Operating Systems

Timers

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

In this episode

- We will replace the delay function that is cpu intensive with a timer mechanism that puts a thread that wants to sleep for a certain number of cycles in the waiting status.
- When enough time is passed, the process is put again in the ready list
- We need
  - A structure representing the timer event
  - A timer counter incremented by the interrupt routine
  - A queue of timer events to store all timers installed by a process
  - A syscall sleep(num_cycles) that installs a timer and puts a process to sleep

Timer

- A timer event stores
  - the awakening time of a process. This is the absolute time when a process should be woken up
  - The pointer to the pcb of the process to wake up
- The list of timer events is ordered by awakening time

```
typedef struct  {
  ListItem list;
  int awakening_time;
  struct PCB* pcb;
} TimerItem;

//list of timers
typedef ListHead TimerHandler;

// initializes a timer
void Timer_init();

// initializes a timer list
void TimerList_init(ListHead* timers);

// inserts in the list a new timer event
// the list is ordered by awakening time
TimerItem* TimerList_add(ListHead* timers,
              int awakening_time,
              struct PCB* pcb);

// returns, if existing
// a timer matching with current_time
     TimerItem* TimerList_current(ListHead* timers,
     int current_time);

// removes the head
int TimerList_removeCurrent(ListHead* timers);

void TimerList_print(ListHead* timers);

// deallocates a timer
int TimerItem_free(TimerItem* item);
```

Global Variables

We need a global variable to keep track of the progressing time.

We need a list of timers to keep track of the processes to wake up

This variable is incremented each time the timerInterrupt is invoked.

In the PID of each process we need to store a pointer (if any) to the timer event potentially installed by the process. This is just for bookkeeping.

```
FILE* log_file=NULL;
PCB* init_pcb;
PCB* running;
int last_pid;
ListHead ready_list;
ListHead waiting_list;
ListHead zombie_list;
ListHead timer_list;

ListHead resources_list;

SyscallFunctionType
syscall_vector[DSOS_MAX_SYSCALLS];
int syscall_numarg[DSOS_MAX_SYSCALLS];

ucontext_t interrupt_context;
ucontext_t trap_context;
ucontext_t main_context;
ucontext_t idle_context;
int shutdown_now=0; // used for termination
char system_stack[STACK_SIZE];

// process wide signal mask
sigset_t signal_set;
char signal_stack[STACK_SIZE];
volatile int disastrOS_time=0;
```

Global Variables

We need a global variable to keep track of the progressing time.

We need a list of timers to keep track of the processes to wake up

This variable is incremented each time the timerInterrupt is invoked.

In the PID of each process we need to store a pointer (if any) to the timer event potentially installed by the process. This is just for bookkeeping.

```
void timerInterrupt(){
  ++disastrOS_time;
  internal_schedule();
  setcontext(&running->cpu_state);
}
```

Global Variables

We need a global variable to keep track of the progressing time.

We need a list of timers to keep track of the processes to wake up

This variable is incremented each time the timerInterrupt is invoked.

In the PID of each process we need to store a pointer (if any) to the timer event potentially installed by the process. This is just for bookkeeping.

```
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;
  struct TimerItem *timer;

  char stack[STACK_SIZE];

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

Scheduler

Each time the scheduler is called we need to scan the list of timers to see if the current time matches the head of the list.

If this occurs, it means that we need to wake up the corresponding process, by moving it from the waiting to the ready queue.

We finally delete the timer event, and clear its reference in the PID of the process

```
void internal_schedule() {

  TimerItem* elapsed_timer=0;
  PCB* previous_pcb=0;
  while( (elapsed_timer
     =TimerList_current(&timer_list,
                        disastrOS_time)) ){
    PCB* pcb_to_wake=elapsed_timer->pcb;
    List_detach(&waiting_list,
                (ListItem*) pcb_to_wake);
    pcb_to_wake->status=Ready;
    pcb_to_wake->timer=0;
    List_insert(&ready_list,
                (ListItem*) previous_pcb,
                (ListItem*) pcb_to_wake);
    previous_pcb=pcb_to_wake;
    TimerList_removeCurrent(&timer_list);
  }
  // regular schedule
  if (ready_list.first){
    PCB* next_process=
         (PCB*) List_detach(&ready_list,
                            ready_list.first);
    running->status=Ready;
    List_insert(&ready_list,
                 ready_list.last,
                (ListItem*) running);
    next_process->status=Running;
    running=next_process;
  }
}
```

sleep(int cycles)

- This new syscall will put a process to sleep for a certain number of cycles
- When a thread calls sleep, a new timer is created
- The awakening time of the timer is set as current_time+cycles
- The process is put in the waiting queue

```
void internal_sleep(){
  if (running->timer) {
    printf("process has already a timer!!!\n");
    running->syscall_retvalue=DSOS_ESLEEP;
    return;
  }
  int cycles_to_sleep=running->syscall_args[0];
  int wake_time=disastrOS_time+cycles_to_sleep;

  TimerItem* new_timer=
     TimerList_add(&timer_list, wake_time, running);
  if (! new_timer) {
    printf("no new timer!!!\n");
    running->syscall_retvalue=DSOS_ESLEEP;
    return;
  }
  running->status=Waiting;
  List_insert(&waiting_list,
     waiting_list.last,
     (ListItem*) running);
  if (ready_list.first)
    running=(PCB*)
     List_detach(&ready_list, ready_list.first);
  else {
    running=0;
    printf ("they are all sleeping\n");
    disastrOS_printStatus();
  }
}
```

Timer test

We simply refer to the test program of the last episode, and we replace the waitABit with a call to sleep

```
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);
    disastrOS_sleep((20-disastrOS_getpid())*5);
  }
  printf("PID: %d, terminating\n",
         disastrOS_getpid());
  disastrOS_exit(disastrOS_getpid()+1);
}
```

Messages

When in kernel mode, we manage the system simply by changing the value of some data structures.

No matter what, when we exit kernel mode, the next process in running will be resumed.