Specifications

Given the requirements for implementing pthreads in xv6, let's begin by defining the thread structure and required system calls

Given these specific requirements, you may want to adjust the `proc_stat` structure as follows:

Use the `ps` system call in your process manager: In your process manager, you can now call the `ps` system call to list all processes. Here's an example of how you might use it in the `list_processes` function:

Sure, let's proceed with the implementation of `pmanager`.

Finally, to initialize `mem_limit` to `0` when a process is created, we modify the `allocproc` function in `proc.c`

based on the specifications given below, write down the changes that need to be done to basic xv6 in form of code. Tell me what changes are added to which files too:

Goal of project

- The project aims to enhance xv6 by adding additional features to its basic structure, specifically threading and process management, which are not provided in the original xv6. The goal is to make xv6 more versatile by incorporating various functions and creating tools for easier management of processes.
- The project involves creating a new system call called "exec2" that allows for allocation of multiple pages for stack and guard usage during the "exec" process. By implementing this feature, users can request and receive the desired number of pages for stack usage instead of the default one page allocation in xv6.

Requirements

- Process with various stack size
  - int exec2(char *path, char **argv, int stacksize);
- Process memory limitation
  - int setmemorylimit(int pid, int limit);
- Process manager
  - List
  - Kill
  - Execute
  - Memlim
  - Exit
- Pthread in xv6
  - int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
  - int thread_join(thread_t thread, void **retval);
  - void thread_exit(void *retval);
  - Other system calls(fork, exec, wait...)

Process with various stack size

- When executing a process in xv6, one page is allocated for the stack and one page is allocated for the guard.

- Create a system call exec2 that allocates the desired number of pages for the stack.
- int exec2(char *path, char **argv, int stacksize);
  • This is a system call that
  allows allocation of multiple pages for stack usage during the "exec" process.
  • The meaning of the first and second arguments is the same as the existing "exec" system call.
- The value of "stacksize" represents the number of pages requested for stack usage.
  • The value of "stacksize" should be an integer between 1 and 100 inclusive.
  • Regardless of the size, at least one page must be allocated for guard usage, and it should be located immediately below the stack pages in virtual memory.

Memory Limitation for Processes

- This feature limits the maximum amount of memory that can be allocated for a specific process.
- When a user process requests additional memory, it should not be allocated if it exceeds the memory limit of that process.
- When a process is initially created, there is no limit, but it can be set through a system call afterwards.
- int setmemorylimit(int pid, int limit);
  • "pid" specifies the process ID for which the memory limit will be set.
  • "limit" represents the maximum amount of memory, in bytes, that the process can use.
  • The value of "limit" must be a non-negative integer. If it is positive, the process will have a memory limit equal to that value.
  If it is 0, there is no limit.
  • If the specified "limit" is smaller than the memory currently allocated to the process, the system call will return -1.
  • If the specified "pid" does not exist or the "limit" is negative, the system call will also return -1.
  • If the operation is successful, the system call will return 0.

Process Manager

- This feature creates a user program that can check and manage information about currently running processes.
- Once executed, the program will prompt the user to enter commands until a "quit" command is given.
- The name of the program is "pmanager".
- The pmanager should support the following commands:

1. list
   • Displays
   information about currently running processes.
   • Includes the
   name, pid, number of stack pages used, allocated memory size, and maximum memory limit for each process.
   • The program
   should consider the information of threads executed by the process.
   • Thread information should not be included.
   • The system call used to obtain process information can be freely defined.
2. kill <pid>
   • Kills the process with the given pid using the kill system call.
   • Prints the success status.
3. execute <path> <stacksize>
   • Runs the program located at path with stacksize number of stack pages.
   • Passes a single argument to the program with path as the 0th argument.
   • pmanager should continue to run without waiting for the program to finish executing.
   • Prints an error message only if execution fails.
4. memlim <pid> <limit>
   • Sets the memory limit for the process with the given pid to the given limit.
   • The value of limit is a non-negative integer. If it is positive, the process has a memory limit of that size. If it is zero, there is no memory limit.

    - The memory of threads executed by the process should be considered.
    - Prints the success status.
5. exit
    - Exits the pmanager.

- The input format for each command must follow these rules:
  - All inputs consist only of alphabets, numbers, spaces, and newlines.
  - There are no leading or trailing spaces in any input.
  - There is exactly one space between a command and an option, and between options.
  - The values for pid, stacksize, and limit are integers between 0 and 1 billion.
  - The length of path is no longer than 50 characters and consists only of alphabets, numbers, and underscores.
- It is recommended to allocate enough space in the arrays to accommodate null characters and other special characters.
- Each command must follow the specified format.
- Any commands not specified in the specifications will not be executed.

Light Weight Process

- Processes are executed independently of each other, do not share resources, and have separate address spaces and file descriptors.
- Pthread is an API primarily used to develop parallel programs in Unix-like operating systems and is one of the representative implementations of LWP.
- Light-weight process (LWP) is a concept that allows multitasking at the user level by sharing resources, address spaces, and other features among different LWPs.
- xv6 implements only the Process feature, and the goal is to implement LWP that performs the Thread function.

Specification

API (thread_create)

1. int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg);
   • Creates and starts a new thread.
   • thread: Specifies the id of the thread.
   • start_routine: Specifies the function that the thread will start from.
   The
   new thread starts from the function pointed by start_routine.
   • arg: Arguments to be passed to the start_routine of the thread.
   • return: Returns 0 if the thread is successfully created, -1 if there is an error.

API (thread_exit)

1. void thread_exit(void *retval);
   • Terminates the thread and returns a value.
   • All threads must terminate using this function, and it does not consider the case where the thread reaches the end of the start function.
   • retval: The value that will be received from the join function after terminating the thread.

API (thread_join)

1. int thread_join(thread_t thread, void **retval);
   • Wait for the termination of the specified thread and retrieve the value returned by thread_exit().
   • If the thread has already terminated, it returns immediately.
   • After the thread has terminated, the resources allocated to the thread need to be released and cleaned up.
   (page tables, memory, stack, etc.)
   • thread: ID of the thread to be joined.
   • retval: Pointer to a variable to store the value returned by the thread.
   • return: Returns 0 if successful, or -1 otherwise.

System Call

- In order for threads to function properly as a single process in xv6, they must be supported by several system calls.
- The following system calls are evaluated to determine if threads work well, even if multiple threads call system calls at the same time.

1. fork:
   • When fork is called in a thread, the existing fork routine should be executed without any problems.
   • The contents of the address space of the corresponding thread should be copied, and a new process should be able to start.
   • It should also be possible to wait using the wait system call.
2. exec:
   • When exec is executed, all threads of the existing process must be cleaned up.
   • One thread starts a new process and the other threads should be terminated.
3. sbrk: sbrk is a system call that allocates memory to a process.
   • Even if multiple threads request memory allocation at the same time, the space provided for allocation should not overlap, and it should be possible to allocate memory properly.
   • Memory allocated by sbrk can be shared by all threads in the process.
4. kill: kill is a system call that terminates a process.
   • If one or more threads are killed, all threads in the process must be terminated.
   • The resources of killed and terminated threads must be cleaned up and recovered.
5. sleep: sleep is a system call that puts a process to sleep for a certain amount of time.
   • If one thread calls sleep, only that thread should sleep.
   • Even while sleeping, it should be possible to be terminated by kill.
6. pipe:
   • There should be no problem with each thread outputting to the screen separately.

Tips

- xv6 uses the default scheduler.
  • If modifications are needed in the scheduler during Project 2, you can modify the default code.
  • Like other processes, threads are also scheduled using the RR by the scheduler.
- If you are unsure about how to fix a certain part, refer to the xv6 code.
  •
  Analyzing the implementation details of fork, exec, exit, wait, etc. will help you with the project.
- You can work on the project even after cloning the git repository.
- Process management is closely related to processes.
  • Please refer to related files such as proc.c and proc.h.
  • For the BACK command handling in the runcmd function in sh.c, please refer to it.
- The child process created through the fork system call copies the memory limit and stack size that the parent process had at that time.
- We recommend using locks appropriately to avoid race conditions.
  • When working on thread-related tasks, we recommend considering a lock for the ptable.
- LWP is treated similar to processes in xv6.
  That is, except for some features specified in the assignment, it behaves like a process.
  • Threads are similar to processes.
  • However, all threads share the pid of the main thread.
- By sharing the process's page table, threads can share address space.
- For proper implementation and operation of threads, please refer to the implementation and operation of pthreads in Linux.

Evaluation

- Code
  • All requirements specified in the specification must be implemented

correctly.
  • Comments must be included in the code.

- Wiki
  • Will be evaluated based on the test program and the wiki.
  • The wiki should be detailed about the design and include scenes of actual operation.
- Submission
  • The deadline must be strictly observed.
  • The last commit/push to GitLab before the deadline will be used for evaluation.

Wiki

- Design
  • Describe how you plan to implement the requirements specified in the specification, what data structures and algorithms are needed, and your own design.
- Implement
  • Provide a detailed description of what is different from the existing code and what the purpose of the code is for the parts you have newly implemented or modified.
- Result
  • Attach the compilation and execution process and the results of the execution of the parts required in the specification that are functioning properly, and describe the operation process.
- Trouble shooting
  • Describe the problems encountered during the assignment and the process of resolving them. If the problem could not be solved, describe what the problem was and how you tried to solve it.

Extra

- It is okay to create a new directory in GitLab. (e.g. project02)

- You can also save the previous Project 1 content in a separate directory. (e.g. project01)

1. **Process with various stack size**
   - Changes:
     - Create a new system call called `exec2` that allows allocation of multiple pages for the stack during the `exec` process.
   - Files to modify:
     - `syscall.h`: Add a new system call number for `exec2`.
     - `syscall.c`: Add the implementation of `exec2` system call.
     - `user.h`: Declare the `exec2` function.
     - `usys.S`: Add the system call stub for `exec2`.
2. **Memory Limitation for Processes**
   - Changes:
     - Create a new system call called `setmemorylimit` that sets the memory limit for a process.
   - Files to modify:
     - `syscall.h`: Add a new system call number for `setmemorylimit`.
     - `syscall.c`: Add the implementation of `setmemorylimit` system call.
     - `user.h`: Declare the `setmemorylimit` function.
     - `usys.S`: Add the system call stub for `setmemorylimit`.
     - `proc.c`: Add the necessary logic to handle memory limits for processes.
3. **Process Manager**
   - Changes:
     - Create a user program called `pmanager` that manages information about running processes.

- Files to modify:
  - `Makefile`: Add the `pmanager` program to the build process.
  - `pmanager.c`: Implement the `pmanager` program according to the specified commands and functionality.

4. **Light Weight Process (LWP)**
   - Changes:
     - Implement the thread-related functions and system calls (`thread_create`, `thread_exit`, `thread_join`).
     - Ensure that the existing system calls (`fork`, `exec`, `wait`, etc.) handle threads correctly.
   - Files to modify:
     - `user.h`: Declare the thread-related functions.
     - `syscall.h`: Add new system call numbers for the thread-related system calls.
     - `syscall.c`: Implement the thread-related system calls.
     - `usys.S`: Add the system call stubs for the thread-related system calls.
     - `proc.c`: Modify the existing system calls to handle threads correctly.
     - `thread.c`: Implement the thread-related functions and their underlying mechanisms.
     - `thread.h`: Define data structures and functions related to threads.