jal     initv
    ori     $v0,$0,10       # These two instructions are equivalent
    syscall                # to the SAL command "done"

initv:   nop                # Entry point to procedure initv
        blez    $a2,beamup  # If veclen<=0, I'm outta here
        or     $t0,$0,$a0    # Reg. t0 points to the array element
        or     $t1,$0,$a2    # Reg. t1 is a counter
loop:    sw     $a3,0($t0)   # Store the value into the array element
        add    $t0,$a1,$t0  # Increment the pointer by the value of size
        addi   $t1,-1       # Decrement the counter
        bgtz   $t1, loop    # branch back to loop if counter >= 0
        # (since we store at the head of the
        # loop, we compute one more address
        # than necessary just to reduce
        # the number of compares & branches)

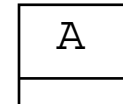
beamup:  jr     $ra         # Beam me up....

```

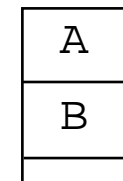
Why a stack is the natural data structure for subprogram calls

```
main(...){
```

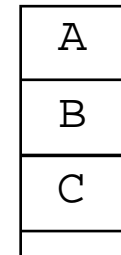
```
  x = A(arg1,...)
```



```
  int A(arg1,...){  
    y = B(arg2,...)
```

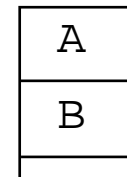


```
    int B(arg2,...){  
      z = C(arg3,...)
```

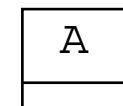


```
      int C(arg3,...){  
        return c; /* in register $2 */  
      }
```

```
    return b;  
  }
```



```
  return a;  
}
```



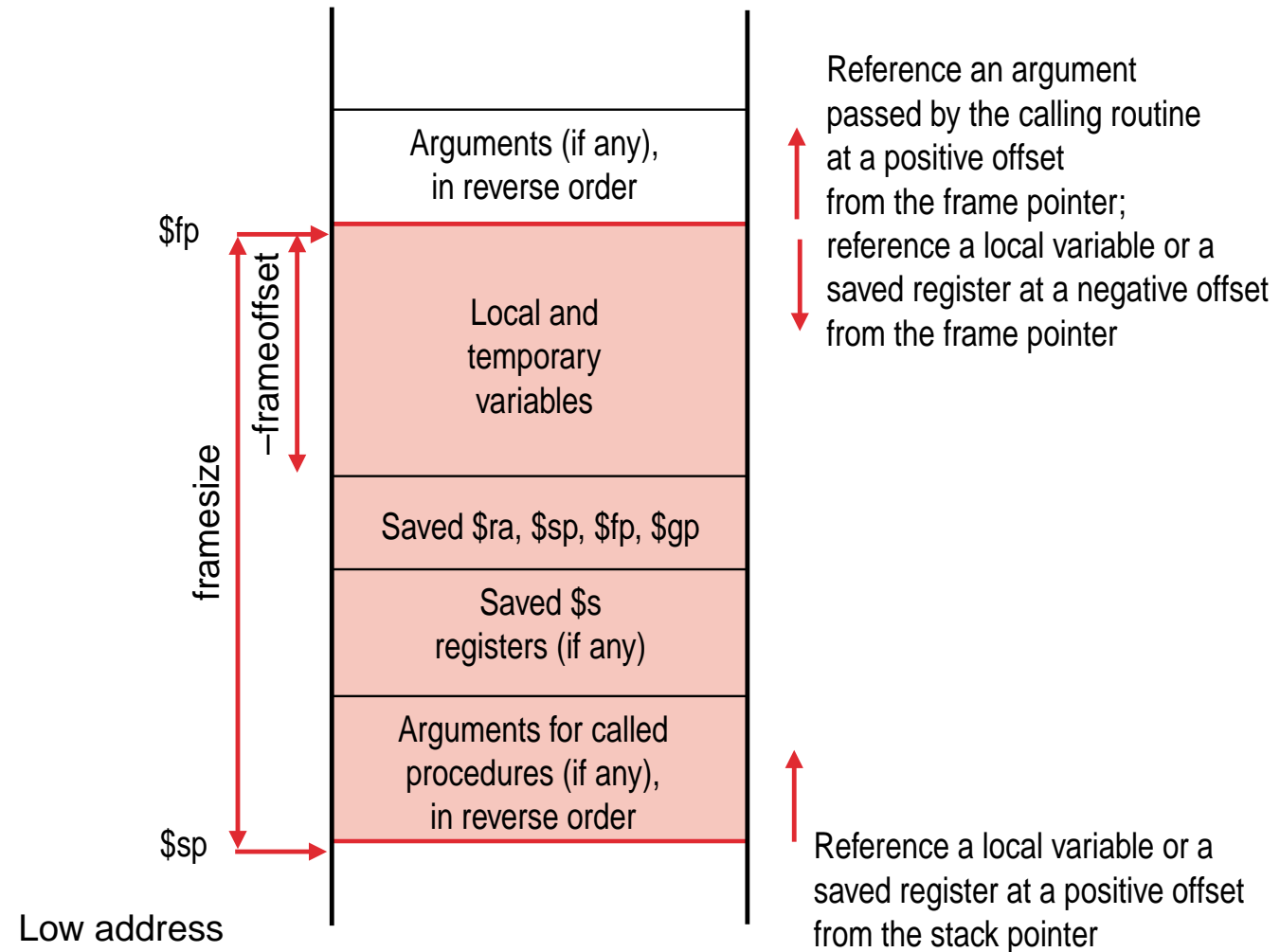
```
}
```

PROCEDURES (7)

- Recommended stack usage in MIPS assembler:
 - ▷ `.frame` pseudo-operation:
`.frame framereg, framesize, returnreg`
 - Must observe convention that $\$sp + framesize = \text{previous } \sp
 - Example:
`.frame $sp, 24, $31`
 - ▷ Adjust stack with instruction
`subu $sp, $sp, framesize`
 - ▷ Save register with instruction
`sw reg, framesize+frameoffset-N($sp)`
 - $N = 0$ for highest-numbered register saved
 - N is incremented by 4 for each subsequent lower-numbered register saved

MIPS STACK USAGE

High address



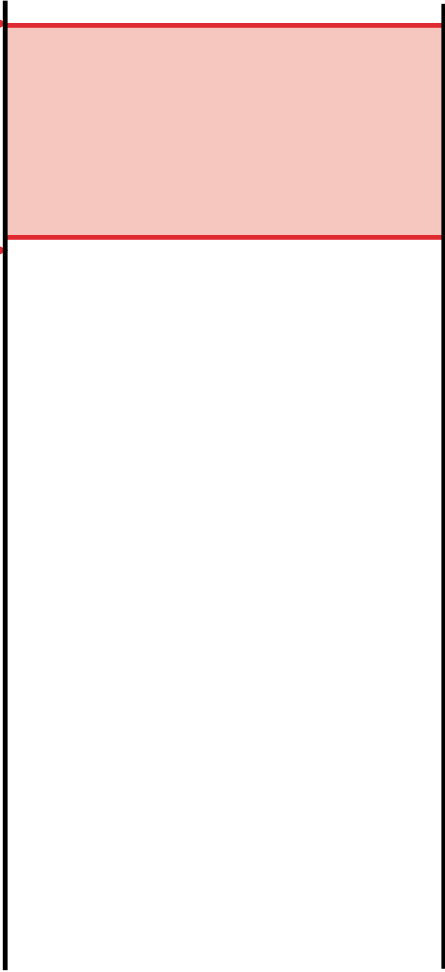
Low address

STACK ALLOCATION

High address

\$fp

\$sp



Before a procedure call

\$fp

\$sp

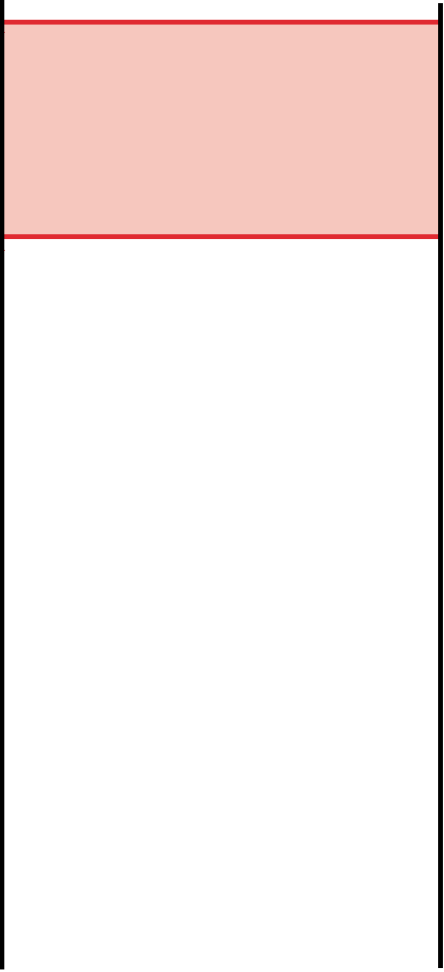
framesize



During a procedure call

\$fp

\$sp



After a procedure call

Low address

FACTORIAL PROGRAM (1)

- Example of tail recursion (recursive computation on the stack)
- Modular design begins by specifying all inputs and outputs
- Recursive algorithm for the factorial function:

▷ Definition: $n! := n(n - 1) \cdots 2 \cdot 1$

(Multiply n by $n - 1$, multiply the result by $n - 2$, ...)

▷ Recursive algorithm:

$$r_k = m_{k-1}r_{k-1}, \quad m_k = m_{k-1} - 1$$

where r_k is the running product and m_k is a counter

▷ Start the algorithm with $r_0 = 1$, $m_0 = n$; then

$$r_1 = n, \quad m_1 = n - 1$$

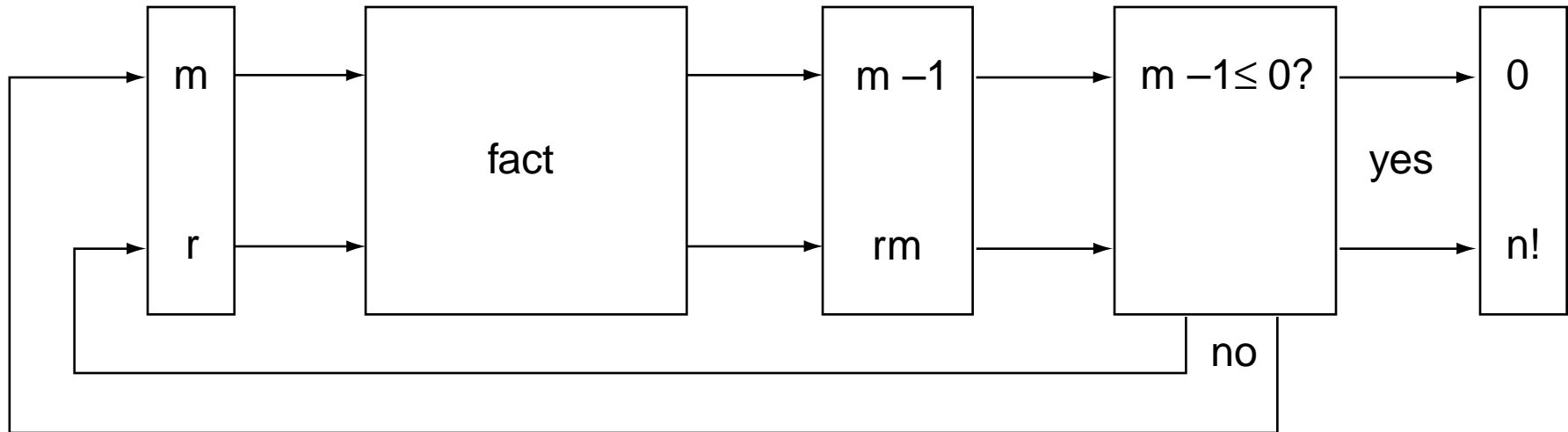
$$r_2 = n(n - 1), \quad m_2 = n - 2$$

⋮

$$r_n = n(n - 1) \cdots 1, \quad m_n = n - n = 0$$

FACTORIAL PROGRAM (2)

- Block diagram of factorial algorithm:



FACTORIAL PROGRAM (3)

- Implement the recursive algorithm for $n!$ as a procedure that calls itself
 - ▷ Only 3 arguments are needed
 - ▷ Arguments could be passed on stack (slow!)
 - ▷ Arguments could be passed in registers (fast; easy to program)
 - ▷ Would be even faster if procedure call overhead were eliminated
- A real-world example in which recursive procedure calls are used:
Traversal of a tree data structure (nested directories, etc.)

```

# fact.s
#
# Computation of the factorial function
# n! := n(n-1)...1
# using the example of stack usage and procedure calls in Goodman and Miller,
# p. 231
#
# This program computes the factorial function of a number n, stores the
# result, and prints the answer on the console. The key routine is called
# fact. This routine functions as a "black box" with 2 inputs (m and r)
# and 2 outputs (m-1 and mr). Suppose that m is initialized to n and that
# r is initialized to 1. Then, if m-1 > 0, the outputs are fed back into
# the inputs. If m-1 <= 0, the mr output is equal to n!
#
# The only tricky part is the use of a recursive procedure, that is,
# a procedure that calls itself. To understand what is happening
# in this program, you MUST single-step through it. Look at the
# registers and the stack as you step.
#
# Design goals and constraints:
# 1. Return the correct value of n! if n >= 0
# 2. Check whether n < 0, and, if yes, print an error message
# 3. Use tail recursion (recursion on the stack)
#
#####
.data
num:      .word 5 # Arg of the factorial function
result:   .word 1 # Will hold num!
string1:  .asciiz " factorial is "
string2:  .asciiz "Error! Argument of factorial is negative"
#####
.text
__start:
        lw $s4,num      # Register 20 gets loaded with the argument
                        # to the factorial function; its value will be used
                        # as a counter
        li $s2,1        # Register 18 will hold the answer
                        # and the running product
        or $v0,$0,$s2   # Copy initial result to $v0 in preparation for the
                        # next instruction
        beqz $s4, OK    # Go to the exit if the argument is 0
                        # Call fact only if the argument is != 0
        jal fact        # Call to the factorial procedure
                        # (register 31 holds the return address)
                        # Note use of a recursive procedure call!

```

```

    or $v1,$0,$v0 # Save result in $v1 so we can use $v0 for syscalls
    bgtz $v0, OK  # Go around the error handling code if
                  # result is OK
err:    ori $v0,$0,4 # Print error message
        la $a0,string2
        syscall
        blez $v1,ttfn # Go to exit code
OK:    sw $v0,result # Save the result
        lw $a0,num # Put num in reg $v0
        ori $v0,$0,1 # Print value of num
        syscall
        ori $v0,$0,4 # Print " factorial is "
        la $a0,string1
        syscall
        or $a0,$0,$v1 # Copy result to reg $a0
        ori $v0,$0,1 # Print value of (num)!
        syscall
ttfn:  ori $v0,$0,10
        syscall # TTFN

fact:  addi $v0,$0,-1 # Entry point for factorial routine
        # A negative result indicates an error;
        # $v0 will stay negative only if it is not
        # overwritten by the instruction before goback
        bltz $s4, goback # Test for invalid (negative) counter
        sw $ra,0($sp) # Push return address onto stack
        add $sp,$sp,-4 # Register 29 points to the first free
        # location
if:    mul $s2,$s2,$s4
        add $s3,$s4,-1 # Subtract 1 from the counter
        blez $s3,endif # and go to endif if <= 0
        move $s4,$s3 # Update the counter
        jal fact # Call fact again
endif: add $sp,$sp,4 # Pop a return address from the stack
        lw $ra,0($sp) # This statement loads the return address
        # that was stored just above the stack
        # pointer into register 31, in preparation
        # for the return
        or $v0,$0,$s2 # Put result in register v0
goback: jr $ra # Go back one step in the recursion,
        # or return to the main routine if this is
        # the return from the first call to fact
#####

```

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

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).

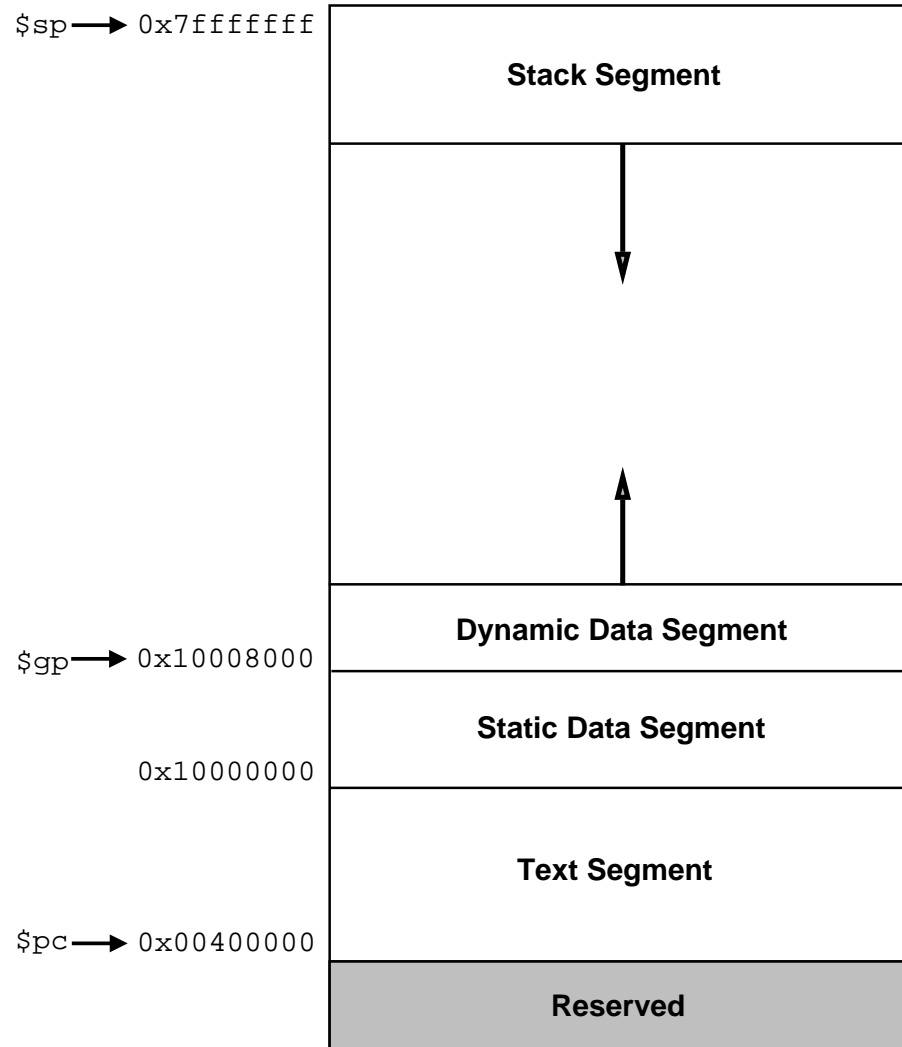
HOW A COMPUTER RUNS PROGRAMS (1)

- 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	0x7fffebe4
[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
...				
[0x7fffeffe0]	0x62696e00	0x484f4d45	0x3d2f686f	0x6d652f74
[0x7fffefff0]	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	l 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
```

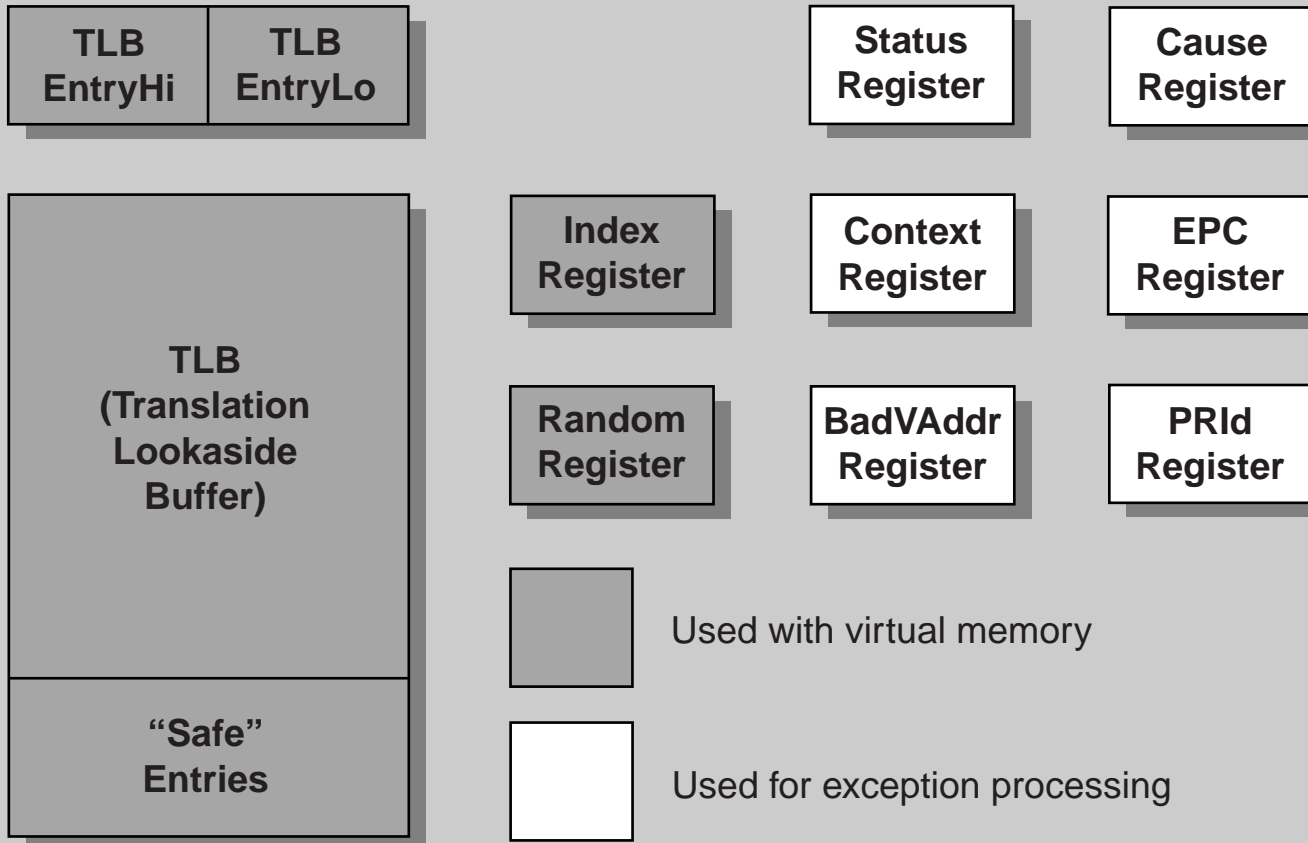
EXCEPTIONS (1)

- An **exception** is an event that causes a process to stop executing
 - ▷ System call (explicit instruction, *e.g.* for I/O)
 - Stopping execution permits another process to execute while the process that made the syscall waits for I/O
 - ▷ Interrupt caused by an event external to the current instruction
 - Example: Hardware controller signals end of I/O
 - ▷ Exception associated with execution of the current instruction
 - Bus error (I/O timeout, load/store kernel physical address)
 - Protection exception
 - Attempt to execute a reserved instruction
 - Cache/TLB miss
 - Floating-point arithmetic exception

EXCEPTIONS (2)

- In the R2000 ISA, exceptions are handled by coprocessor 0
- How the R2000 processor and the UNIX kernel perform **exception handling**:
 1. Processor exits user mode & is forced into kernel mode.
 2. The address of an **exception vector** (exception handling program) is loaded into the program counter (PC).
 - ▷ Reset exception (reboot): the processor transfers control to the Reset exception vector at address `0xbfc00000`
 - ▷ UTLB Miss: Control is transferred to the exception vector pointed to by the contents of address `0x80000000`
 - ▷ All other exceptions are handled by the kernel
 - The general exception handler pointed to by the contents of address `0x80000080` takes control, gets the cause from the Cause register and transfers the correct exception handler

MIPS CP0 and Exception Handling Registers



HOW A COMPUTER BOOTS UP

- Before any processes can execute, the operating system kernel must be loaded into memory
 1. When the processor recognizes the Reset exception, the Reset exception vector loads a short program called the **bootstrap loader** from ROM, EPROM or NVRAM into memory and transfers control to it
 - ▷ In ancient times, the binary machine instructions for the bootstrap loader were toggled in from the system console, or read from cards or paper tape
 2. The bootstrap loader reads another program from a fixed location on disk (called the **boot block**) and transfers control to it
 3. The program read in by the bootstrap loader reads the kernel from disk and transfers control to it

PROCESSES (4)

- In making a **process switch**, the UNIX kernel:
 - ▷ Pushes the register context and system stack onto a **context stack** maintained by the kernel
 - ▷ Uses a scheduling algorithm to find the best process to execute next
 - ▷ Retrieves (pops) the context of the next process from the context stack
 - ▷ Restores the context (registers, stack, mode)
 - ▷ Loads the entry address of the next process into the program counter