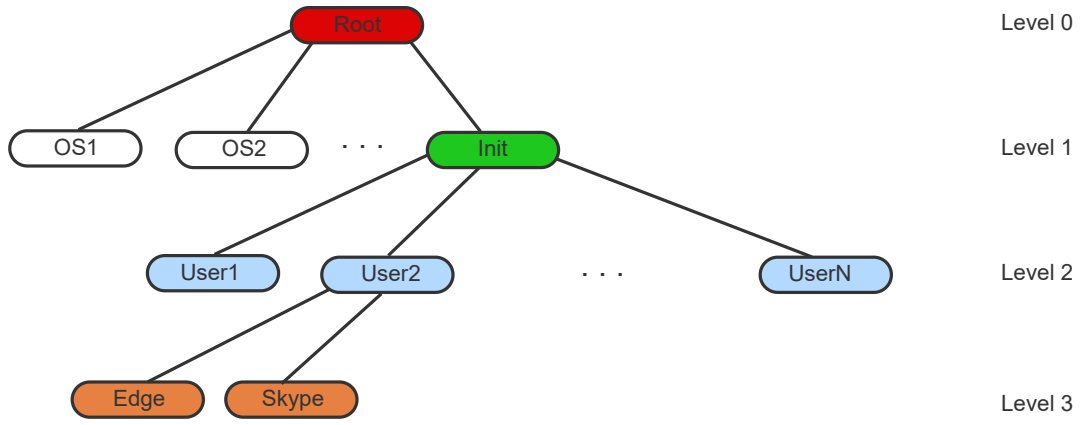
2.4 Operations on processes

The process creation hierarchy

A ***process creation hierarchy*** is a graphical representation of the dynamically changing parent-child relationships among all processes. The process creation hierarchy changes each time a process is created or destroyed.

PARTICIPATION ACTIVITY 2.4.1: The process creation hierarchy.

Start 2x speed



Captions

1. When the OS starts, a Root process is created (Level 0), which becomes the parent of other OS processes (Level 1).
2. One of the OS processes becomes the highest ancestor of all user processes. Ex: In Unix this ancestor process is called Init.
3. Whenever a new user logs in, a new user process is created (Level 2). The main task of each user process is to communicate with the user.
4. Each User process can itself create child processes to perform various tasks (Level 3). Ex: A browser (Edge), a communication tool (Skype), or a scientific simulator (Weather).
5. Each child processes can in turn create other child processes (Level 4). Ex: Weather could create processes to perform independent sub-computations in parallel.
6. When a process is destroyed, the question arises: What to do with the destroyed process's children?
7. Some OSs (Ex: Unix) make the Init process the new parent of the deleted process's children.
8. Other OSs choose to destroy the process as well as all of the process's descendants.

Feedback?

PARTICIPATION ACTIVITY 2.4.2: Number of parents and children.

1) A process at level i in the process creation hierarchy can have _____ link(s) to processes at the next higher level i-1.

Correct

1

1

Check Show answer

Except for Root, every process has exactly one parent process at the next higher level.

2) A newly created process (still in the state new) at level i can have _____ links to processes at level i+1.

0

Check Show answer

Correct

0

Child processes are created at runtime, not when a new process is established.

Feedback?

PARTICIPATION ACTIVITY 2.4.3: Link direction in the process creation hierarchy.

1) Each link in the process creation hierarchy corresponds to a _____.

- pointer directed toward the Root
- pointer directed away from the Root
- bidirectional pointer

Correct

Both directions must be represented so that any process can reach the parent and vice versa.

Feedback?

Process creation

When the currently running process initiates the creation of a child process, the OS executes a create process function, create(). The ***create process function*** allocates a new PCB, fills the PCB entries with initial values, and links the PCB to other data structures in the system:

1. A new PCB is allocated. The PCB is uniquely identified by a pointer or an array index p.
2. The fields cpu_state, memory, scheduling_information, and accounting_information are filled using the initial values supplied to the function as parameters.
3. Assuming the state transition diagram consists of only the three states, running, ready, and blocked, the process state field is set to ready.
4. The parent field of p is set to point to self, which is the calling process and thus the parent of p.
5. In turn, p is inserted into the calling process's list of children as a new child.
6. The remaining fields are set to NULL, since at creation p has no children, no open files, and no other resources.
7. p is inserted into the RL.
8. The scheduler function is called to select the process to run. Depending on the priorities of the two processes, the scheduler could choose to start the new child process p or to continue the execution of p's parent process.

Figure 2.4.1: The create process function.

```
create(state0, mem0, sched0, acc0) {
   p = allocate new PCB
   p.cpu_state = state0
   p.memory = mem0
   p.scheduling_information = sched0
   p.accounting_information = acc0
   p.process_state = ready
   p.parent = self
   insert p into self.children
   p.children = NULL
   p.open_files = NULL
   p.other_resources = NULL
   insert p into RL
   scheduler()
}
```

Feedback?

PARTICIPATION ACTIVITY 2.4.4: Filling the PCB data structure.

The create() function fills the various fields of p's PCB from different sources. Identify the correct source for each given sample field.

Select the definition that matches each term

1) input parameter
   - (•) p.memory

Correct
The initial value for p.memory is supplied as a parameter to create(), along with p.cpu_state, p.scheduling_information, and p.accounting_information.

2) constant value
   - (•) p.process_state

Correct
The initial process state is either ready or new, depending on which state the OS supports.

3) pointer or index
   - (•) p.parent

Correct
p.parent is made to point to the PCB of the calling processes.

4) NULL
   - (•) p.open_files

Correct
No files are open at the time of creation. The field p.open_files is set to NULL along with p.children and p.other_resources.

Reset

Feedback?

PARTICIPATION ACTIVITY 2.4.5: Managing a new PCB.

1) The initial CPU state of a newly created process is NULL or undefined.
   - ( ) True
   - (•) False

Correct
The program counter must be set to the starting address. Also, the stack pointers must be set to point to the top of the stack. Other registers and flags are generally set to 0.

2) The initial value of the process_state field could be set

Correct

to blocked if memory or some other critical resource is unavailable.

- ○ True
- ◉ False

A process enters the blocked state only when requesting an unavailable resource during execution. When a resource is unavailable at creation, the OS would postpone or decline the creation.

3) When a new process p is created, the scheduler could select p to run next.

- ◉ True
- ○ False

Correct

If p's priority is higher than or equal to the priority of p's parent, then p could be chosen by the scheduler.

Feedback?

Process destruction

A process p can be terminated by the parent process by calling the function destroy(p). Completing all work or committing a fatal error also results in the process's destruction. The ***destroy process function*** destroys a process by freeing the PCB data structure and removing any references to the PCB from the system.

Depending on the OS, the destroy function may also destroy all of the process's descendants to prevent having "orphan" processes in the system. The destruction of the entire hierarchy of descendants is accomplished by calling destroy(c) recursively on all children c of p.

The destroy() function performs the following steps:

1. After calling destroy(c) on all child processes, remove p from either the RL (when p is ready) or from the waiting list of a resource (when p is blocked).
2. Remove p from the list of children of the calling process.
3. Release all memory and other resources, close all open files, and deallocate the PCB.
4. Call the scheduler to select the next process to run. The call must be made outside of the destroy(p) function to guarantee that the scheduler executes only once, after the entire hierarchy of processes has been destroyed, rather than as part of every recursive call to destroy(c).

Figure 2.4.2: The destroy process function.

```
destroy(p) {
   for all c in p.children destroy(c)
   remove p from RL or waiting list
   remove p from parent's list of children
   release p.memory
   release p.other_resources
   close all p.open_files
   deallocate PCB
}
scheduler()
```

Feedback?

PARTICIPATION ACTIVITY 2.4.6: Descendants of a destroyed process.

When a process p is destroyed, the following choices can be made for p's children:

1) The children are recursively destroyed along with p.

- ◉ Beneficial
- ○ Detrimental

Correct

This solution prevents the creation of any orphan processes.

2) The children are assigned to a specialized OS process (Ex: the init process in Unix) as the new parent.
   - (•) Beneficial
   - ( ) Detrimental

Correct

This solution allows the child processes to continue executing while still under the control of another process responsible for their eventual termination.

3) The children could be left without any parent.
   - ( ) Beneficial
   - (•) Detrimental

Correct

This solution could result in runaway processes (Ex: infinite loops), which would continue using resources but would be hard to detect and eliminate.

Feedback?

PARTICIPATION ACTIVITY 2.4.7: Implementing the destroy function.

1) The responsibility to close all files should rest with the programmer, not the destroy function.
   - ( ) True
   - (•) False

Correct

A process can be destroyed at unpredictable times and the programmer has no way of knowing when files would need to be closed (to prevent any loss of data not yet saved to disk).

2) Any process is either on the RL and in the ready or running state, or on a resource's waiting list and in the blocked state. To efficiently find on which lists p resides, additional information must be kept in the PCB.
   - (•) True
   - ( ) False

Correct

Only one RL, but potentially many waiting lists, exist in the system. To find the relevant list quickly, the PCB must maintain a pointer to the current list.

Feedback?

PARTICIPATION ACTIVITY 2.4.8: Destroying self.

1) What would be the most likely outcome if a process called destroy(self):
   - ( ) The process and all descendants would terminate safely.
   - ( ) The function would destroy all descendants but not the process itself.
   - (•) The process would crash.

Correct

The destroy function is executing as part of the current process. Releasing all memory, resources, PCB, and closing all files would prevent the process from completing the entire destroy function.

Feedback?

CHALLENGE ACTIVITY 2.4.1: Operations on processes.

371440.2479476.qx3zqy7

Jump to level 1

1 2 3

Assume the following operations are executed.

p1 creates p2, p3, p4
p4 creates p5
p2 creates p6, p7
p7 creates p8, p9

Which processes are destroyed by the operation `destroy(p4)`?

- [x] p1
- [ ] p2
- [ ] p3
- [x] p4
- [ ] p5
- [ ] p6
- [ ] p7

Which processes are destroyed by the operation `destroy(p6)`?

- [ ] p1
- [ ] p2
- [ ] p3
- [x] p4
- [ ] p5
- [ ] p6
- [ ] p7

1 2 **3**

Check Next

✗ Each incorrect answer is highlighted.
Expected:
`destroy(p4)` destroys p4, p5
`destroy(p6)` destroys p6

`destroy(p4)` destroys all processes descending from p4 along with p4.
p4 has child process p5.

`destroy(p6)` destroys all processes descending from p6 along with p6.
p6 does not have any child processes, so only p6 is destroyed.

Feedback?

EXERCISE | 2.4.1: The suspend and activate functions.

The following pseudo code implements the suspend() and activate() functions. Two new states are introduced, suspended_ready and suspended_blocked, to keep track of the state in which a process was suspended. That is, a ready process moves to the suspended_ready state by the suspend function. Similarly, a blocked process moves to the suspended_blocked state by the suspend function. The activate function reverses the transitions.

```
suspend(p) {
   if (p.process_state == blocked)
      p.process_state = suspended_blocked
      else p.process_state = suspended_ready
}

activate(p) {
   if (p.process_state == suspended_ready)
      p.process_state = ready
      else p.process_state = blocked
   scheduler()
}
```

(a) What changes must be made to the scheduler or other functions to make suspend/activate work correctly?

Solution

(b) Why is the scheduler called only in activate but not in suspend?

Solution

(c) A process must be prevented from calling suspend() or activate() on itself. Why?

Solution

(d) A process must be prevented from calling suspend() on an already suspended process, or calling activate() on a currently active (ready) process. Why?

Solution

Feedback?

How was this section? **Provide feedback**