Week5 Report in Class (Fri56)

> 11911839 聂雨荷

Q1

> 代码中如何区分父子进程?父子进程的执行顺序是否是固定的?

- We can use the return value of `fork()` function to determine which of process is the parent and which is the child, on success if
  - `fork() == 0`, current process is the child process
  - `folk() == childPID`, current process is the parent process
- The execution order of parent-child processes is not fixed

Q2

> 请回答第四步僵尸进程中列举的第4种情况的结果会是什么。

- If the child process does not exit but the parent process does exit, it will become orphan process (孤儿进程).
- The child process is adopted to the `init` process (PID=1) or to the registered grandfather process.

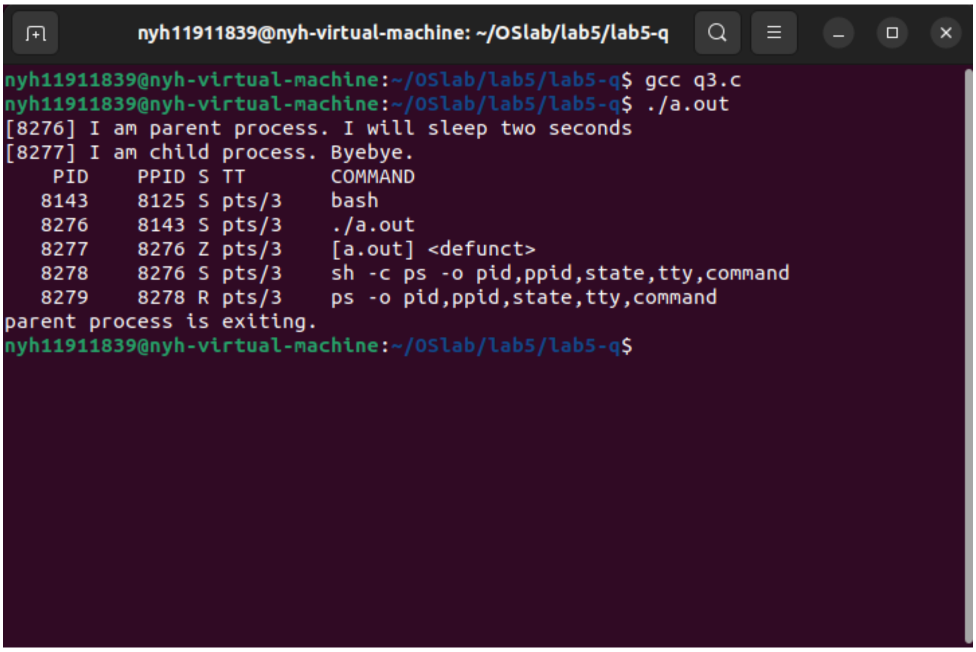
Q3

> 请编写一段c语言代码 (截图)，用于产生僵尸进程，并截图僵尸进程的状态 (ps).

```c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

int main()
{
    pid_t pid = fork();
    if (pid < 0)
    {
        perror("fork error:");
        exit(1);
    }
    else if (pid == 0)
```

```c
    {
        printf("[%d] I am child process. Byebye.\n", getpid());
        exit(0);
    }
    printf("[%d] I am parent process. I will sleep two seconds\n", getpid());
    sleep(2);
    system("ps -o pid,ppid,state,tty,command");
    printf("parent process is exiting.\n");
    return 0;
}
```



Q4

> lab5 的 ucore 代码具体通过哪条指令以什么形式跳转至 init_main()

- `init_main` 不断调用 `cpu_idle()` 函数，进而调用到 `schedule()` 函数，`schedule` 会找到允许调度的进程
- 我们会在这中间调用 `forkets` 函数将 `init_main` 进程的中断帧放在 `sp`，从中恢复所有的寄存器，通过 `kernel_thread_entry`，我们将 `s0` 寄存器存放着新进程执行的函数，`s1` 存放着传给函数的参数，我们把参数放在 `a0` 寄存器，并跳转到 `s0` 执行 `init_main`，从而完成调用

```asm
.text
.globl kernel_thread_entry
kernel_thread_entry:
    move a0, s1
    jalr s0  # jump to init_main() function
    jal do_exit
```

Q5

> lab5 的 ucore 代码中是如何调用到 kernel_thread_entry 的?

1. initial `init_main` process, idle process
2. call `kernel_thread` function to create a new process `init_main`, and it will add this process to the process linked list,below 3-5 explain the details
3. `kernel_thread` will call `do_fork` function to initial a new PCB

```c
// kernel_thread - create a kernel thread using "fn" function
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));

    tf.gpr.s0 = (uintptr_t)fn;
    tf.gpr.s1 = (uintptr_t)arg;
    tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;
    tf.epc = (uintptr_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

4. `do_fork` will call the `copy_thread` function to copy the current process, besides, it will add the new process to the process linked list

```c
/* do_fork -     parent process for a new child process
 * @clone_flags: used to guide how to clone the child process
 * @stack:       the parent's user stack pointer. if stack==0, It means to fork a kernel thread.
 * @tf:          the trapframe info, which will be copied to child process's proc->tf
 */
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;
    //    1. call alloc_proc to allocate a proc_struct
    //    2. call setup_kstack to allocate a kernel stack for child process
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    4. call copy_thread to setup tf & context in proc_struct
```

```
    //    5. insert proc_struct into hash_list && proc_list
    //    6. call wakeup_proc to make the new child process RUNNABLE
    //    7. set ret vaule using child proc's pid
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    if ((ret = setup_kstack(proc)) == -E_NO_MEM) {
        goto bad_fork_cleanup_proc;
    }
    copy_mm(clone_flags, proc);

    copy_thread(proc, stack, tf);

    const int pid = get_pid();
    proc->pid = pid;
    list_add(hash_list + pid_hashfn(pid), &(proc->hash_link));
    list_add(&proc_list, &(proc->list_link));
    nr_process++;

    wakeup_proc(proc);
    ret = pid;
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}
```

5. `copy_thread` will copy and switch a new process, besides, it sets the `ra` to be the entry of `forkret` function

```
// copy_thread - setup the trapframe on the  process's kernel stack top and
//             - setup the kernel entry point and stack of process
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct trapframe));
    *(proc->tf) = *tf;

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;

    proc->context.ra = (uintptr_t)forkret;
    proc->context.sp = (uintptr_t)(proc->tf);
```

```
}
```

6. since `idle_proc` need reschedule, it will call `schedule` function to check whether there is a process schedulable

```
void cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
        schedule();
        }
    }
}

```

7. `schedule` will call `proc_run` function to wake up selected process

```
void
schedule(void) {
    bool intr_flag;
    list_entry_t *le, *last;
    struct proc_struct *next = NULL;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        last = (current == idleproc) ? &proc_list : &(current->list_link);
        le = last;
        do {
            if ((le = list_next(le)) != &proc_list) {
                next = le2proc(le, list_link);
                if (next->state == PROC_RUNNABLE) {
                    break;
                }
            }
        } while (le != last);
        if (next == NULL || next->state != PROC_RUNNABLE) {
            next = idleproc;
        }
        next->runs ++;
        if (next != current) {
            proc_run(next);
        }
    }
    local_intr_restore(intr_flag);
}
```

8. `proc_run` will call `switch_to` to switch new process

```
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

9. `switch_to` function save and swap registers that need to be saved, `ra` register is set to be the entry of `forkret` function, so it's going to return the `forkret` function

```
#include <riscv.h>

.text
# void switch_to(struct proc_struct* from, struct proc_struct* to)
.globl switch_to
switch_to:
    # save from's registers
    STORE ra, 0*REGBYTES(a0) # here proc->context.ra = (uintptr_t)forkret;
    STORE sp, 1*REGBYTES(a0)
    STORE s0, 2*REGBYTES(a0)
    STORE s1, 3*REGBYTES(a0)
    STORE s2, 4*REGBYTES(a0)
    STORE s3, 5*REGBYTES(a0)
    STORE s4, 6*REGBYTES(a0)
    STORE s5, 7*REGBYTES(a0)
    STORE s6, 8*REGBYTES(a0)
    STORE s7, 9*REGBYTES(a0)
    STORE s8, 10*REGBYTES(a0)
    STORE s9, 11*REGBYTES(a0)
    STORE s10, 12*REGBYTES(a0)
    STORE s11, 13*REGBYTES(a0)

    # restore to's registers
    LOAD ra, 0*REGBYTES(a1)
    LOAD sp, 1*REGBYTES(a1)
```

```
    LOAD s0, 2*REGBYTES(a1)
    LOAD s1, 3*REGBYTES(a1)
    LOAD s2, 4*REGBYTES(a1)
    LOAD s3, 5*REGBYTES(a1)
    LOAD s4, 6*REGBYTES(a1)
    LOAD s5, 7*REGBYTES(a1)
    LOAD s6, 8*REGBYTES(a1)
    LOAD s7, 9*REGBYTES(a1)
    LOAD s8, 10*REGBYTES(a1)
    LOAD s9, 11*REGBYTES(a1)
    LOAD s10, 12*REGBYTES(a1)
    LOAD s11, 13*REGBYTES(a1)

    ret # jump to the ra register

```

10. `forkret` function will enter `forkrets` function. This end up how the program is calling those two functions.

```
// forkret -- the first kernel entry point of a new thread/process
// NOTE: the addr of forkret is setted in copy_thread function
//       after switch_to, the current proc will execute here.
static void
forkret(void) {
    forkrets(current->tf);
}
```

11. `forkrets` function set stack to this new process's trapframe and `__trapret` restore all the registers directly from inside the interrupt frame

```
    .globl forkrets
forkrets:
    # set stack to this new process's trapframe
    move sp, a0
    j __trapret
```

12. the trapframe contains `epc` register which points to the `kernel_thread_entry`, so the program jump to where the `kernel_thread_entry` function is located

```
.text
.globl kernel_thread_entry
kernel_thread_entry: # void kernel_thread(void)
    move a0, s1
    jalr s0
    jal do_exit
```