Operating Systems

User Space Preemtive Scheduler

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

In this episode

- We will construct on the structure laid out in the previous lesson, a user space preemptive scheduler
- We need
  - a surrogate of the timer interrupt (we will use signals)
  - contexts for coroutines to store the possible "stacks" of our system
    - N contexts (one per process)
    - 1 context to handle the OS trap
    - 1 context to handle the timer interrupt
- Our system is purely user space.
  - We cannot implement the fork directly since it would involve copying "the memory" and preserving the links. Too complicated without access to the page table.
  - We will replace the fork() with a spawn(<function pointer>) similar in behavior to pthread create
  - Our spawn preserves the parent-child relationship of fork

Contexts

- 1 context per process (together with its stack), we put it in the PCB
- 1 context for the "interrupt"
- 1 context for the OS trap
- 1 context for the main, used on shutdown

```
typedef struct PCB{
  ListItem list;  // MUST BE THE FIRST!!!
  int pid;
  int return_value; // ret value for the parent
  ProcessStatus status;
  int signals;
  int signals_mask;
  ListHead descriptors;
  struct PCB* parent;
  ListHead children;

  ucontext_t cpu_state; // the context
  char stack[STACK_SIZE]; // the context stack

  int syscall_num;
  long int syscall_args[DSOS_MAX_SYSCALLS_ARGS];
  int syscall_retvalue;
} PCB;

// in disastrOS.c, global variables

ucontext_t interrupt_context;
ucontext_t trap_context;
ucontext_t main_context;
```

Interrupts

We will mimic the interrupts with SIGALARM.

- SIGALARM is a signal that can be programmed to be sent periodically to our process
- The signal handler for sigalarm, will switch to the interrupt context,
- The interrupt is a function that will call the scheduler and "jump" transfer control to the next process, whose context is stored in

  running->cpu_state

```
//set up the signal action
void setupSignals(void) {
  struct sigaction act;

  // timerHandler is the function called
  // when signal is received!
  act.sa_sigaction = timerHandler;

  // restart the signal handler,
  // and take the handler from the
  // sa_sigaction field
  act.sa_flags = SA_RESTART | SA_SIGINFO;

  // handle only sigalarm
  sigemptyset(&act.sa_mask);
  sigemptyset(&signal_set);
  sigaddset(&signal_set, SIGALRM);

  // install the handler
  if(sigaction(SIGALRM, &act, NULL) != 0) {
    perror("Signal handler");
  }

  // start a system  timer that will raise
  // a sigalarm each INTERVAL ms
  struct itimerval it;
  it.it_interval.tv_sec = 0;
  it.it_interval.tv_usec = INTERVAL * 1000;
  it.it_value = it.it_interval;
  if (setitimer(ITIMER_REAL, &it, NULL) )
     perror("setitiimer");
}
```

Interrupts

We will mimic the interrupts with SIGALARM.

- SIGALARM is a signal that can be programmed to be sent periodically to our process
- The signal handler for sigalarm, will switch to the interrupt context,
- The interrupt is a function that will call the scheduler and "jump" transfer control to the next process, whose context is stored in

  running->cpu_state

```
void timerHandler(int j,
         siginfo_t *si,
         void *old_context) {
  // this saves the running context in the PCB
  swapcontext(&running->cpu_state,
              &interrupt_context);
}

void timerInterrupt(){

  ++disastrOS_time;
  printf("time: %d\n", disastrOS_time);

  // call the scheduler!!!
  internal_schedule();

  // this jumps to the pcb context
  // WITHOUT saving the state of the interrupt
  // next calls to the interrupt will restart
  // from the beginning of the function!
  setcontext(&running->cpu_state);
}
```

Trap

When invoking the syscall, we will

- pack the arguments in registers
- trap to the OS (swapping context). The swap operation will save the state in the process PCB

The trap will

- Decode the syscall, and call the appropriate routine in the syscall vector
- jump to the context of the running PCB

```
int disastrOS_syscall(int syscall_num, ...) {
  assert(running);
  va_list ap;
  if (syscall_num<0||syscall_num>DSOS_MAX_SYSCALLS)
    return DSOS_ESYSCALL_OUT_OF_RANGE;

  // pack the arguments of the
  // syscalls in the running pcb
  int nargs=syscall_numarg[syscall_num];
  va_start(ap,syscall_num);
  for (int i=0; i<nargs; ++i){
    running->syscall_args[i] = va_arg(ap,long int);
  }
  va_end(ap);
  running->syscall_num=syscall_num;

  // save the state of the CPU and
  // in PCB and jump to the trap
  swapcontext(&running->cpu_state, &trap_context);
  return running->syscall_retvalue;
}
```

Trap

When invoking the syscall, we will

- pack the arguments in registers
- trap to the OS (swapping context). the swap operation will save the state in the process PCB

The trap will

- Decode the syscall, and call the appropriate routine in the syscall vector
- jump to the context of the running PCB

```
void disastrOS_trap(){
  int syscall_num=running->syscall_num;

  if (syscall_num<0||syscall_num>DSOS_MAX_SYSCALLS){
    running->syscall_retvalue
     = DSOS_ESYSCALL_OUT_OF_RANGE;
    goto return_to_process;
  }
  SyscallFunctionType my_syscall
     =syscall_vector[syscall_num];
  if (! my_syscall) {
    running->syscall_retvalue
      =DSOS_ESYSCALL_NOT_IMPLEMENTED;
    goto return_to_process;
  }

  (*syscall_vector[syscall_num])();

 return_to_process:
  if (running) {
    // trap to the process context saved
    // before the syscall
    setcontext(&running->cpu_state);
  } else {
    printf("no active processes\n");
    disastrOS_printStatus();
  }
}
```

Initializing Contexts

- We need to set the trap context so that it executes the disastrOS_trap()
- The trap context should "mask" the timer interrupts. This is done by setting the signal mask in the context
- We need to set the interrupt context so that it executes timerInterrupt()

```
void disastrOS_start(void (*f)(void*),
              void* f_args, char* logfile){

// .........

  // we will come back here on shutdown
  getcontext(&main_context);
  if (shutdown_now)
    exit(0);

  // setting system trap
  getcontext(&trap_context);
  trap_context.uc_stack.ss_sp = system_stack;
  trap_context.uc_stack.ss_size = STACK_SIZE;

  // we mask sigalarm when handing a trap
  sigemptyset(&trap_context.uc_sigmask);
  sigaddset(&trap_context.uc_sigmask, SIGALRM);
  trap_context.uc_stack.ss_flags = 0;
  trap_context.uc_link = &main_context;
  makecontext(&trap_context, disastrOS_trap, 0);

  // the interrupt and the system
  // share the same stack
  interrupt_context=trap_context;
  interrupt_context.uc_link = &main_context;
  sigemptyset(&interrupt_context.uc_sigmask);
  makecontext(&interrupt_context, timerInterrupt,0);

// .........

}
```

spawn

- We substitute the fork with a spawn routine that starts a thread.
- A thread has the following prototype

    void f(void* arg)

- Spawn creates a new PCB and preserves the parent child-relation as done by fork. The new PCB is put in ready state.
- The context in the newly created PCB should accept timer interrupts.
- The stack is stored in the PCB

The syscall arguments are:

  - args[0]: f_ptr
  - args[1]: arg
- The newly created context will start the function pointed by f_ptr, with arguments arg

```
void internal_spawn(){
  static PCB* new_pcb;
  new_pcb=PCB_alloc();
  if (!new_pcb) {
    running->syscall_retvalue=DSOS_ESPAWN;
    return;
  }

  new_pcb->status=Ready;
  new_pcb->parent=running;
  PCBPtr* new_pcb_ptr=PCBPtr_alloc(new_pcb);
  assert(new_pcb_ptr);
  List_insert(&running->children,
              running->children.last,
              (ListItem*) new_pcb_ptr);
  List_insert(&ready_list,
              ready_list.last,
             (ListItem*) new_pcb);

  running->syscall_retvalue=new_pcb->pid;

  getcontext(&new_pcb->cpu_state);
  new_pcb->cpu_state.uc_stack.ss_sp=new_pcb->stack;
  new_pcb->cpu_state.uc_stack.ss_size=STACK_SIZE;
  new_pcb->cpu_state.uc_stack.ss_flags=0;
  sigemptyset(&new_pcb->cpu_state.uc_sigmask);
  new_pcb->cpu_state.uc_link = &main_context;
  void (*new_function)(void*)=
     (void(*)(void*))running->syscall_args[0];
  makecontext(&new_pcb->cpu_state,
             (void(*)())new_function,
             1,
             (void*)running->syscall_args[1]);
}
```

Run baby run

We just implemented an user space preemptive scheduler.

We can now run a bunch of threads

There will be a thread listening the keyboard and printing the status on the screen each time we press enter

There will be a bunch of threads executing dummy iterations. The number of iterations depends on the PID

Init

- will spawn the threads
- wait for the termination of all threads

When done it will call disastros_shutdown() to return to the main.

The main will just start the system, with the disastrOS_start.

```
void waitABit() {
  for (int i=0; i<100000000; ++i);
}
// we need this to handle the sleep state
void sleeperFunction(void* args){
  printf("Hello, I am the sleeper,
         and I sleep %d\n",disastrOS_getpid());
  while(1) {
    getc(stdin);
    disastrOS_printStatus();
  }
}

void childFunction(void* args){
  printf("Hello, I am the child
          function %d\n",disastrOS_getpid());
  printf("I will iterate a bit,
          before terminating\n");
  for (int i=0; i<(disastrOS_getpid()+1); ++i){
    printf("PID: %d, iterate %d\n",
           disastrOS_getpid(), i);
    waitABit();
  }
  printf("PID: %d, terminating\n",
         disastrOS_getpid());
  disastrOS_exit(disastrOS_getpid()+1);
}
```

Run baby run

We just implemented an user space preemptive scheduler.

We can now run a bunch of threads

There will be a thread listening the keyboard and printing the status on the screen each time we press enter

There will be a bunch of threads executing dummy iterations. The number of iterations depends on the PID

Init

- will spawn the threads
- wait for the termination of all threads

When done it will call disastros_shutdown() to return to the main.

The main will just start the system, with the disastrOS_start.

```
void initFunction(void* args) {
  disastrOS_printStatus();
  printf("hello, I am init and I just started\n");
  disastrOS_spawn(sleeperFunction, 0);


  printf("I feel like to spawn 10 nice threads\n");
  int alive_children=0;
  for (int i=0; i<10; ++i) {
    disastrOS_spawn(childFunction, 0);
    alive_children++;
  }

  disastrOS_printStatus();

  printf("waiting for childs to terminate...\n");
  int retval;
  int pid;
  while(alive_children>0 &&
   (pid=disastrOS_wait(0, &retval))>=0){
    disastrOS_printStatus();
    printf("initFunction,
          child: %d terminated,
          retval:%d, alive: %d \n",
        pid, retval, alive_children);
    waitABit();
    --alive_children;
  }
  printf("shutdown!");
  disastrOS_shutdown();
}
```

Run baby run

We just implemented an user space preemptive scheduler.

We can now run a bunch of threads

There will be a thread listening the keyboard and printing the status on the screen each time we press enter

There will be a bunch of threads executing dummy iterations. The number of iterations depends on the PID

Init

- will spawn the threads
- wait oir the termination of all threads

When done it will call disastros_shutdown() to return to the main.

The main will just start the system, with the disastrOS_start.

```
int main(int argc, char** argv){
  char* logfilename=0;
  if (argc>1) {
    logfilename=argv[1];
  }
  // we create the init process processes
  // the first is in the running variable
  // the others are in the ready queue
  printf("the function pointer is: %p",
          childFunction);
  // spawn an init process
  printf("start\n");
  disastrOS_start(initFunction, 0, logfilename);
  return 0;
}
```

Exercises

Modify the scheduler routine and the spawn function so that each thread can be given a priority.

The scheduler will pick first the threads with higher priority.

Once a thread is picked, its priority value is decreased each time the thread is evicted.

When the priority is 0, its value is restored to the initial priority.