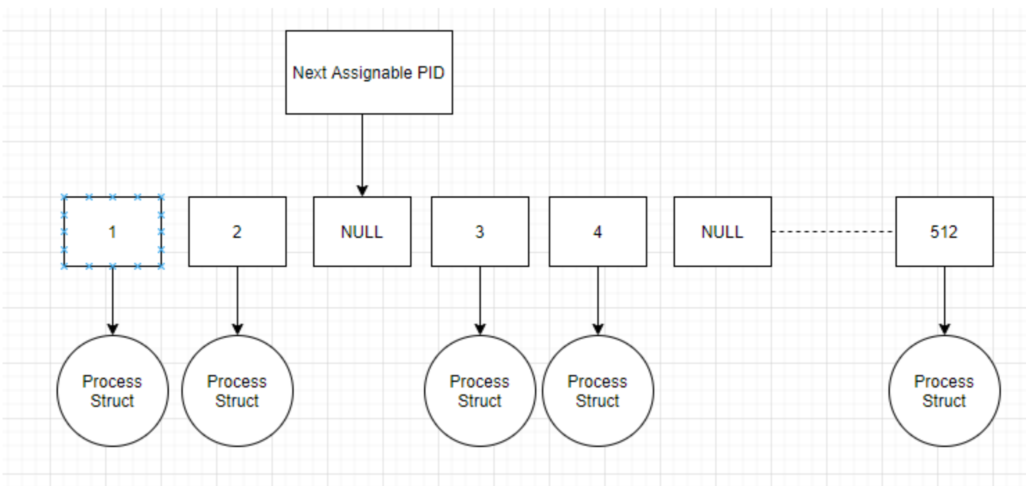
Design Document

This system was designed to use 1 thread per process. It is also assumed that only 1 program can run at a time, all other programs must be forked.

Process structs will be the main form of communication between (forked)programs/other processes. Process_control.c holds all the functions required to create, remove, assign, and grab processes among other things. Getpid() will return the pid stored in the thread struct (assigned by the process and also stored in the process struct). Getppid() will return the parent_pid stored in the process struct that is grabbed using the current thread process struct. New processes will be created in thread_create() in thread.c and this function will be called by thread_fork() which will be called by fork(). Execv's implementation will be very similar to runprogram.c however it will add more checks for MAX_ARGS and will copy over the arguments into a user stack for md_usermode. Waitpid() and exit() will communicate with each other using condition variables.

Process Control (PID)



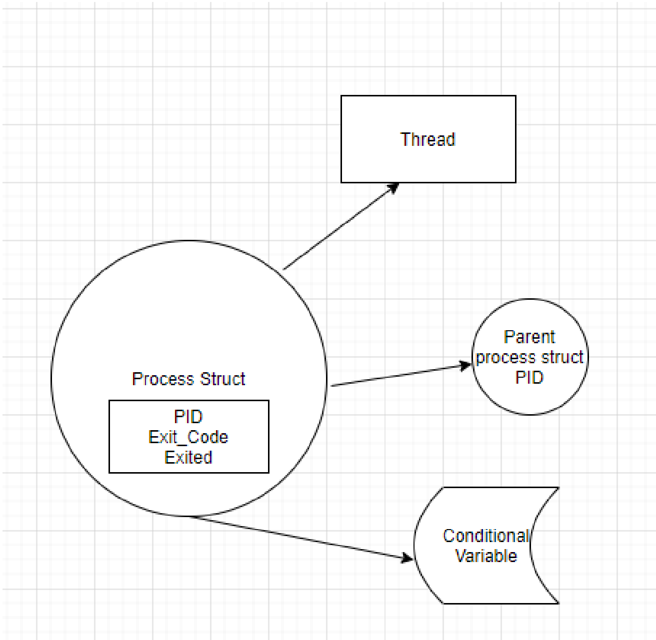
For this project a process struct and process struct array seemed like the best option to go with due to O(1) indexing, easy tracking, and code simplicity.

Process_list – This array is allocated when defined in process_control.c, each index within the process_list is a pid that holds a process struct.

PROCESS_MAX - is defined in process_control.h to be 512 processes. Although the system does reassign PIDs 512 is a good number to have just in case the system decides to run a large program with many forks. This number could be changed at any time.

Process struct – The process struct contains a pointer to the thread of the process. The Pid of the process. A pointer to the parent process pid, an exit code, an exited flag, and a conditional variable struct pointer.

Although the pointer to the thread has no uses at the moment it was kept there as it could be very beneficial in the future when trying to find a thread using the process control. The parent_pid is a pointer to the parent process pid. The exit_code holds exit codes when exiting a process, and the conditional variable is used to wait on child processes, and to signal parent processes.




```
//Process struct
struct process {
     struct thread *t;
     pid_t pid;
     pid_t *parent_pid;
     int exit_code;
     int exited;
     struct cv *exit;
};
```

Process_Control_bootstrap() – the process_control_bootstrap is called in main.c and it is tasked with creating a process lock, checking if the process list has been allocated and creating a process for the first thread.

check_process() – This is a simple function that checks if there are any open PIDs in the process list.

new_process(thread, parent_pid) – This function is called in the thread_create() function in thread.c. The function takes in a pointer to the thread being created and the pid of the parent. New_process() creates a new struct process and conditional variable assigns the parent pid to the process then acquires a lock while it finds the next open PID slot from [0 to PROCESS_MAX] in the process_list. It then assigns the pid to the thread and process, releases the lock and returns the pid. On failure this function returns -1.

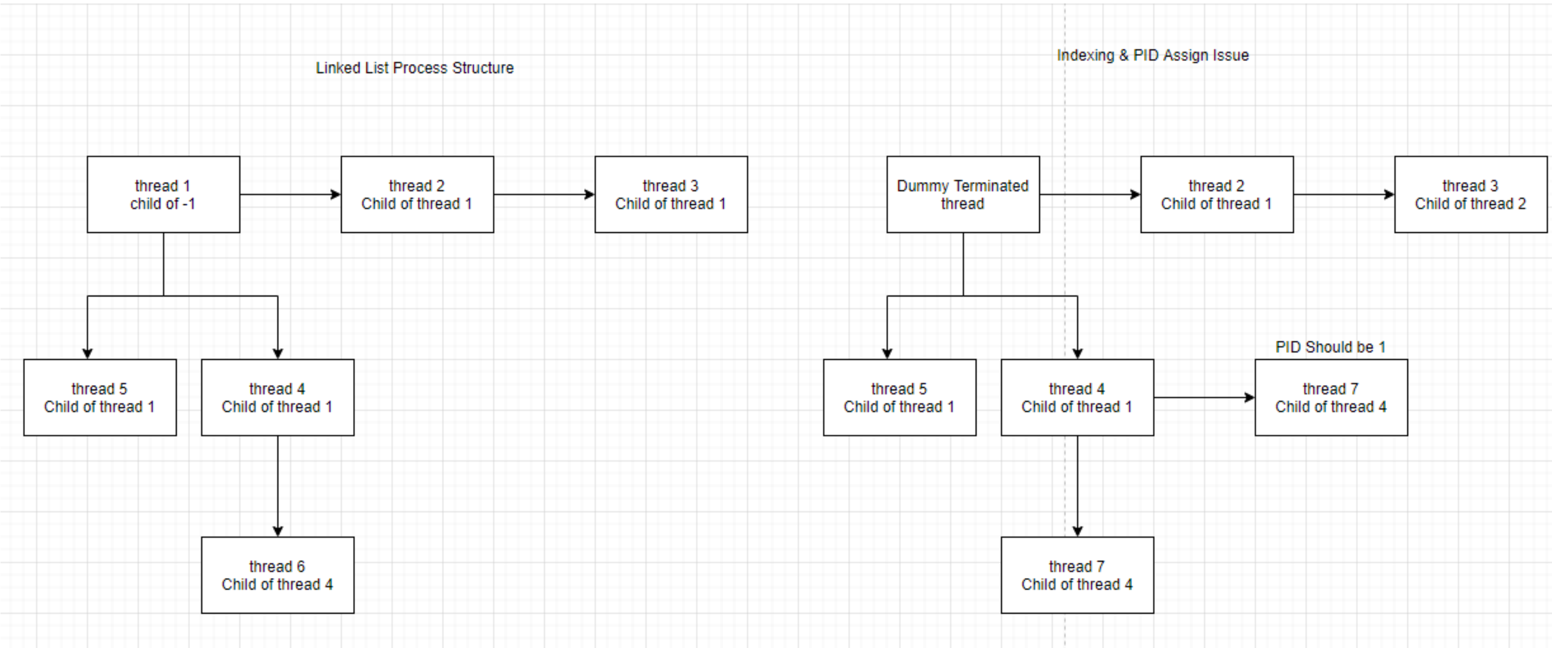
Remove_process(pid) – Remove pid simply calls get_process() with the given pid then destroys that process's conditional variable and frees the process before setting the process_list value of that pid to NULL.

Get_process(pid) – get pid simply returns the process of that pid by indexing it from process_list.

Alternative Approaches:

An alternative approach would have been using linked list. This would have been an easier way of keeping track of parent PIDs since children could branch off the parent. It would've have also been much easier in creating (almost) infinite PIDs without having to reallocate memory to a list if that was our goal. However, this would have made it harder to index, and assign PIDs. An array would have probably been still required to keep track of PIDs or some other system

Alternatively, the process struct values could have all been put in the thread struct, however this would have made things much more complicated when dealing with terminated threads, exit codes, and parent PIDs. Again, some form of array or system would've been required to keep track of PID reassigning.



Sys_getpid

sys_getpid takes in no arguments. It finds the calling process's pid by returning curthread->pid since the pid of each thread is stored in the thread struct on creation using new_process().

Alternative Approaches:

It would've been possible to return the pid using the get_process() function in process_control.c however that would have been redundant since it would be passing in the curthread->pid into that function to find the process. It was simply better to return the pid stored in the thread struct.

Sys_getppid

Sys_getppid takes no arguments. It finds the calling process's parent pid if the parent has not terminated yet. the get_process() function is used with curthread->pid to return the current process, then parent_pid is extracted from its storage in the process struct and get_process() is called again with that parent_pid. There is a check after that to make sure the parent process returned is not NULL (terminated). If it is NULL then -1 is returned, otherwise the pid of the parent is returned from the process struct.

Alternative Approaches:

Alternatively, the parent pid could have been stored in the thread struct like the thread's own pid was stored. However, a process struct made much more sense due to the fact that threads could be terminated before their children, and a system would have to be made to track and update the parent pids.

Sys_fork

Sys_fork begins by calling check_process() to make sure that there is an open PID, if not it returns an error code. Then it initializes a fork_values struct that is defined in syscall.h.

```
struct fork_values {
    struct addrspace *addr;
    struct trapframe *tf;
//  pid_t parent_pid;
};
```

The address space is copied into the pointer using as_copy and the trapframe is copied using copyin, both checking and returning errors, if any.

Thread_fork is then used to initiate the fork with new_forkentry (a function in the same fork.c) being passed through with the fork_values struct. On success sys_fork returns 0 and sets retval to the new threads pid.

In new_fork_entry, the address space is copied once more from the fork_vals struct to the current thread->t_vmspace, and then it is activated. The trapframe values a3, v0, and epc are all adjusted and the trapframe is sent to mips_usermode.

Alternative Approaches:

There are multiple other approaches that could have been taken in regard to sys_fork. The fork entry could have been implemented in syscall.c however it seemed easier to have everything regarding fork in one file for coding purposes.

Another alternative to this implementation could've been when to copy the trapframe and address space. In this implementation the trapframe is copied in sys_fork and the fork_values struct is used to pass those values over to the fork entry function. However, another approach could have been passing the trapframe and address space then copying both in the fork entry function. The current implementation seemed like the easiest approach due to the casting requirements involved in calling thread_fork().

Sys_execv

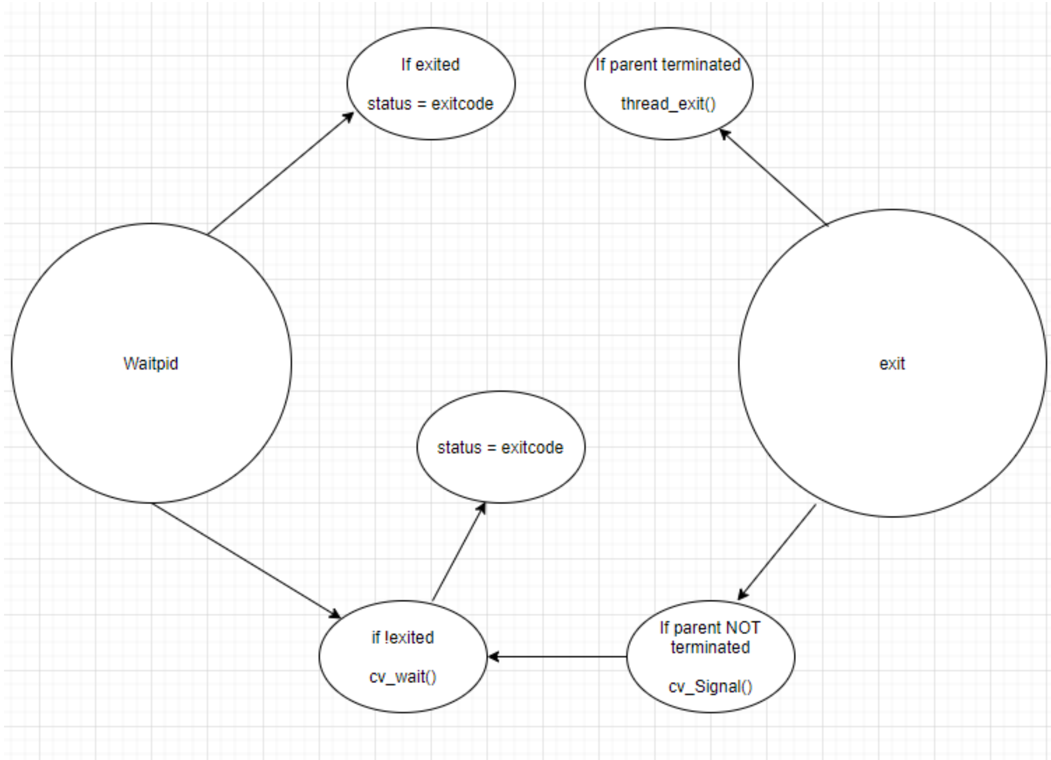
MAX_ARGS – this was set to 32 at the start of the execv.c, this number does not include the program name as it is subtracted at the start of sys_execv after counting the number of arguments.

free_array(array, len) – This function frees an array sent to it.

Sys_execv begins by checking that its arguments are not NULL. Then it checks if the number of arguments is larger than the given MAX_ARGS. It then checks if the number of arguments doesn't equal zero. It then parses the given args into a parse_args array. runprogram.c was copied into execv.c to open the file, set up the address space, and load the program. The arguments are then copied back onto the user stack and md_usermode is called with the number of arguments, the address space, and stack pointers. If the number of arguments is 0, then only the code from runprogram.c runs.

Conditional Variables

Only two conditional variable functions were changed/implemented in synch.c. cv_wait was created to be used in sys_wait, and cv_signal was created to be used in sys_exit. This is how our processes can wait on a process and get their exit code. Each process has its own conditional variable that is created when new_process() is called.



Sys_waitpid

Sys_waitpid starts by checking if there are any issues with its given arguments. It then uses get_process() to get the current thread process and the process it would like to wait on. If the process it is waiting on has already exited (indicated by the exited flag in the process struct) then the status is set to the exit_code and the pid is returned. If the waited upon process did not exit then a lock is acquired and a conditional variable wait is issued. The lock is then released after the parent has been signaled, the status is set to the exit code, and the pid is returned.

Sys_exit

Sys_exit uses get_process() to retrieve the current thread process and it's parent thread process. It then acquires a lock and stores the exit code and flags the process as exited in the current thread process struct. After the lock is released it checks if the parent process has already been terminated. If the parent process has been terminated the thread exits and the process is removed, otherwise it acquires another lock and signals the parent using a conditional variable, then the thread_exits() and the process is removed. The exitcode is still safe even though the thread has exited because it is stored in the process struct and not within the thread.

Sys_write

Sys_write was implemented in syscall.c using the code given in the spec's pdf for this assignment.

Alternative Approaches:

This function could have been written in its own file in userprog, however due to simplicity it was implemented in syscall.c

Kill_curthread

Kill_curthread – calls remove_process() using the curthread->pid to remove the process off the process list. It then calls thread_exit() to exit the thread.