PROJECT 2

Process and Resource Management

1 PROJECT OVERVIEW

In this project, we will examine the portion of the kernel that addresses the management of processes and resources. We will develop a system that will allow us to create the data structures to represent processes and resources. We will also implement the operations invoked by processes to manipulate other processes or to request/release various resources. In the extended version of the manager, we will also implement operations that emulate the actions of hardware interrupts. The manager will be tested using a presentation shell developed as part of the project. This will allow us to test the manager without running the actual processes and using the machine's actual hardware interrupts. Instead, the presentation shell will play the role of both the currently running process and the hardware; it will accept commands typed in by the user, and will invoke the corresponding function of the manager.

2 BASIC PROCESS AND RESOURCE MANAGER

2.1 Process States

We assume there are only three process states: *ready, running, blocked*. The following table lists the possible operations a process may perform and the resulting state transitions.

Operation	old state		new state
Create	*(none)*	→	*ready*
Request	*running*	→	*blocked*
Release	*blocked*	→	*ready*
Destroy	*any*	→	*(none)*
Scheduler	*ready*	→	*running*
	running	→	*ready*

All of the above operations except the *Scheduler* are invoked directly by the currently running process—they represent *kernel calls*. The *Scheduler* is a function that is invoked automatically at the end of each of the operations.

2.2 Representation of Processes

Each process is represented by a data structure called the *process control block (PCB)*, as explained in Section 4.4.1 (Chapter 4). For this project, we use the following fields of the PCB:

ID
Memory
Other_Resources
Status
Creation_Tree
Priority

- *ID* is a unique process identifier by which the process may be referred to by other processes.
- *Memory* is a linked list of pointers to memory blocks requested by and currently allocated to the process. This field is used only if memory management is incorporated into this project (see Section 5, extension 2.)
- *Other_Resources* jointly represents all resources other than main memory that have been requested by and currently allocated to the process. It is implemented as a linked list.
- *Status* consists of two subfields, *Status.Type* and *Status.List*. Their meaning is explained in Section 4.4.1 (Chapter 4).
- *Creation_Tree* also consist of two subfields, *Creation_Tree.Parent* and *Creation_Tree.Child*. Their meaning also is explained in Section 4.4.1 (Chapter 4).
- *Priority* is the process priority and is used by the *Scheduler* to decide which process should be running next. We assume that priority is represented by an integer and is static.

Each PCB is created and destroyed dynamically using the operations *Create* and *Destroy* invoked by the currently running process. The only exception is the special *Init* process, whose PCB is created automatically when the system starts up, and destroyed when the system is shut down (see Section 2.5).

2.3 Representation of Resources

We assume that there is a set of resources a process may request, use, and later release. Examples of such resources are printers, terminals, other devices, or various software components or services. All resources are serially reusable, i.e., they can be used only by one process at a time. Each resource is represented by a data structure called the *resource control block (RCB)*. A RCB consists of the following fields:

RID
Status
Waiting_List

- *RID* is a unique resource identifier by which the resource may be referred to by processes.
- *Status* indicates whether the resource is currently free or allocated to other process.
- *Waiting_List* is a list of processes blocked on this resources. These are all the processes that have requested the resource but could not obtain it because it is currently being used by another process.

All resources are static and their RCBs are created by the system at start-up time.

2.4 Operations on Processes and Resources

The manager must support the four basic operations on processes: *Create, Destroy, Suspend*, and *Activate*. These have been outlined in Section 4.4.2 (Chapter 4). They may be simplified for this project because we are assuming only a single CPU, and because some of the fields have been omitted from the PCB.

The two operations on resources, *Request* and *Release*, are outlined in Section 4.5 (Chapter 4). The following algorithms present these operations at a more detailed level:

```
(1)  Request(RID) {
(2)    r = Get_RCB(RID);
(3)    if (r->Status == 'free') {
(4)       r->Status = 'allocated;'
(5)       insert(self->Other_Resources, r;)
(6)    } else {
(7)      self->Status.Type = 'blocked;'
(8)      self->Status.List = r;
(9)      remove(RL, self);
(10)     insert(r->Waiting_List, self; }
(11)   Scheduler();
(12) }
```

If the requested resource is currently available, the operation changes the resource status to *'allocated'* and makes it available to the calling process by entering a pointer to the RCB into the list of the process resources. If the requested resource is not currently available, the calling process is blocked. This includes changing the *Status.Type* to *'blocked'* and the *Status.List* to point to the RCB of the requested resource. The process also must be moved from the RL to the waiting list of the resource. Finally, the scheduler is called to select the process to run next.

```
(1)  Release(RID) {
(2)    r = Get_RCB(RID);
(3)    remove(self->Other_Resources, r);
```

```
(4)    if (Waiting_List == NIL} {
(5)      r->Status = 'free';
(6)    } else {
(7)      remove(r->Waiting_List, q);
(8)      q->Status.Type = 'ready';
(9)      q->Status.List = RL;
(10)     insert(RL, q); }
(11)   Scheduler();
(12) }
```

The operation first removes the resource from the process resource list. Then, if there is no process blocked on this resource (i.e., the *Waiting_List* is empty), the operation changes the status of the named resource to *free*. Otherwise, the process at the head the *Waiting_List* is removed, its status is changed to ready, and it is inserted on the RL. The scheduler is then called.

2.5 The Scheduler

We use a preemptive multilevel priority scheduler with fixed-priority levels. That means, the RL of processes maintained by the scheduler consists of n levels. Each level contains a queue of zero or more processes (pointers to their PCBs). Processes at level 0 have the lowest priority; processes at level $n - 1$ have the highest priority. The priority is assigned to a process at the time of its creation and remains unchanged for the duration of its lifetime. When a new process is created or an existing process is unblocked, it is inserted at the end of the list of processes for its priority level.

The scheduler services the RL in FIFO order, starting with the highest-priority level. That means, when asked to find the highest-priority ready process, it selects the process at the head of the highest nonempty queue. Note that this scheduling discipline can easily lead to starvation. A process at level q will get a chance to run only when all processes at levels higher than q have either terminated or blocked on a resource.

The organization of the scheduler is simplified if we know that there is always at least one process in the system ready to run. To guarantee this condition, we implement a special process, called *Init*. This process is created automatically when the system is started. It is assigned the lowest priority (0). The *Init* process plays two important roles. First, it acts as a "dummy" process that runs when no other process is ready to run. To ensure that, the *Init* process is not allowed to request any resources to prevent it from ever becoming blocked. Under this rule, the RL never becomes completely empty. Second, it is the very first process that runs when the system is started. It has no parent, but it may create other processes at higher priorities, which, in turn, may create other processes, and so on. Thus, the *Init* process is the root of the process creation tree.

The general structure of a priority scheduler was presented in Section 5.1.2 (Chapter 5). For this project, we simplify this as follows:

```
(1) Scheduler() {
(2)   find highest priority process p
(3)   if (self->priority < p->priority ||
(4)       self->Status.Type != 'running' ||
```

```
(5)       self == NIL)
(5)       preempt(p, self)
(6) }
```

The scheduler function is called at the end of any kernel call. Note that it is running as part of the current process, i.e., the process that issued the kernel call. The scheduler's task is to determine whether the current process continues to run or whether it should be preempted by another process. To make this decision, it finds the highest-priority ready process *p* (line 2). If *p*'s priority is higher than the priority of the current process, i.e., the processes within which the scheduler is running (line 3), *p* must preempt the current process (line 5).

Note that the condition (*self->priority* < *p->priority*) may be satisfied in two situations:

(1) When the scheduler is called at the end of a *Release* operation, the priority of the process unblocked as a result of the *Release* operation may be higher than the priority of the current process;

(2) When the scheduler is called at the end of a *Create* operation, the priority of the new process may be higher than the priority of the current process.

In both cases, the current process must be preempted.

There are two other situations under which the newly selected process must preempt the current process:

(1) When the scheduler is called from a *Request* operation and the requested resource is not available, the state of the current process has already been changed to *'blocked'*. (This change has been made on line 5 of the request operation.) When the scheduler is called from a *Timeout* operation (see Section 3), the status of the current process has been changed to *'ready'*. (This change has been made on line 4 of the timeout operation.) In both of these cases, i.e., when the status of the current process is not *'running'*, the current process must stop running and the highest-priority process *p* must be resumed in its place.

(2) When the scheduler is called from a *Destroy* operation where the current process named itself to be destroyed, its PCB no longer exists when it reaches the scheduler. The PCB has been deleted by the *Destroy* operation. As in the previous cases, the highest-priority process *p* must be resumed instead.

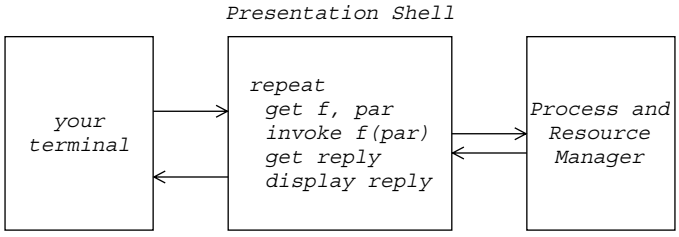
The preempt operation consists of the following tasks. It changes the the status of the selected process *p* to *'running'*. If the current process still exists and is not blocked, its status is changed to *'ready'* so that it would be resumed at a later time. Finally, the actual *context switch* takes place, which saves the state of the CPU (i.e., contents of all registers) in the PCB of the current process as the *CPU_State* and loads the *CPU_State* of process *p*.

In this project, we do not have access to the physical CPU to save and restore its registers. Thus, the task of the context switch is to display only on your terminal the name of the process that would now be running. At that point, the user terminal begins to play the role of the currently running process, as we describe next.

2.6 The Presentation Shell

If the process and resource manager described in the previous section were deployed in a real system, the kernel calls would be invoked by the currently running process. At the end of kernel call, the scheduler would either preempt the current process or resume its execution.

To be able to test and demonstrate the functionality of our manager, we develop a presentation shell, which repeatedly accepts commands from your terminal, invokes the corresponding function of the manager, and displays a reply message on the screen. The following diagram illustrates the basic structure:



Using the above organization, your terminal represents the currently running process. That means, whenever you type in a command, it is interpreted as if the currently running process issued this command. The presentation shell invokes the corresponding function, f, of the process and resource manager, and passes to it any parameters (*par*) supplied from the terminal. The function invocation results in some changes in the PCB, RCB, and other data structures. When the execution reaches the scheduler, it decides which process is to execute next and makes the appropriate changes to the status fields of the affected processes. Then, instead of saving and loading the CPU state (since there is no actual CPU), the scheduler only displays a message on the screen, informing you of any changes in the system state. In particular, it will always tell you which process is currently running, i.e., which process your terminal and keyboard are currently representing. The function (f) also may return a value, e.g., an error code, which the shell also displays on your screen.

To simplify the interactions with the shell, we must devise a syntax similar to the UNIX shell, where each line starts with a short command name, followed by optional parameters separated by blanks. The shell interprets each line and invokes the corresponding manager function. For example, the command line *"cr A 1"* could be used to invoke the function *Create(A, 1)*, which creates a new process named A at the priority level 1. Similarly, *"rq R"* would result in the call *Request(R)*.

The following sequence illustrates the interaction with the shell. Lines preceded by the asterisk (*) are responses displayed by the shell; lines preceded by the prompt sign (>) are commands typed in:

```
...
* Process A is running
> cr B 2
* Process B is running
> cr C 1
```

```
* Process B is running
> req R1
* Process B is blocked; Process A is running
...
```

We assume that a process A with a priority 1 already exists and is currently running. You type in the command *"cr B 2"*, which is interpreted by the shell as "A creates a new process named B with a priority 2." The shell invokes the *Create* operation with the appropriate parameters, which makes all the appropriate changes to the data structures. Since B's priority is higher than A's, B is selected by the scheduler to run next; the message "Process B is running" informs you that you are now acting on behalf of process B. Thus, the next command, *"cr C 1"*, is interpreted as process B creating a new process C with priority 1. Since C's priority is lower than B's, B continues running. The next command is a request (by B) to get the resource R1. Assuming R1 is currently not available, process B blocks and A, which is currently the highest-priority process, resumes its execution.

3 EXTENDED PROCESS AND RESOURCE MANAGER

In this section we extend the basic process and resource manager described in Section 2 to incorporate some aspects of the hardware. Notably, the extended manager will be able to handle hardware interrupts coming from a timeout generator and from I/O completion.

3.1 Timeout Interrupts

To implement time-sharing, assume that a hardware timer is provided, which periodically issues a timeout interrupt. This invokes a function, *Timeout()*, of the extended manager. This function performs the following task:

```
(1) Timeout() {
(2)   find running process q;
(3)   remove(RL, q);
(4)   q->Status.Type = 'ready';
(5)   insert(RL, q)
(6)   find highest priority process p;
(6)   p->Status.Type = 'running';
(7)   Scheduler();
(8) }
```

The function removes the currently running process *q* from the head of the RL, changes its status from *'running'* to *'ready'*, and inserts it back at the end of the RL. It then changes the status of the currently highest-priority process *p* from *'ready'* to *'running'*. Finally, the scheduler is called. We assume that *Timeout* runs as part of a special low-priority process. Thus, when the scheduler compares the priorities of *p* and *self* (line 3 of the *Scheduler* function), it starts the process *p*.

Note that this function has no effect if there is only one process at that level. If there are multiple processes at that level, then repeated invocations of *Timeout* cycle through all processes in the queue in a RR fashion.

3.2 Input/Output Processing

We represent all I/O devices collectively as a resource named *IO*; the RCB of this resource has the following form:

IO
Waiting_List

This is analogous to the RCBs of other resources, except that the status field is omitted. A process wishing to perform I/O issues a request for the *IO* resource. The request has the following form:

```
(1) Request_IO() {
(2)   self->Status.Type = 'blocked';
(3)   self->Status.List = IO;
(4)   remove(Ready List, self);
(4)   insert(IO->Waiting_List, self);
(5)   Scheduler();
(6) }
```

Normally the caller would specify the details of the I/O request as parameters to the call; we will omit this in this project, since there are no actual I/O devices employed.

The *Request_IO* function blocks the calling process and moves its PCB from the RL to the tail of the waiting list associated with the IO resource. The scheduler is called to select another process to continue running.

A process blocked on I/O remains in the waiting list until an I/O interrupt is generated, indicating that the I/O request has been completed and the waiting process may be unblocked. We make the following simplifying assumption: all I/O requests complete in the order in which they were submitted (see Section 5, extension 3.) Under the simplifying assumption, the following function is invoked by an I/O completion interrupt:

```
(1) IO_completion() {
(2)   remove(IO->Waiting_List, p);
(3)   p->Status.Type = 'ready';
(4)   p->Status.List = RL;
(4)   insert(RL, p);
(5)   Scheduler();
(6) }
```

The function simply unblocks the first process on the waiting list of the IO resource and moves it to the RL. We assume again that *IO_completion* runs as part of a low-priority process such that the scheduler selects one of the other processes from the RL to run next.

3.3 The Extended Shell

In the extended form of the process and resource manager, the user terminal plays a dual role.

1. It continues representing the currently running process, as was the case with the basic manager version. Thus, typing any of the commands introduced in Section 2.6 will have the same effect as before. In addition, a process is now able to issue an additional call, *Request_IO()*.
2. The terminal also represents the hardware. Notably, the user is able to emulate the generation of the two types of interrupts—timeout and I/O completion—by triggering the corresponding functions of the manager.

To incorporate the above extensions into the shell, three new commands must be provided to invoke the functions *Request_IO()*, *IO_competion()*, and *Timeout()*. None of these functions require any parameters.

4 SUMMARY OF SPECIFIC TASKS

1. Design and implement the process and resource manager; the basic version includes the functions *Create(), Destroy(), Suspend(), Activate(), Request()*, and *Release()*, together with the underlying data structures; the extended version provides the additional functions *Request_IO(), IO_completion()*, and *Timeout()*.
2. Define a command language for the presentation shell. Then design and implement the shell so that you can test and demonstrate the functionality of your process and resource manager. For each command, the shell should respond by displaying the name of the currently running process, and any errors and other relevant system events that may have occurred.
3. Instantiate the manager to include the following at start-up:
   - **(a)** A RL with at least three priorities.
   - **(b)** A single process, called *Init*, that runs at the lowest-priority level and is always ready to run. This may create other processes at higher levels.
   - **(c)** At least three fixed resources, say A, B, C, that processes can request and release.
   - **(d)** An IO resource that processes may request and become blocked on until the next I/O completion interrupt.
4. Test the manager using a variety of command sequences to explore all aspects of its behavior. Demonstrate also a deadlock situation.

5 IDEAS FOR ADDITIONAL TASKS

1. Extend the concept of a resource to include multiple identical units. That is, each RCB represents a *class* of resources. Instead of the simple *Status* field, it has an "*inventory*" field that keeps track of which or how many units of the given resource are currently available. A process then may specify how many units of a resource class it needs. The request is granted when enough units are available; otherwise, the process is blocked. Similarly, a process may release some or all of the units it has previously acquired. This may now potentially unblock multiple processes at once.
2. Incorporate main memory as another resource class in the process and resource manager. That means, define a RCB for memory. The *Status* field is replaced by a

pointer to the list of holes, which represents the current inventory of the memory resource. Adapt and incorporate the memory *request* and *release* functions developed as part of Project 3 (Main Memory). Processes may request new memory blocks by specifying their size and later release them.

3. Remove the restriction that all I/O requests will complete in the order of their submission. This requires keeping track of which I/O completion interrupt corresponds to which I/O request, which a real system must do. In this project, the *Request_IO()* must be extended to record the ID of the calling process. The *IO_completion()* function then specifies the process whose request has been satisfied.