1. What are the ELF magic numbers?
   * Elf mag0-3 contains \177, E, L, F
   * Elf class contains class of the file - always Elfclass32. Elfclassnone and Elfclass64 are also available
   * Elf data encoding used - none, elf2lsb, 2's complement lsb first, and elf2msb 2's complement msb first
   * Elf version - indicates version of the elf. Two options EV\_NONE and EV\_CURRENT are available.
   * Elf OSABI - OS/syscall application binary interface (abi) version. Options available here are
   * ELFOSABI\_SYSV - UNIX System V ABI
   * ELFOSABI\_HPUX - HP-UX operating system
   * ELFOSABI\_STANDALONE - Standalone (embedded) application
   * Elf ABI version - This represents the syscall ABI version
   * Elf PAD - Indicates the start of the padding bytes upto EI\_NIDENT
2. What is the difference between UIO\_USERISPACE and UIO\_USERSPACE? When should one use UIO\_SYSSPACE instead?

* Uio\_userispace - program/code space for user process
* uio\_userspace - data space for user process
* uio\_syspace - kernel space for user process.
* You would use kernel space for passing info to other processes

1. Why can the struct uio that is used to read in a segment be allocated on the stack in load\_segment() (i.e., where does the memory read actually go)?

Because it has data that defines the details of the transfer. For example, the data blocks required for the IO, the number of data blocks involved, remaining amount of data to be transferred, whether the transfer is from kernel to user space or vice versa etc. The memory read would go to the offset specified in the uio, all the while updating the uio\_resid (reflecting the amt read) and updating uio\_offset to match.

1. In runprogram(), why is it important to call vfs\_close() before going to usermode?

It is imp to call vfs\_close() because once the program reaches enter\_new\_process() the control never returns and hence the file would never be closed.

1. What function forces the processor to switch into usermode? Is this function machine dependent?

enter\_new\_process() which contains the mips\_usermode is the method that forces to switch to user mode. It is machine dependent as mips\_usermode uses asm\_usermode function

1. In what file are copyin and copyout defined? memmove? Why can't copyin and copyout be implemented as simply as memmove?

kern/vm/copyinout.c defines copyin and copyout functions. memmove is defined in the /common/libc/string/memmov.c defines memmove function. copyin and copyout move data between user/kernel space and performs validations necessary for this. memmove does not such thing.

1. What (briefly) is the purpose of userptr\_t?

It is used to check if the block of user memory falls within the proper userspace region while copying data to/from the kernel space.

1. What is the numerical value of the exception code for a MIPS system call?

Numerical value is 8

1. How many bytes is an instruction in MIPS? (Answer this by reading syscall() carefully, not by looking somewhere else.)

4 bytes

1. Why do you "probably want to change" the implementation of kill\_curthread()?

Because the given implementation does not handle the actual killing of a process. During development of a system call a process may encounter a fatal fault and hence we need to implement kill thread to gracefully exit the user program. Also the development code does not have a active exit system call defined yet.

1. What would be required to implement a system call that took more than 4 arguments?

The first four(32bit)/two(64bit)/two(if 32bit-64bit combo) parameters would be set in the a0-a3 regs. For the rest you pass a pointer to a stack. Which can then be copied using copyin function to get the data from stack (using trap frames stack pointer)

1. What is the purpose of the SYSCALL macro?

It is an interface for all the system calls define in syscall.h. Its definition loads the syscall number into v0 register and then jumps to the share syscall code defined for mips.

1. What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)

syscall is the instruction that makes the actual system call

1. After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: Now that OS/161 supports 64-bit values, lseek() takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg3), and needs to return a 64-bit offset value. In void syscall(struct trapframe \*tf) where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?

32bit file handle in a0 reg. 64bit offset in a2 and a3. 32bit whence value needs to be copyed by using copyin function which would use trap frames stack pointer.

**System Call Implementation**

Structures added to achieve system calls:

1.Struct proc:

* Pid -> which is the pid number a particular process.
* New list structures to hold the list of child processes.
* Semaphore for signaling by sys\_exit to sys\_wait.
* A process field pointing to the parent
* An enum state with 3 states defined (run,zombie,orphan)

To support it, we created a proclist library with the list of process and various methods to access the data structure.

getpid()

Description:

The system call defined here is *int sys\_getpid(int \*retval)*. This a very simple system call and only returns the *pid* associated with a process.

Implementation:

For the implementation the following things are added:

1. *pid\_t pid* in struct proc (proc.h) – This will store the unique pid for a process
2. A pid system which uses the following
   1. *struct lock \*pid\_gen\_lock* – Used to lock resources required to generate pid.
   2. *int32\_t pid\_gen* – A counter that keeps on incrementing. The new pid value generated is validated before assigning to a process. It is checked to see if its already assigned to another process.
   3. *uint32\_t counter* – Used to count the number of pids(processes) in the system. A max, PID\_MAX(32767), number of processes are allowed.
   4. *bool pid\_arr[PID\_MAX]* – Tracks whether a pid is assigned or unassigned in the system.
3. A set of functions
   1. pid\_retrieve – Retrieves the next pid to be assigned
   2. pid\_reclaim – Reclaims an assigned pid when a process is destroyed
   3. pid\_bootrap – Initializes the pid system. That is the counter values, locks etc.
4. *sys\_getpid()* – function that returns the *pid* of the calling process

Pros & Cons:

Using a boolean array to track the pids is time efficient but not space efficient.

fork()

Description:

Fork is going to make a copy of the currently running process along with its thread. We have defined the system call *int sys\_fork(struct trapframe \*tf, int \*retval)*. First the function should copy the trap frame of the invoking process using *kmalloc* and *memcpy*. We then have to copy the current process structure using *proc\_fork*. This will also copy the *filetable* of the parent process. Then the address space of the current process should be copied using the functions *proc\_getas()* and *as\_copy()* functions. We then assign this new copy of address space to the child proc structure copied earlier. Now, current thread is cloned using *thread\_fork* which is passed *enter\_forked\_process* as the entry point for the new thread. *enter\_forked\_process* would be passed the copy of parent trapframe which would then be copied to a trapframe in the userspace. This would complete the setup for the child process and it could continue in user mode. After *thread\_fork* in parent process we set the retval as the child process’ *pid* and return.

Implementation:

For the implementation following things are added:

1. *proclist p\_child* structure in proc.h – Tracks the children of a process.
2. *enter\_forked\_process* helper function – This would be the entry point of new thread. It is here that the trap frame of the parent process is copied and later updated to return 0 to the child process.
3. *sys\_fork()* – Function that does the following
   1. copy trap frame of parent
   2. create and copy process structure for the child
   3. copy the address space of the parent process
   4. create a copy of the current process thread
   5. return value pid of child to parent
   6. return value 0 to child

Pros & Cons:

* 1. Pro : Easier to manage and more powerful than threads. We can (with relative ease compared to thread) manage processes.
  2. Con : Its very expensive to create a process and to maintain it (extra memory requirements etc.)

Sys\_waitpid()

Description: upon calling sys\_waitpid, the process waits for the child of pid p to exit. Upon the child exiting, it will then deallocate the process structure and any main memory held by that process. Every process created has a semaphore associated. We us these semaphores to signal to waitpid that the child has indeed exited. It works so closely in conjunction to sys\_exit, that it can be considered a part of the larger problem, and many of the design decisions (including having advanced list traversal macros and having semaphores) arose from the realization that both of these function needs to work together.

Implementation:

For implementation, the following things are added.

* 1. A sophisticated list structure (proclist and proclistnode) to track all the child processes of the parent. Without it, tracking children for any parent becomes logistical nightmare.
  2. We also had to create a library of functions around the data structure for ease of implementing waitpid.
  3. Sys\_waitpid – a function that does the following
     1. Retrieves the child pid for which it has been asked to wait.
     2. Waits for the process with that pid
     3. Upon the exit of the child, it will return the exit value given by child.

Sys\_Exit()

Description: This method is called every time any user process exits. Of the two, most of the heavy lifting is done by sys\_exit. When a process exits, it traverses through the list of all the child processes it has.   
If the child has exited already (in zombie state), it will cleanup the main memory and structure allocation memory consumed by the process. But if the child is still running, it will remove the child from the list of children and render it orphan (changing state to orphan)

For implementation, following things were added

1. Semaphores, so that the child has a way of signaling if it has exited.
2. Sys\_Exit() function does the following (according to our design)
   1. It traverses through all the children of the process, removing each of them as it traverses, and does one of the following.
      1. If the child is in zombie state, then it will be destroyed
      2. Else it will be orphaned
   2. It will then put itself in zombie state for easy identification of sys\_waitpid
   3. Get the exit code from the recently excited process and attach it to the curp->returnValue
   4. Finally, it needs to call thread\_exit(), as the control cant flow back to userland after exit.

Pro and Cons:

Pro: The approach taken by us saves us a lot of time in managing children of every parent. Even though there was some overhead in making the structure, it paid off big way because list traversal was amongst the most common thing in sys\_waitpid and sys\_exit.

Cons: Our approach doesn’t do much for managing global list of processes.

Alternative approaches:

1. We could have, and in fact wanted to, implement a global process table just like global file table. But the effort for making a fully functioning full-fledged process table was too much for the time considerations of the project.
2. The responsibilities of waitpid and exit could have been exchanges according to the design principals adopted. We felt the present design and sharing of responsibilities make most sense, while was also reasonably easy to implement.