Lecture 5: Process

What is process, history of OS, data structures for process, fork, exec, exit, wait

1. What is a process

A process is a program loaded in the memory.

2. History of OS

’45-’55 : No OS.

’55-’65 : FMS (Fortran Monitoring System). Runs a set of programs one by one.

cpu utilization problem: over 90% time the cpu was idle

’65- : Multiprogramming. Load several programs at once. Run one by one. If the current one is blocked because of I/O, go to the next process. Fairness problem: cpu-bound program is favored.

Now: Time Sharing. Allocate x ticks to each process. Now all processes will be stopped either if it executes an I/O instruction or the timer expires.

3. Data structures for process

process descriptor:

- one for each process.

- contains information about the corresponding process such as process ID, ticks

remained, register values of the last stop-point, memory location, size, etc.

- task\_struct{} is the process descriptor in Linux. (include/linux/sched.h)

- "current" points to the current process

- thread\_union{} = task\_struct{} + Kernel Mode Stack

- init\_task is the kernel's process descriptor (arch/x86/kernel/init\_task.c;

include/linux/init\_task.h)

struct task\_struct init\_task = INIT\_TASK(init\_task);

INIT\_TASK(init\_task)={

.....

.parent = &init\_task,

.comm = "swapper",

........

}

process queue:

- linked list of process descriptors

- &init\_task is the start of this queue.

- traversing the process queue

void display\_processes(){

struct task\_struct \*temp;

temp=&init\_task;

for(;;){

print process id, program name, process state (id, comm, state) for temp

temp=next\_task(temp); // find next process

if (temp==&init\_task) break;

}

printk("\n");

}

run queue

- A linked list of runnable processes.

- The scheduler looks at this queue to pick the next process **after each interrupt**

- Each cpu has its own run queue

- Each run queue is an array of queues based on the priority

: struct prio\_array{}.queue[]

(homework)

3.1) task\_struct is defined in include/linux/sched.h (search for "task\_struct {"). Which fields of the task\_struct contain information for process id, parent process id, user id, process status, the memory location of the process, the files opened, the priority of the process, program name?

3.2) Display all processes with "ps –ef". Find the pid of "ps -ef", the process you have just executed. Find the pid and program name of the parent process of it, then the parent of this parent, and so on, until you see the init\_task whose process ID is 0.

3.3) Define display\_processes() in init/main.c (right before the first function definition). Call this function in the beginning of start\_kernel(). Confirm that there is only one process in the beginning. Find the location where the number of processes becomes 2. Find the location where the number of processes is the greatest. Use "dmesg" to see the result of display\_processes().

3.4) Make a system call that, when called, displays all processes in the system. Run an application program that calls this system call and see if this program displays all processes in the system.

3.5) Run three user programs, f1, f2, and f3, and run another program that calls the above system call as follows. State 0 means runnable and 1 means blocked. Observe the state changes in f1, f2, f3 and explain what these changes mean.

f1:

int i,j; double x=1.2;

for(i=0;i<100;i++){

for(j=0;j<1000000;j++){ // make f1 busy for a while

x=x\*x;

}

// and then sleep 1sec

usleep(1000000);

}

f2:

int i,j; double x=1.2;

for(i=0;i<100;i++){

for(j=0;j<1000000;j++){ // make f2 busy for a while

x=x\*x;

}

// and then sleep 2sec

usleep(2000000);

}

f3:

int i,j; double x=1.2;

for(i=0;i<100;i++){

for(j=0;j<1000000;j++){ // make f3 busy for a while

x=x\*x;

}

// and then sleep 3sec

usleep(3000000);

}

ex1.c:

for(i=0;i<100;i++){

sleep(5);

syscall(17); // show all processes

// assuming the system call number in homework (3.4) is 17

}

#./f1&

#./f2&

#./f3&

#./ex1

4. Kernel code for process data structures and the scheduler

include/linux/sched.h

struct task\_struct{

long state; // -1: unrunnable, 0: runnable, >0 : stopped

int prio; // priority

const struct sched\_class \*sched\_class; // scheduling functions depending on

// scheduling class of this process

struct sched\_entity se: // scheduling info

struct list\_head tasks; //points to next task

struct mm\_struct \*mm; // memory occupied by this process

pid\_t pid;

struct task\_struct \*parent;

struct list\_head children;

uid\_t uid; // owner of this process

char comm[TASK\_COMM\_LEN]; // program name

struct thread\_struct thread; // pointer to saved registers

struct fs\_struct \*fs;

struct files\_struct \*files;

struct signal\_struct \*signal;

struct sighand\_struct \*sighand;

}

struct sched\_class{// fair class has func name such as task\_tick\_fair, enqueue\_task\_fair..

// rt class has task\_tick\_rt, enqueue\_task\_rt, ...

void (\*enqueue\_task)(struct rq \*rq, struct task\_struct \*p, ...);

void (\*dequeue\_task)(struct rq \*rq, struct task\_struct \*p, ...);

struct task\_struct \*(\*pick\_next\_task)(struct rq \*rq);

void (\*task\_tick)(struct rq \*rq, struct task\_struct \*p,...);

.........

}

struct sched\_entity{

u64 sum\_exec\_runtime;

u64 vruntime; // actual runtime normalized(weighted) by the number of

// runnable processes. unit is nanosecond

................

}

struct list\_head is little tricky. It does not point to the next item directly. For example,

struct list\_head tasks;

does not mean “(current->tasks).next” points to the next task.

include/linux/list.h

struct list\_head{

struct list\_head \*next, \*prev;

};

(current->tasks).next simply points to another struct list\_head that is included in the next process. To find the next process, use macro: list\_entry() or next\_tasks()

list\_entry( (current->tasks).next, struct task\_struct, tasks)

or

next\_task(current)

We can display all processes in the system by

struct task\_struct \*temp;

temp=&init\_task;

for(;;){

printk(“pid %d “,temp->pid);

temp=list\_entry(temp->tasks.next, struct task\_struct, tasks);

if (temp==&init\_task) break;

}

To display run queues, it is more difficult. Each process has a priority (“prio” field in task\_struct), and there are 0 to 139 priorities. For each priority we have different run queue. “run\_list” points to the next process in the run queue with the priority of the corresponding process. this\_rq() will point to the “struct rq” structure for the current cpu. This structure contains 140 run queues.

kernel/sched.c

union thread\_union{

struct thread\_info thread\_info;

unsigned long stack[THREAD\_SIZE/sizeof(long)]; // 8192 bytes

}

#define next\_task(p) list\_entry(rcu\_dereference((p)->tasks.next), struct task\_struct, tasks)

#define for\_each\_process(p) for(p=&init\_task;(p=next\_task(p))!=&init\_task;) do

arch/i386/kernel/init\_task.c

union thread\_union init\_thread\_union;

struct task\_struct init\_task = INIT\_TASK(init\_task);

include/asm-i386/thread\_info.h

struct thread\_info{

struct task\_struct \*task; // main task structre

struct exec\_domain \*exec\_domain; // execution domain

long flags;

long status;

\_\_u32 cpu; // cpu for this thread

mm\_segment\_t addr\_limit; // 0-0xBFFFFFFF (3G bytes) for user-thread

// 0-0xFFFFFFFF (4G bytes) for kernel-thread

}

kernel/sched.c

#define DEF\_TIMESLICE (100\*HZ/1000) // 100 ms for default time slice. HZ=1000

// HZ is num of timer interrupts per second.

struct prio\_array{ // run queue

unsigned int nr\_active;

struct list\_head queue[MAX\_PRIO]; // run queues for various priorities

};

void \_\_activate\_task(p, struct rq \*rq){ // wake up p

struct prio\_array \* target = rq->active;

enqueue\_task(p, target);

p->array = array;

}

process priority: each process has priority in “prio” (value 0..139)

0..99 is for real time task. 100..139 for user process

140 priority list

The kernel calls schedule() at the end of each ISR(Interrupt Service Routine) to pick the next process.

kernel/sched.c

void schedule(){

struct task\_struct \*prev, \*next;

struct prio\_array \*array;

prev = current;

rq = this\_rq(); // run queue of the belonging cpu

array = rq->active; // active run queue

deactivate\_task(prev, rq);

idx = sched\_find\_first\_bit(array->bitmap);

queue = array->queue + idx;

/\*\* old code

next = list\_entry(queue->next, struct task\_struct, run\_list); // next task

array = next->array;

\*\*/

next=pick\_next\_task(rq, prev);

rq->curr = next; // next is the curr task

context\_switch(rq, prev,next); // move to next

}

struct task\_struct \* pick\_next\_task(..){

class=sched\_class\_highest;

p=class->pick\_next\_task(rq);

return p;

}

struct task\_struct \*pick\_next\_task\_fair(rq){ // for cfs case

cfs\_rq=&rq->cfs;

se=pick\_next\_entiry(cfs\_rq);

p = task\_of(se);

return p;

}

struct sched\_entity \*pick\_next\_entity(..){

rb\_entry(.....); // find the next task in rb tree

}

#define this\_rq() (&\_\_get\_cpu\_var(runqueues))

static DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct rq, runqueues);

#define DEFINE\_PER\_CPU\_SHARED\_ALIGNED(type, name) \

\_\_attribute\_\_((\_\_section\_\_(“.data.percpu”))) \_\_typeof\_\_(type) per\_cpu\_\_##name

The above will make

static struct rq per\_cpu\_ruqueues;

void wake\_up\_new\_task(struct task\_struct \*p, ..){

struct rq \*rq, \*this\_rq;

int this\_cpu, cpu;

rq = task\_rq\_lock(p, ...); // the runqueue of the cpu this task belongs to

cpu = task\_cpu(p); // cpu p belongs to

\_\_activate\_task(p, rq); // insert p in rq

}

void scheduler\_tick(){ // timer interrupt calls this to

// decrease time slice of current p (in old code)

// in new code (after 2.6.23), it increases curr->se->vruntime

int cpu=smp\_processor\_id();

struct rq \*rq=cpu\_rq(cpu);

struct task\_struct \*curr=rq->curr;

curr->sched\_class->task\_tick(rq, curr, 0); // task\_running\_tick() in old code

.......

}

/\*\* old code

void task\_running\_tick(rq, p){

if (!--p->time\_slice){ // decrease it. if 0, reschedule

dequeue\_task(p, rq->active);

set\_tsk\_need\_resched(p);

p->time\_slice=task\_timeslice(p); // reset time slice

enqueue\_task(p, rq->active); // put back at the end

}

}

\*\*/

void task\_tick\_fair(rq, curr, ..){

se=&curr->se; // sched\_entity

cfs\_rq=cfs\_rq\_of(se);

entity\_tick(cfs\_rq, se, ...);

}

void entity\_tick(cfs\_rq, ...){

update\_curr(cfs\_rq);

.........

}

void update\_curr(cfs\_rq){

struct sched\_entity \*curr=cfs\_rq->curr;

now=rq\_of(cfs\_rq)->clock;

delta\_exec=now - curr->exec\_start; // running time so far for curr

\_\_update\_curr(cfs\_rq, curr, delta\_exec);

curr->exec\_start = now;

}

void \_\_update\_curr(cfs\_rq, struct sched\_entity \*curr, delta\_exec){

curr->sum\_exec\_runtime += delta\_exec;

delta\_exec\_weighted=calc\_delta\_fair(delta\_exec, curr);

curr->vruntime += delta\_exec\_weighted;

}

5. fork

When the kernel starts, we have only one process: "init\_task", which represents the kernel itself. Other processes are created by "fork".

1) fork system call duplicates a process.

example:

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int x;

main(){

   x= fork();

   if (x!=0) {

      printf("korea %d\n", x);

      while (1);

   }

   else {

      printf("china\n");

      while (1);

   }

}

What is the result of above code?

2) Algorithm of fork

fork() ==> mov $2, %eax

int $0x80

==> system\_call ==> arch/x86/kernel/process\_32.c/sys\_fork

=> kernel/fork.c/do\_fork()

fork is translated into 2 assembly instructions as below by C library:

mov $2, %eax

int $0x80

The ISR for interrupt 0x80 is “system\_call” which calls in turn “sys\_fork” when eax=2. “sys\_fork” calls do\_fork and do\_fork does followings:

(1) copy the body of the parent process

(2) copy thread\_union of the parent process, and adjust some information

(3) insert child into the process queue

(4) return 0 to the child, and return child’s pid to the parent

3) kernel\_thread

kernel\_thread() is similar to fork() except it does not copy the body of the parent.

start\_kernel(){

       trap\_init();

       init\_IRQ();

       time\_init();

       console\_init();

       ...............

       rest\_init();

   }

   rest\_init(){

       .........

       kernel\_thread(kernel\_init, ...........);

pid=kernel\_thread(kthreadd, ....);

schedule();

       cpu\_idle();

}

The above Linux code calls kernel\_thread(kernel\_init, ....). After this call the kernel is duplicated (but only the thread\_union of the kernel is duplicated), and the child's starting location is “kernel\_init()”. Similarly, after kernel\_thread(kthreadd,...), another child is born whose starting location is kthreadd. Since we have three processes, they will be scheduled one by one.

6. exec

1) exec system call transforms one process to another

ex1.c

main(){

   printf("I am ex1\n");

}

ex2.c

main(){

   execve("./ex1", 0,0);

   printf("I am ex2\n");

}

$cc ex1.c -o ex1

$cc ex2.c -o ex2

$ex2

What will be the result?

2) Algorithm of exec

exec ==> mov $11, %eax ==> system\_call ==> sys\_execve

int $0x80

“exec” is translated into 2 assembly instructions as below by C library:

mov $11, %eax

int $0x80

The ISR for interrupt 0x80 is “system\_call” which calls in turn “sys\_execve” when eax=11. (sys\_execve is in arch/x86/kernel/process\_32.c)

“sys\_execve” calls do\_execve(fs/exec.c) which does following things:

- remove old body

- load new body

- update the task\_struct

- update the KMS (the stack portion in thread\_union) such that

KMS.eip = starting location of the new body

3) example code

"exec1.c"

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int x;

void main(){

char \* exec2 = "./exec2";

char \* argv[2];

argv[0]=exec2;

argv[1]=0;

x=fork();

if (x!=0){

printf("korea %d\n",x);

execve("./exec2", argv, 0);

printf("exec failed\n");

}

else{

printf("japan\n");

for(;;);

}

}

"exec2.c"

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int x;

void main(){

printf("china\n");

}

gcc -o exec2 exec2.c

gcc -o exec1 exec1.c

exec1

7. shell

shell uses fork() and exec() to run the command:

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

void main(){

   int x,y;

   char buf[50];

   char \* argv[2];

   for(;;){

      printf("$ ");

      scanf("%s", buf); // get command. no arg can be input this way

      argv[0]=buf;

      argv[1]=NULL;

      x=fork();

      if (x==0){ // child

          printf("I am child to execute %s\n", buf);

          y=execve(buf, argv, 0);

          if (y<0){

             printf("exec failed. errno is %d\n", errno);

             exit(1);

          }

      } else wait();

   }

}

8. Initialization process of Linux

start\_kernel()

==> rest\_init()

==> kernel\_thread(kernel\_init, ....);

// now we have two processes (init\_task and kernel\_init)

==> kernel\_thread(kthreadd, ...); // init\_task runs first and create another thread.

// now we have three processes(init\_task, kernel\_init, kthredd)

==> schedule(); // init\_task calls schedule. the scheduler picks kernel\_init.

// prio of init\_task is 140. prio of the other two is 120.

==> kernel\_init()

==> do\_basic\_setup()

...........

init\_post();

==> init\_post()

==> if (ramdisk\_execute\_command){

run\_init\_process(......);

}

==> execve(“/init”, ...) ; //kernel\_init is transformed into /sbin/init.

==> /init

==> for(i=0;i < number of programs listed in /etc/inittab; i++){

x=fork();

if (x==0){ // child

execve(next program listed in /etc/inittab, ...);

}

} // parent goes back to the loop to create the next child

for(;;){ // parent

waits here;

}

==> fork();

==> child init calls execve(“/sbin/agetty”,..); // child init

// is transformed into /sbin/agetty

==> when user logins to this server /sbin/agetty execs to /bin/login

and display

login:

==> when user types login ID and password correctly /bin/login makes a child and

execs the child to the shell as specified in /etc/passwd, which is usually /bin/bash

root:x:0:0:root:/root:/bin/bash

...............

==> /bin/bash runs the shell code: display '#', read command, fork, let the child exec to the command, etc.

==> when user types ps -ef, shell forks and execs the child to ps -ef

init\_task->/sbin/init->/sbin/agetty->/bin/login->/bin/bash->ps –ef

9. What happens when you enter a Linux command?

Shell code again

.........

   for(;;){

      printf("$ ");

      scanf("%s", buf); // get command. no arg can be input this way

      .................

      x=fork();

      if (x==0){ // child

          y=execve(buf, argv, 0);

............

      } else wait();

   }

1) Shell runs printf("$"), and this library function calls write(1, "$", 1) which will display prompt. (printf => write => INT 128 => sys\_write => display "$")

$

2) Shell runs <scanf("%s", buf);>, and this library function calls read(0, buf, n) which will make the shell sleep until the user enters a command. Making shell sleep means setting shell's state to TASK\_INTERRUPTIBLE (a blocked state) and taking it out of the run queue. Since shell cannot be scheduled, the scheduler picks kernel (pid=0) as the next process and runs cpu\_idle(). (scanf => read => INT 128 => sys\_read => make shell sleep; cpu jumps to cpu\_idle)

3) A user types command.

$ ls<Enter>

4) Each key typing will raise keyboard interrupt.

press l => INT 33 => atkbd\_interrupt => store l => cpu goes back to cpu\_idle() => release l => INT 33 => atkbd\_interrupt => ignore key release => cpu goes back to cpu\_idle() => press s => INT 33 => ...... => press <Enter> => INT 33 => atkbd\_interrupt => store "ls" in shell's buf and wakes up shell. Waking up shell means set its state to

TASK\_RUNNING and put it back to the run queue. Now the scheduler picks shell as the next process and shell resumes execution.

5) shell runs x=fork(), and fork() will make a child.

(fork => INT 128 => do\_fork() => make a child; assume the scheduler picks parent first)

6) parent shell runs wait(), and wait() will make it sleep. Now the scheduler picks child.

(wait => INT 128 => sys\_wait)

7) child shell runs execve("ls", ....) which will change it to /bin/ls program. The scheduler picks the child again (parent is still sleeping).

(execve => INT 128 => do\_execve)

8) /bin/ls runs and shows all file names in the current directory in the screen. At the end, it calls exit(). Exit() will make it a zombie (set its state to TASK\_ZOMBIE) and sends a signal to parent. This signal wakes up parent (set its state to TASK\_RUNNING). The scheduler now picks parent.

9) parent goes back to the beginning of for(;;) loop and runs printf("$").

$

10. Homework

1) Run the program below. What happens? Explain the result.

ex1.c:

void main(){

int x;

x=fork();

printf("x:%d\n", x);

}

2) Try below and explain the result.

ex1.c:

void main(){

fork();

fork();

for(;;);

}

# gcc –o ex1 ex1.c

# ./ex1 &

# ps –ef

3) Run following code. What happens? Explain the result.

ex1.c:

void main(){

int i; float y=3.14;

fork();

fork();

for(;;){

for(i=0;i<1000000000;i++) y=y\*0.4;

printf("%d\n", getpid());

}

}

4) Try below and explain the result.

ex1.c:

void main(){

char \*argv[10];

argv[0]=”./ex2”;

argv[1]=0;

execve(argv[0], argv, 0);

}

ex2.c

void main(){

printf("korea\n");

}

#gcc –o ex1 ex1.c

#gcc –o ex2 ex2.c

#./ex1

5) Run following code and explain the result.

void main(){

char \*argv[10];

argv[0]=”/bin/ls”;

argv[1]=0;

execve(argv[0], argv, 0);

}

6) Run following code and explain the result.

void main(){

char \*argv[10];

argv[0]=”/bin/ls”;

argv[1]="-l";

argv[2]=0;

execve(argv[0], argv, 0);

}

11. Homework

1) Try the shell code in section 7. Try Linux command such as "/bin/ls", "/bin/date", etc.

2) Print the pid of the current process (current->pid) inside rest\_init() and kernel\_init(). The pid printed inside rest\_init() will be 0, but the pid inside kernel\_init() is 1. 0 is the pid of the kernel itself. Why do we have pid=1 inside kernel\_init()?

3) What happens if the kernel calls "kernel\_init" directly instead of calling kernel\_thread(kernel\_init, ...) in rest\_init()? Trace the kernel code and show where the kernel falls into panic.

4) The last function call in start\_kernel() is rest\_init(). If you insert printk() after rest\_init(), it is not displayed during the system booting. Explain the reason.

void start\_kernel(){

............

printk("before rest\_init\n"); // this will be printed out

rest\_init();

printk("after rest\_init\n"); // but this will not.

}

5) The CPU is either in some application program or in Linux kernel. You always should be able to say where is the CPU currently. Suppose we have a following program (ex1.c).

void main(){

printf("korea\n");

}

When the shell runs this, CPU could be in shell program or in ex1 or in kernel. Explain where is CPU for each step of this program. Start the tracing from the moment when the shell prints a prompt until it prints next prompt.

shell: printf("$"); // CPU is in shell

=> write(1, "$", 1) // CPU is in c library

=> INT 128 // CPU is in c library

kernel:

sys\_write() // CPU is in kernel and display '$' in screen

// after sys\_write() kernel schedules shell again

// and CPU goes back to shell

shell: scanf("%s", buf); // CPU is in shell

..............

.............

6) Trace fork, exec, exit, wait system call to find the corresponding code for the major steps of each system call.

7) Explain the result of following:

#./startsys;./sysnum;./stopsys

where, startsys sets the kernel flag so that system call number can be displayed, stopsys resets it, and sysnum calls printf.

sysnum.c:

void main(){

printf("hi\n");

}

startsys.c:

void main(){

syscall(31); // start printing sysnum

}

stopsys.c:

void main(){

syscall(32); // stop printing sysnum

}

8) When the shell runs

execve(argv[0], argv, 0);

how the Linux knows the value of argv?

9) Write a program that causes divide-by-zero fault:

int x,y;

y=0;

x=20/y;

This program, when run, will print:

Floating point exception

and dies. It dies because of divide-by-zero exception. Modify the kernel such that the system prints instead (when this program runs):

Divide-by-zero exception

Floating point exception

\*\* to make a call to a new function from entry.S, you need to protect registers as follows:

SAVE\_ALL

call new\_function

RESTORE\_REGS

11. exit

All programs end with "exit()" system call. Even If the programmer didn't put "exit()" in his code, the compiler will provide it in crtso (C run-time start-off function).

1) algorithm

- remove body

- make it a zombie

- send SIGCHLD to the parent

- adopt children to init process

- schedule next process

2) kernel code

exit -> sys\_exit ->

kernel/exit.c: do\_exit(){

struct task\_struct \*tsk = current;

exit\_mm(tsk); // remove body

exit\_sem(tsk);

\_\_exit\_files(tsk);

\_\_exit\_fs(tsk); // remove resouces

exit\_notify(tsk); // send SIGCHLD to the parent, ask init to adopt my own child,

// set tsk->exit\_state = EXIT\_ZOMBIE to make it a zombie

tsk->state = TASK\_DEAD;

schedule(); // call a scheduler

}

12. wait

The parent should wait in "wait" to collect the child; otherwise the child stays as a zombie consuming 8192 bytes of the memory.

1) algorithm

if child has exit first (that is, if the child is a zombie)

let it die completely (remove its process descriptor)

else (child is not dead yet)

block parent

remove parent from the run queue

schedule next process

When later the child exits, the parent will get SIGCHLD, wakes up, and be inserted into the run queue.

2) kernel code

wait -> sys\_wait4 ->

kernel/exit.c : do\_wait(){

struct task\_struct \*tsk;

DECLARE\_WAITQUEUE(wait, current); // make a "wait" queue

add\_wait\_queue(&current->signal->wait\_chldexit, &wait);

current->state = TASK\_INTERRUPTIBLE; // block parent

tsk = current;

do{

list\_for\_each(\_p, &task-children){

p = list\_entry(...);

if (p->exit\_state == EXIT\_ZOMBIE){ // if child has exit first

wait\_task\_zombie(p, ...); // kill it good

break;

}

// otherwise, it's still alive. wait here until it is dead

wait\_task\_contiuned(p,...);

break;

}

schedule();

}