Indian Institute of Technology Bhilai
Department of Computer Science
CSL301: Operating Systems
Scope: xv6
Difficulty Level: Moderate

Take Home Assignment - 2
August 28, 2025

Instructions -

- Follow the steps given in the question and update all required files
- Create a Word (.doc/docx) file with the following (Use LibreOffice in Ubuntu) in step-by-step format as mentioned in the question. Provide only the code that you have added and the file name under which the update has been made.
- Save your final shell source code with the following naming convention: `<rollnumber>_sh.c`
- Do not include full files, just the specific changes made. Save this file using: `<rollnumber>_part.pdf`
- Submit these files as follows: Place all the files above in a single folder and compress it. Name the compressed archive as `HA2_<rollnumber>_part.zip`

Note: Do not include explanations or code outside the specified text file and annotated source file.
Each screenshot should clearly show your QEMU terminal running and reflecting the update for each task.

Objective

- Implement a new system call `pinfo(int pid, struct proc_info *info)` in xv6.
- Retrieve process information including PID, name, state, and memory size.
- Practice modifying both kernel and user-space code.
- Test your implementation using multiple processes generated by a helper program.

Background and Hints

Each process in xv6 is represented by a `struct proc` in `kernel/proc.h`. Relevant fields include:

- `pid` – the process ID
- `name` – name of the process (string)
- `state` – process state (`UNUSED, USED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE`)
- `sz` – memory size of the process

Safe Locking in xv6

In xv6, the process table (`ptable.proc[]`) is a shared resource accessed by multiple processes. To prevent race conditions, **safe locking** must be used with `ptable.lock`.

Note: Safe locking is a core concept in concurrency and will be taught in your concurrency class. This assignment gives a practical example of using locks in the kernel.

1. Files that require `spinlock.h`**:**

- `proc.c` – Required to acquire/release `ptable.lock` in `get_proc_info()`.
- `sysproc.c` – Required if accessing `ptable` or calling functions that acquire locks.
- `proc.h` – Optional; include only if declaring functions or structures that reference locks.
- Other kernel files accessing `ptable` (e.g., `fork.c`, `trap.c`) – include `spinlock.h` if you acquire or release a lock.
- User programs (`pinfo.c`, `testproc.c`) do NOT include `spinlock.h`.

2. How to implement safe locking:

1. Include the spinlock header in kernel files:

   ```
   #include "spinlock.h"
   #include "proc.h"
   ```

2. Acquire the lock before accessing the process table:

   ```
   acquire(&ptable.lock);
   ```

3. Access or modify the shared data (e.g., read PID, name, state, memory size).

4. Release the lock after finishing, even if returning due to an error:

   ```
   release(&ptable.lock);
   ```

Example in `get_proc_info`:

```
int
get_proc_info_kernel(int pid, struct proc_info *info)
{
        struct proc *p;

        for(p = proc; p < &proc[NPROC]; p++){
                acquire(&p->lock);
                if(p->pid == pid){
                        info->pid = p->pid;
                        safestrcpy(info->name, p->name, sizeof(info->name));

                        // Convert enum state to string
                        char *st = "UNKNOWN";
                        switch(p->state){
                                case UNUSED:   st = "UNUSED";   break;
                                case USED:     st = "USED";     break;
                                case SLEEPING: st = "SLEEPING"; break;
                                case RUNNABLE: st = "RUNNABLE"; break;
                                case RUNNING:  st = "RUNNING";  break;
                                case ZOMBIE:   st = "ZOMBIE";   break;
                        }
                        safestrcpy(info->state, st, sizeof(info->state));

                        info->sz = p->sz;

                        release(&p->lock);
                        return 0;
                }
                release(&p->lock);
        }
        return -1;
}
```

Key Points:

- Always acquire the lock before reading/modifying shared data.

- Always release the lock, even on errors.
- Use safestrcpy for kernel string copying.
- Only kernel files require spinlock.h, not user programs.
- Understanding locks is part of your concurrency course; this assignment gives a hands-on example.

Kernel-side Modifications:

1. Create procinfo.h in both kernel/ and user/. Define struct proc_info with fields:
   - int pid;
   - char name[16];
   - char state[16];
   - uint64 sz;
2. Implement get_proc_info(int pid, struct proc_info *info) in proc.c.
3. Add syscall number in syscall.h, implement sys_get_proc_info() in sysproc.c, and wire it in syscall.c.

User Program: pinfo.c

The user program should:

- Accept a PID as a command-line argument.
- Call get_proc_info(pid, &info).
- Display: PID, Name, State, and Memory Size.
- Handle invalid PIDs gracefully.

Testing with testproc.c

A helper program testproc.c should be created to fork multiple processes for testing.

```
#include "user/user.h"

int main(void) {
        int i;
        int num_children = 5;

        for(i = 0; i < num_children; i++) {
                int pid = fork();
                if(pid < 0) {
                        printf("Fork␣failed\n");
                        exit(1);
                }
                if(pid == 0) {
                        printf(‘‘Child process %d started with PID %d\n’’, i+1, getpid());
                        while(1);   // never exits
                }

        }

        // Parent waits for all children to finish
        //   for(i = 0; i < num_children; i++) {
                //        int wpid = wait(0);
                //       printf("Parent: child PID %d finished\n", wpid);
                //    }

        exit(0);
}
```

Instructions

1. Place testproc.c in user/.
2. Add _testproc to UPROGS in the Makefile.
3. Compile xv6 using make qemu and first run testproc in xv6 and then press enter after that run pinfo.
4. Run the program inside the xv6 shell:

```
$ testproc
$ pinfo 3
$ pinfo 4
$ pinfo 5
```

5. Observe how pinfo displays the information for multiple processes.

Expected Output and Testing Guidelines

```
PID: 2
Name: sh
State: RUNNING
Size: 4096
```

Testing tips:

- Ensure testproc is running so children exist when checking with pinfo.
- Check multiple PIDs including invalid ones to confirm proper error handling.
- Verify that state strings correctly match the process enum (RUNNING, SLEEPING, etc.).

Submission and Evaluation

Deliverables:

- Modified files: proc.c, proc.h, sysproc.c, syscall.c, syscall.h, pinfo.c.
- Screenshot of pinfo output for at least 2 processes.

Evaluation Criteria:

- Correctness of implementation
- Proper state conversion from enum to string
- Safe locking
- Robust error handling
- Clean, readable code