Operating Systems

Kernel Structures Implementation

Giorgio Grisetti
grisetti@diag.uniroma1.it

Department of Computer Control and Management Engineering
Sapienza University of Rome

Variables

Recall the structures needed to manage processes

We need

- a PCB ptr to store address of first process (init)
- a PCB ptr to store the current process
- N queues
  - ready, running, waiting, etc
- a vector of function pointers to store the imlpementation of the syscalls
- a vector that tells how many arguments each syscall takes

variables declared extern in disastrOS_globals.h, defined in disastrOS.c

```
// pcb of the init process
extern PCB* init_pcb;

// pcb of the current proc running process
extern PCB* running;

// last pid to be generates
extern int last_pid;

// list of the ready processes
extern ListHead ready_list;

// list of the waiting processes
extern ListHead waiting_list;

// zombies
extern ListHead zombie_list;

// type of syscall function pointer
typedef void(*SyscallFunctionType)();


/* not exported, only in disastrOS.c*/
// vector of syscall ptrs
SyscallFunctionType
   syscall_vector[DSOS_MAX_SYSCALLS];

// how many args each syscall has
int syscall_numarg[DSOS_MAX_SYSCALLS];
```

PCB

The variables to be stored in the PCB depend on the implementation of the rest of the system

PCBs are organized in a list

Since we do not want to use any assembly, we will put in the PCB also a bunch of "fake" registers to store the syscall arguments

We want to organize our processes in a tree, similar to what the PCB does.

- The PCB has to store a list of pointers to the PCBs of the child processes

We don’t want to use malloc inside kernel. We will use a SLAB.

Declared in disastrOS_pcb.h

```
typedef enum ProcessStatus {
  Invalid=-1, Created=0x0, Running=0x1,
  Ready=0x2, Waiting=0x3, Suspended=0x4
  Zombie=0x5
} ProcessStatus;

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;

  //the one below is a hack for the syscalls
  //in a real system one needs to use the
  // cpu to pass
  //arguments to a syscall

  // we use long int so we can store
  // pointers on 64 bit machines
  int syscall_num;
  long int syscall_args[DSOS_MAX_SYSCALLS_ARGS];
  int syscall_retvalue;

  // more stuff to come

} PCB;
```

PCB

At runtime, the global variable running will store the pointer to the currently running process

In disastrOS_pcb.h we provide functions

- to allocate/deallocate
  - PCB list items
  - PCBptr list items
- print a list of PCBs
- retrieve a pcb ptr from a list, give its pid

```
// initializes the SLAB allocator
// for the PCB structures called
// in disastrOS_init
void PCB_init();

PCB* PCB_alloc(); // alloc & init  a new pcb block

int PCB_free(PCB* pcb); // frees a pcb block

void PCB_print(PCB* pcb); // prints a PCB

// returns a pcb whose PID is pid from a list
PCB* PCB_byPID(ListHead* head, int pid);

// prints a list of PCB
void PCBList_print(ListHead* head);

// this is a list of *pointers* to pcb
typedef struct PCBPtr{
  ListItem list;
  PCB* pcb;
} PCBPtr;

//
PCBPtr* PCBPtr_alloc(PCB* pcb);

int PCBPtr_free(PCBPtr* pcb);

PCBPtr* PCBPtr_byPID(ListHead* head, int pid);

void PCBPtrList_print(ListHead* head);
```

Syscall

Recall what a syscall invokation does

- gets the arguments from the stack and packs them in registers inside the pcb
- calls the OS through a trap
- on return, unpacks the arguments from the registers
- returns to the caller

We emulate the same mechanism with a function with variable arguments (see ... in the arg list)

The syscall is by definition invoked by the running process,

We can unpack the arguments from the stack and store them in the running PCB

Once this is done, we call disastrOS_trap() and we return the value stored in the running PCB

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

  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;
  disastrOS_trap();
  return running->syscall_retvalue;
}
```

Trap

The trap is technically an interrupt handler

- its task is to retrieve the syscall number from the CPU registers
- invoke the function whose address is at that location in the syscall vector

In the software you will find some conditional printf used for debug (disastrOS_debug(...)).

Not reported here for brevity

```
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;
  }

  (*my_syscall)();

 return_to_process:
  if (!running) {
    printf("no active processes\n");
    disastrOS_printStatus();
  }
}
```

fork()

Fork is easy:

- Creates a new pcb and puts it in the ready queue
- Sets the running as parent of the current pcb.
- Adds to the parent pcb child ptr list a pointer to the newly created pcb
- returns to the parent the PID of the children

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

  new_pcb->status=Ready;

  // sets the parent of the newly created
  // process to the running process
  new_pcb->parent=running;

  // adds a pointer to the new process
  // to the children list of running
  PCBPtr* new_pcb_ptr=PCBPtr_alloc(new_pcb);
  assert(new_pcb_ptr);

  List_insert(&running->children,
              running->children.last,
             (ListItem*) new_pcb_ptr);

  //adds the new process to the ready queue
  List_insert(&ready_list,
             ready_list.last,
             (ListItem*) new_pcb);

  //sets the retvalue for the caller
     to the new pid
  running->syscall_retvalue=new_pcb->pid;
}
```

wait(int pid)

Wait(int pid) has two behaviors

- if pid==0, the process waits for the termination of any of his children
- if pid!=0, the process waits for the termination of the specific child

We mimic these behavior in our implementation.

If the child process is already in the zombie status,

- the wait() returns to the invoking process the value returned by the child and stored in its PCB

otherwise

- the process is put in the waiting list and we set as next running the first in the ready queue

```
void internal_wait(){
  int pid_to_wait=running->syscall_args[0];
  int* result=(int*) running->syscall_args[1];
  // the process has no children
  if (running->children.first == 0){
    running->syscall_retvalue = DSOS_EWAIT;
    return;
  }
  // we scan the list of pcbs and we stop either
  // - at the first process in
  //   zombie status, if pid==0;
  // - at the first process whose
  //   pid is the queried one
  PCB* awaited_pcb=0;
  PCBPtr* awaited_pcb_ptr=0;
  ListItem* aux=running->children.first;
  while(aux){
    awaited_pcb_ptr=(PCBPtr*) aux;
    awaited_pcb=awaited_pcb_ptr->pcb;
    if (pid_to_wait==0&&awaited_pcb->status==Zombi
      break;
    if (pid_to_wait&&awaited_pcb->pid==pid_to_wait
      break;
    aux=aux->next;
  }
  // if we were looking for a process not in child
  // we need to return an error
  if(pid_to_wait>0){
    if(! awaited_pcb){
      pid_to_wait = -1;
      running->syscall_retvalue = DSOS_EWAIT;
      return;
    }
  }
```

wait(int pid)

wait(int pid) has two behaviors

- if pid==0, the process waits for the termination of any of his children
- if pid!=0, the process waits for the termination of the specific child

We mimic these behavior in our implementation.

If the child process is already in the zombie status,

- the wait() returns to the invoking process the value returned by the child and stored in its PCB

otherwise

- the process is put in the waiting list and we set as next running the first in the ready queue

```
//... continues
  // if the pid is in zombie status,
  // we return the value and free the memory
  if (awaited_pcb && awaited_pcb->status==Zombie) {
    // remove the awaited pcb from children list
    List_detach(&running->children,
                (ListItem*) awaited_pcb_ptr);
    PCBPtr_free(awaited_pcb_ptr);

    // remove he pc from zombie pool
    List_detach(&zombie_list,
                (ListItem*) awaited_pcb);
    running->syscall_retvalue = awaited_pcb->pid;
    if (result)
      *result = awaited_pcb->return_value;
    PCB_free(awaited_pcb);
    return;
  }

  // all fine, but the process is not a zombie
  // need to sleep
  running->status=Waiting;
  List_insert(&waiting_list,
              waiting_list.last,
              (ListItem*) running);

  // pick the next
  PCB* next_running=
     (PCB*) List_detach(&ready_list,
            ready_list.first);
  running=next_running;
}
```

exit(int exit_code)

when a process ends

- release the resources (not done in this stub)
- attach all children to init
- send a signal to all children
- become a zombie
- send a signal to the parent
- if the parent of the process executed a wait or waitpid **and** is in the waiting status we need to wake him up
- we need to send a signal to the parent telling the one of his children died
- if the parent is died, we need to reparent the process to init

```
void internal_exit(){
  // 2nd register in pcb contains the exit value
  running->return_value=running->syscall_args[0];

  assert(init_pcb);
  while(running->children.first){

    // detach from current list
    ListItem* item =
          List_detach(& running->children,
                         running->children.first);
    assert(item);

    // attach to init's children list
    List_insert(& init_pcb->children,
                  init_pcb->children.last, item);

    // send SIGHUP
    PCBPtr* pcb_ptr=(PCBPtr*) item;
    PCB* pcb=pcb_ptr->pcb;
    pcb->signals|=(DSOS_SIGHUP & pcb->signals_mask);
  }

  running->status=Zombie;
  List_insert(&zombie_list,
              zombie_list.last,
             (ListItem*) running);
  running->parent->signals
  |= (DSOS_SIGCHLD&running->parent->signals_mask);
```

exit(int exit_code)

when a process ends

- release the resources (not done in this stub)
- attach all children to init
- send a signal to all children
- become a zombie
- send a signal to the parent
- if the parent of the process executed a wait or waitpid **and** is in the waiting status we need to wake him up
- we need to send a signal to the parent telling the one of his children died
- if the parent is died, we need to reparent the process to init

```
if (running->parent->status==Waiting
    && running->parent->syscall_num==DSOS_CALL_WAIT
    && (running->parent->syscall_args[0]==0 ||
     running->parent->syscall_args[0]==running->pid)
    ){
  PCB* parent=
   (PCB*) List_detach(&waiting_list,
            (ListItem*) running->parent);
  assert(parent);
  parent->status=Running;

  PCBPtr* self_in_parent=
       PCBPtr_byPID(&parent->children,
                    running->pid);
  List_detach(&parent->children,
                 (ListItem*) self_in_parent);

  parent->syscall_retvalue=running->pid;
  int* result=(int*)parent->syscall_args[1];
  if (result)
    *result=running->return_value;

  // the process finally dies
  ListItem* suppressed_item =
    List_detach(&zombie_list, (ListItem*) running);
  PCB_free((PCB*) suppressed_item);
  running=parent;
} else {
  // we put the first ready process in running
  PCB* next_running=
  (PCB*)List_detach(&ready_list,ready_list.first);
  next_running->status=Running;
  running=next_running;
}
}
```

preempt()

- Fake syscall that invokes the scheduler to set as running the next process in the ready queue
- The scheduler *only* selects the next running process!

```
void internal_schedule() {
  if (running) {
    disastrOS_debug("PREEMPT - %d ->", running->pid);
   }
  // at least one process should be
  // in the running queue
  // if not, no preemption occurs

  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;
  }
  //disastrOS_printStatus();

  if (running) {
    disastrOS_debug(" %d\n", running->pid);
  }
}

void internal_preempt() {
  internal_schedule();
}
```

Initialization

- initialize allocators
- populate the syscall vector with appropriate function pointers
- initialize the global variables
- start a function, that we will call init

```
void disastrOS_start(void (*f)(void*),
                     void* f_args, char* logfile){
  /* INITIALIZATION OF SYSTEM STRUCTURES*/
  disastrOS_debug("initializing system structures\n")
  PCB_init();
  init_pcb=0;

  for (int i=0; i<DSOS_MAX_SYSCALLS; ++i){
    syscall_vector[i]=0;
  }
  syscall_vector[DSOS_CALL_PREEMPT]= internal_preempt
  syscall_numarg[DSOS_CALL_PREEMPT]= 0;

  syscall_vector[DSOS_CALL_FORK]    = internal_fork;
  syscall_numarg[DSOS_CALL_FORK]    = 0;

  syscall_vector[DSOS_CALL_SPAWN]   = internal_spawn;
  syscall_numarg[DSOS_CALL_SPAWN]   = 2;

  syscall_vector[DSOS_CALL_WAIT]    = internal_wait;
  syscall_numarg[DSOS_CALL_WAIT]    = 2;

  syscall_vector[DSOS_CALL_EXIT]    = internal_exit;
  syscall_numarg[DSOS_CALL_EXIT]    = 1;

  ....
  // add your own
```

Initialization

- initialize allocators
- populate the syscall vector with appropriate function pointers
- initialize the global variables
- start a function, that we will call init

```
  // setup the scheduling lists
  running=0;
  List_init(&ready_list);
  List_init(&waiting_list);
  List_init(&zombie_list);
  List_init(&resources_list);
  List_init(&timer_list);


  /* INITIALIZATION OF SYSCALL
     AND INTERRUPT INFRASTRUCTIRE*/
  running=PCB_alloc();
  running->status=Running;
  init_pcb=running;

  // we start the first process
  disastrOS_debug("starting\n");
  (*f)(f_args);
}
```

Testing

In this episode we only layed out the data structures for process management, in a simplified form.

There is no real task execution, as there is no real context switching, however we can see the evolution of the data structures by providing an execution trace.

The execution trace is the sequence of instructions executed by the cpu

by explicitly calling preempt we switch to the next thread, simulating a scheduler call.

```
.

void initFunction(void* args) {
  disastrOS_printStatus();
  // below a sequence of actions performed
  // by the running process
  // process switch might occur as a consequence of
  // a preempt action
  // or a system call

  // now we are in init
  // we pretend to fork
  printf("fork ");
  int fork_result = disastrOS_fork();
  printf(" child pid: %d\n", fork_result);
  disastrOS_printStatus();

  // we are still in the parent process;
  // we switch context;
  printf("preempt \n");
  disastrOS_preempt();
  disastrOS_printStatus();

  // parent forks three times
  printf("fork ");
  fork_result = disastrOS_fork();
  printf(" child pid: %d\n", fork_result);
  disastrOS_printStatus();
  printf("fork ");
  fork_result = disastrOS_fork();
  printf(" child pid: %d\n", fork_result);
  disastrOS_printStatus();

 ...
```

Exercises

Add a syscall that

- reverts the ready list,
- executes the next process in the ready list

If the ready list is empty the syscall returns an error.

Modify the test program to verify that the newly implemented syscall is correct