Project 02

Implementing a simple kernel-level thread

Due date
2025. 05. 25. 23:59

Overview

- This project involves implementing a simplified version of kernel-level threads in the xv6 operating system.
- Kernel-level threads enable a process to have multiple execution contexts simultaneously, with each thread maintaining its own context (register state, stack) while sharing the process's resources (address space, file descriptors).

Key Implementation Requirements

- A new system call **clone()** to create a new kernel-level thread
- A new system call **join()** to wait for a kernel-level thread to terminate
- User space library functions **thread_create()** and **thread_join()** that wrap these system calls
- Modify related functions in kernel/proc.c to make kernel-level threads work properly

Special Characteristics

These xv6 kernel-level threads differ from traditional ones in several ways:

- Each thread has its own file descriptor table (copied during clone)
- If the main thread calls exit(), the entire process terminates (including all threads)
- If any other thread calls exit(), only that thread terminates
- You can reuse the `struct proc` as the thread control block (TCB)

API Details: System calls

int clone(void(*fcn)(void*, void*), void *arg1, void *arg2, void *stack);

- Creates a new kernel-level thread sharing the calling process's address space
- fcn: Function where the new thread starts execution
- arg1, arg2: Arguments passed to the thread function
- stack: Thread's userspace stack (must be page-aligned and at least 1 page in size)
- Returns the thread's ID (pid) on success, -1 on failure

API Details: System calls

int join(void **stack);

- Waits for a child thread to terminate
- stack: Address of a void* variable where the thread's stack address will be copied
- Returns the PID of the terminated thread or -1 on failure

API Details: User Library Functions

- Must be implemented in user/thread.h and user/thread.c
  - Add thread.o to `ULIB` for thread library support in Makefile
- **int thread_create(void (*start_routine)(void *, void *), void *arg1, void *arg2);**
  - Allocates memory for the thread's stack and calls clone()
  - Returns the PID of the new thread or -1 on error
- **int thread_join();**
  - Calls join() to wait for a thread to terminate and frees the thread's stack
  - Returns the PID of the terminated thread or -1 on error

System Call Modifications

Ensure these system calls work properly with threads:

- **fork**: Thread should be able to call fork normally, copying its address space
- **exec**: Should clean up all threads and start a new process
- **sbrk**: Multiple threads should be able to allocate memory safely
- **kill**: Terminating one thread should terminate all threads in the process
- **sleep**: Should only affect the calling thread
- **pipe**: All threads should be able to output to the screen

Tips for Development

- Use the default xv6 scheduler (round-robin scheduling)
- Study xv6 code carefully, especially **fork, exec, exit,** and **wait** implementations
- Use appropriate locks to prevent race conditions
- Ensure proper resource management to avoid memory leaks
- **Clearly understand the differences between threads and processes.**
  - Carefully analyze which resources are shared and which are independent.
  - The base code provided for this project already includes necessary additions and modifications.
  - Document in the wiki what has been changed and why these changes were made.

Evaluation

- **Completeness** The xv6 operating system must function correctly according to the specification requirements.
- **Wiki & Comment** Grading will be based on the wiki documentation, so the wiki should be written in as much detail as possible.
- **Deadline** The submission deadline must be strictly observed. After the deadline, your GitHub writing permissions will be revoked.
- **DO NOT SHARE AND COPY!!**

Wiki

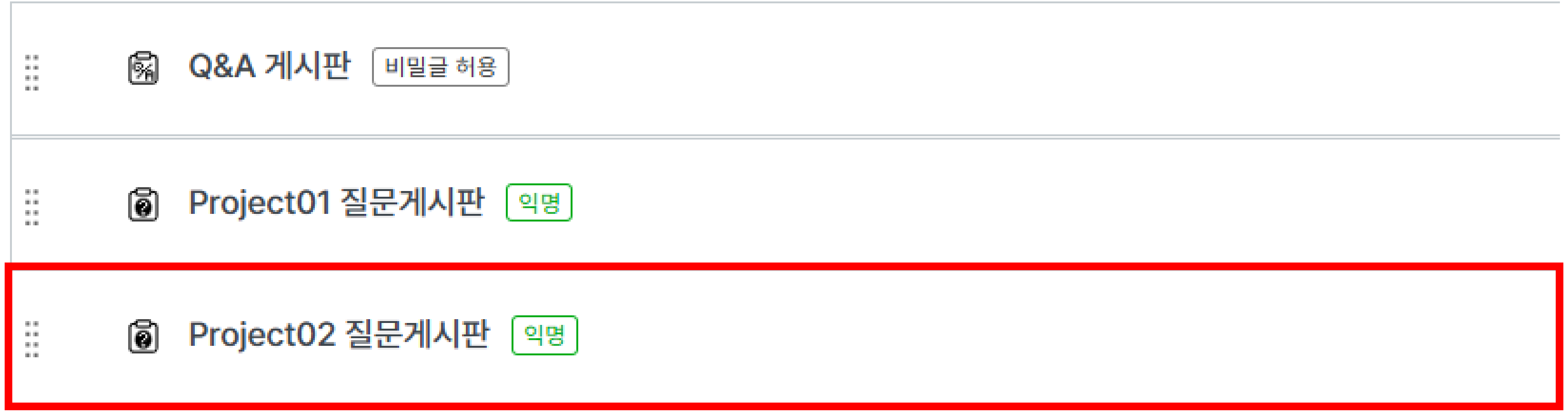
- **Design** Outline your implementation approach for meeting the project requirements
- **Implementation** Explain key code modifications and their purpose, focusing on changes from the original code.
- **Results** Show evidence of successful implementation with:
  Compilation process, Screenshots of working code, Explanation of program flow
- **Troubleshooting** Describe any problems encountered, solutions applied, and any unresolved issues.
- Additional content may be included if relevant.

Submission

- Submit your implemented code and wiki through GitHub.
  - **Refer to the announcement and create a new repository.**
  - Rename the repository to “**project02-[student ID]**”
- The wiki file should be named "**OS_project02_[class number]_[student ID].pdf**".
- Submission deadline: **May 25, 2025, 23:59**
  - Late submissions will be accepted via **email** until **May 26, 2025, 23:59**, but will only receive **50%** of the possible score.

Q&A

- For questions related to the project, please use the question board (Project 02 Question Board) on the LMS.
- Questions sent by email will **not** be answered.
- For questions not related to the project, please use the Q&A board or send an email.



Q&A 게시판 비밀글 허용

Project01 질문게시판 익명

Project02 질문게시판 익명

Q & A