2.3 The process control block

Contents of the PCB

The PCB is the instantiation of a process. Upon creation, the OS assigns every process a unique identifier. This identifier, p, could be a pointer to the PCB structure or an index into an array of PCBs.

The specific implementation and the contents of a PCB vary between different OSs but the following is a generic structure representative of most modern OSs.

Table 2.3.1: A generic PCB of process p.

PCB field	Explanation
CPU_state	When p is stopped, the current state of the CPU, consisting of various hardware registers and flags, is saved in this field. The save information is copied back to the CPU when p resumes execution.
process_state	Stores p's current state. Ex: Running, ready, or blocked.
memory	Describes the area of memory assigned to p. In the simplest case the field would point to a contiguous area of main memory. In systems using virtual memory (to be introduced in a later chapter) the field could point to a hierarchy of memory pages or segments.
scheduling_information	Contains information used by the scheduler to decide when p should run. The information typically records p's CPU time, the real time in the system, the priority, and any possible deadlines.
accounting_information	Keeps track of information necessary for accounting and billing purposes. Ex: The amount of CPU time or memory used.
open_files	Keeps track of the files currently open by p.
other_resources	Keeps track of any resources, such as printers, that p has requested and successfully acquired.
parent	Every process is created by some other running process. The ***parent process*** of a process p is the process that created p. The parent field records the identity of p's parent.
children	A ***child process*** c of process p is a process created by p. Process p is c's parent. The identity of every child process c of p is recorded in the children field.

Feedback?

PARTICIPATION ACTIVITY | 2.3.1: Possible changes to PCB.

Indicate which PCB fields may change during a process's lifetime.

1) CPU_state

May change (selected)
Will not change

Correct

The CPU_state changes during every context switch.

2) process_state

May change (selected)
Will not change

Correct

The process_state keeps changing as the process requests resources or is moved to/from the CPU

3) open_files

May change (selected)
Will not change

Correct

The process can open and close any number of files during execution.

4) parent

May change
Will not change (selected)

Correct

The parent is the creator of the current process and by definition cannot change.

5) children

May change (selected)
Will not change

Correct

The process can create and destroy multiple children at run time.

Feedback?

PARTICIPATION ACTIVITY | 2.3.2: Runtime changes to PCB.

Indicate which PCB fields may change while a process is in the running state.

1) CPU_state

May change
Will not change (selected)

Correct

The CPU_state is copied into the CPU when the process starts running and is not updated until the process stops again.

2) process_state

May change
Will not change (selected)

Correct

The process_state is set to "running" when the process starts running and changes to "ready" or "blocked" only when the process stops running.

3) open_files

May change (selected)
Will not change

Correct

The process can open and close any number of files during execution.

4) parent

May change
Will not change (selected)

Correct

The parent is the creator of the current process and by definition cannot change.

5) children

May change (selected)
Will not change

Correct

The process can create and destroy multiple children at run time.

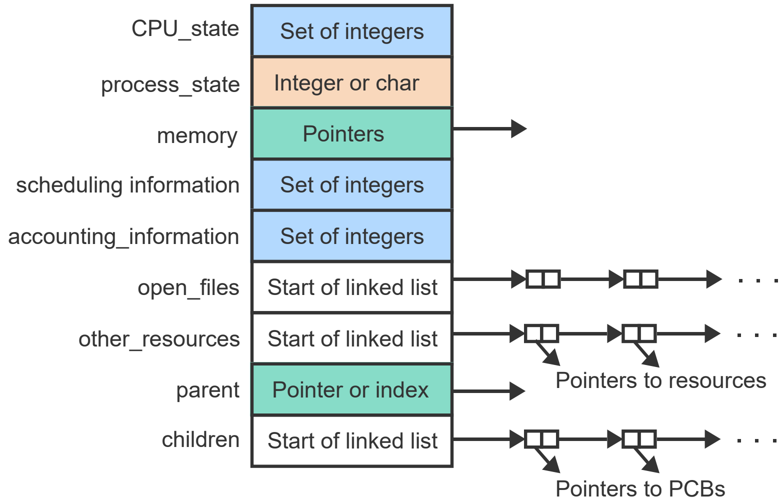
Feedback?

The PCB data structure

The entries of a PCB are composed of different data types, including integers, characters, or pointers. Consequently, each PCB is implemented as a heterogeneous data structure. Ex: "struct" in the C language.

PARTICIPATION ACTIVITY | 2.3.3: Composition of the PCB.

Start 2x speed



Captions

1. An empty PCB structure is created for a new process.
2. Some fields are represented as sets of integers. Ex: The CPU_state contains copies of CPU registers. The scheduling_information has the priority and other data.
3. The process_state encodes the possible states, such as ready, running, or blocked, using most likely integers or characters.
4. The memory field contains one or more pointers (memory address) and the parent field designates the parent process using a pointer or array index.
5. The open_files field is a pointer to a linked list of open files where each list element points to one open file.
6. The remaining two fields are also pointers to linked lists. One is the list of other resources held by the process and the other is a list of child processes.

Feedback?

PARTICIPATION ACTIVITY | 2.3.4: Minimizing the process_state field.

1) What would be the most space-efficient data type to represent the process_state in C?

char

Check Show answer

Correct

char

Each character is a single byte or 8 bits and can represent up to 2^8 different states.

2) What is the minimum number of bits needed to represent the process_state field, if 3 states are supported: running, ready, and blocked?

2

Check Show answer

Correct

2

2 bits can represent up to 4 states, which would be sufficient for the states running, ready, and blocked.

3) What is the minimum number of

bits needed to represent the process_state field, if 6 states are supported: running, ready, blocked, new, suspended, and terminated?

3

Check Show answer

Correct

3

3 bits can represent up to 8 states, which would be sufficient for the states running, ready, blocked, new, suspended, and terminated.

Feedback?

PARTICIPATION ACTIVITY | 2.3.5: Reducing the PCB.

1) The PCB data structure could be reduced if a process could generate only one child at a time.

- True
- False

Correct

The children field is a pointer to the first child in the list. The restriction would eliminate dynamic memory management but the same pointer to the first child process would still be needed.

Feedback?

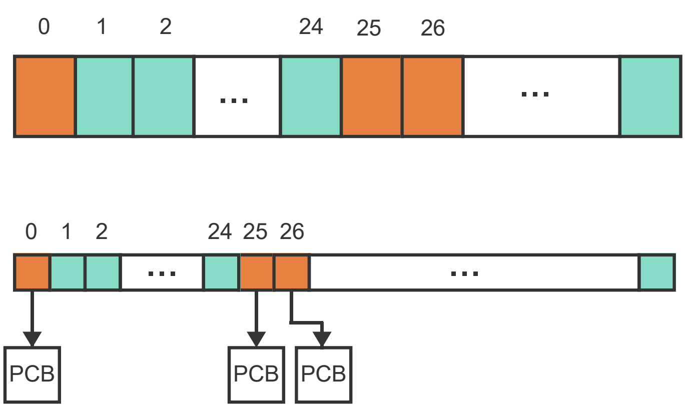
Organizing PCBs

The OS must allocate and deallocate PCBs efficiently as processes are created and destroyed. Two ways exist to organize all PCBs:

1. An array of structures. The PCBs are marked as free or allocated, which eliminates the need for any dynamic memory management. The main drawback is a lot of wasted memory space to maintain a sufficient number of PCB slots.
2. An array of pointers to dynamically allocated PCBs. The pointer array wastes little space and can be made much larger than the array of structures. The drawback is the overhead of dynamic memory management to allocate each new PCB and to free the memory when the process terminates.

PARTICIPATION ACTIVITY | 2.3.6: Allocating and deallocating PCBs.

Start 2x speed



Captions

1. With an array of structures, each slot contains either a PCB (orange) or is marked as free (green).
2. Allocating a new PCB involves selecting a free slot (26) and filling the new PCB's entries. No dynamic memory allocation takes place.

3. With an array of PCB pointers, each entry points to a PCB structure (orange) or is marked as free (green).
4. To allocate a new PCB requires first the dynamic creation of the new data structure. Next, a free slot in the array is located (26) and made to point to the new PCB.
5. To destroy a PCB in the array implementation, the array slot is simply marked as free.
6. To destroy a PCB with the array of PCB pointers, the PCB structure must be freed and the corresponding pointer entry marked as free.

Feedback?

PARTICIPATION ACTIVITY 2.3.7: PCB implementation options.

1) The set of all PCBs could be implemented as a _______

- ○ 2-dimensional integer array, where the first dimension is the number of available PCB slots.
- ◉ 1-dimensional array of structures, where the array dimension is the number of available PCB slots.
- ○ linked list of 1-dimensional integer arrays, where each list element is a PCB.

Correct

The fields of a PCB are of different data types and thus each PCB can be represented by a structure.

Feedback?

PARTICIPATION ACTIVITY 2.3.8: Reducing overhead of PCB management.

1) When PCBs are allocated and freed dynamically, the overhead of the dynamic memory management could be reduced by reusing PCBs of destroyed processes.

- ◉ True
- ○ False

Correct

A PCB of a destroyed process could be reused by marking the slot in the pointer array with a new state (for example, "available"), instead of freeing the PCB data structure and setting the pointer to NULL.

Feedback?

PARTICIPATION ACTIVITY 2.3.9: Space-efficiency in PCB reuse.

1) Reusing PCBs of destroyed processes rather than freeing the data structures is more time-efficient _________.

- ○ and also more space-efficient
- ○ and equally space-efficient
- ◉ but less space-efficient

Correct

The PCBs are never freed and thus the total space required corresponds to the maximum number of PCBs ever existing concurrently, rather than just the average.

Feedback?

Avoiding linked lists

Linked lists require dynamic memory management, which is costly. The Linux OS has pioneered an approach that eliminates this overhead.

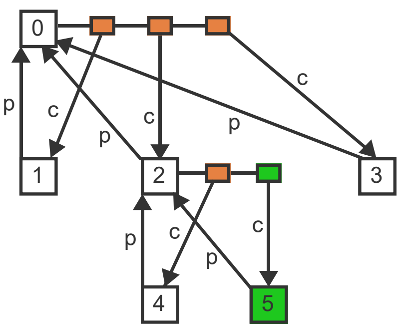
Instead of a separate linked list anchored in the parent's PCB, the links are distributed over the child PCBs such that each points to the immediate younger sibling and immediate older sibling. The original 2 fields, parent and children, in the PCB of a process p are replaced by 4 new fields:

- parent: points to p's single parent as before
- first_child: points to p's first child
- younger_sibling: points to the sibling of p created immediately following p
- older_sibling: points to the sibling of p created immediately prior to p

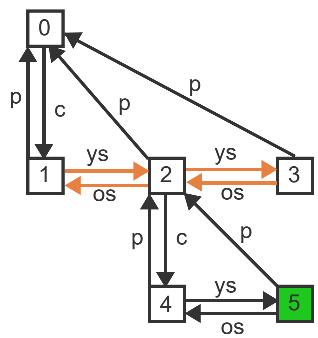
This implementation requires no dynamic memory management.

PARTICIPATION ACTIVITY | 2.3.10: Avoiding a linked list implementation.

Start 2x speed



Implementation using linked lists



Implementation without linked lists

Captions

1. Process 0 has 3 children (1, 2, 3). A child pointer (c) points from 0's linked list to each child. Each child has a parent pointer (p) to 0. Process 2 has 1 child (4) linked to 2 in an analogous manner.
2. In the list-free implementation, all children point to the parent as before (p) but the parent points only to the first child (c)
3. Instead of the separate linked list, all child processes are linked to each other using the younger sibling (ys) and older sibling (os) links.
4. To add a new child under the linked-list implementation, a new link is dynamically created to point to the new PCB (5) and the PCB is linked to the parent (2).
5. To add a new child (5) under the list-free implementation, only the older sibling (4) is modified to point to the new younger sibling (5). No dynamic memory allocation takes place.

Feedback?

PARTICIPATION ACTIVITY | 2.3.11: Representing parent/child relationships.

1) The list-free implementation requires _____ fields in the PCB instead of 2 to represent the parent/child relationships.

4

Correct

4

The original representation had the fields parent and children. The list-free implementation has the fields

Check Show answer

parent, child, older sibling, and younger sibling.

Feedback?

PARTICIPATION ACTIVITY | 2.3.12: Efficiency of avoiding linked lists.

1) The list-free implementation is _______ the linked-list implementation.

- (•) more time-efficient than
- ( ) less time-efficient than
- ( ) equally time-efficient as

Correct

The main reason for the list-free implementation was to improve time-efficiency by avoiding the dynamic memory management needed for the linked lists.

2) Adding a new child process with the list-free implementation uses _____ the linked-list implementation.

- ( ) less memory than
- ( ) more memory than
- (•) the same amount of memory as

Correct

Two pointers are required per child with either implementation. With a linked-list, the 2 pointers are in a linked-list node. With the list-free implementation, the 2 pointers are added to the PCB. So the memory usage is the same.

Feedback?

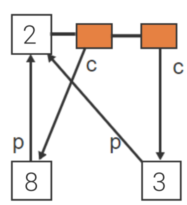
CHALLENGE ACTIVITY | 2.3.1: Avoiding linked lists.

371440.2479476.qx3zqy7

Start

1 2 3 4 5

The following diagram shows the parent-child relationships among three processes.



Linked lists can be avoided by implemer
additional pointers, younger sibling (ys)
sibling (os), in each PCB.

Complete the table by showing which field points to which process using the four different poi
an entry is unknown, enter -.

	2	8	3
p	Ex: 10		
c			
ys			
os			

1 2 3 4 5

Check Next

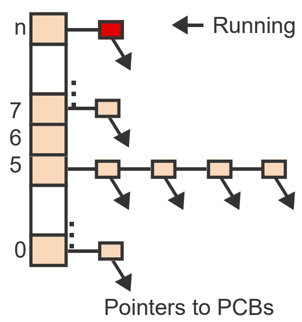
Feedback?

Managing PCBs

The OS maintains all PCBs organized on various lists. A ***waiting list*** is associated with every resource and contains all processes blocked on that resource because the resource is not available. Another important list is the ***ready list*** (***RL***): A list containing all processes that are in the ready state and thus are able to run on the CPU. The RL also includes the currently running process. The RL maintains all processes sorted by their importance, which is expressed by an integer value called the priority. The RL can be a simple linked list where the priority of a process is the current position in the list. The RL can also maintain processes in different lists sorted by priorities.

PARTICIPATION ACTIVITY | 2.3.13: RL as a priority list.

Start 2x speed



Captions

1. An RL consisting of priorities 0 through n currently contains processes at levels 0, 5, and 7. The highest priority process (level 7) is running.
2. A new process at priority 5 is entered into the RL. Process at level 7 continues to run.
3. When a new process is entered at a priority above 7 then the currently running process (level 7) turns from running to ready and the new process starts to run.

Feedback?

PARTICIPATION ACTIVITY | 2.3.14: Ranges of priorities.

With RL implemented as a priority list, the range of priorities is [n1:n2], where n1 and n2 can be:

1) n1 = 0, n2 > 0

- (selected) True
- False

Correct

The most common choice is to start the range with n1 = 0 and chose n2 to provide the desired number of priority levels. Ex: [0:7] provides 8 priority levels.

2) n1 = 0, n2 = 0

- (selected) True
- False

Correct

The range can be [0:0], in which case the priority list degenerates to a single linked list.

3) n1 < 0, n2 > 0

- (selected) True
- False

Correct

Depending on the implementation's needs, the range can start and end with any integer, including negative integers. Ex: [-5:5] provides 11 priority levels, where the positive range and the negative range could be given special significance, such as differentiate between interactive and background processes.

Feedback?

PARTICIPATION ACTIVITY | 2.3.15: RL for multiple CPUs.

1) A single RL could not be used with multiple CPUs.

- (selected) True
- False

Incorrect

Starting with the highest-priority non-empty list, multiple processes can be marked as running in the RL. Ex: With 2 CPUs, the processes at level n and 7 in the above example could both be in the running state.

Feedback?

EXERCISE | 2.3.1: Creation hierarchy without linked lists.

Processes 0-4 are related as follows: 1, 2, 3 are children of 0, and 4 is a child of 2. PCBs are implemented as an array indexed by the process number. Each PCB has the links: parent (p), first child (c), younger sibling (ys), and older sibling (os).

(a) Complete the PCB array to show the values of the 4 links (p, c, ys, os) for all processes, to reflect the parent-child hierarchy.

	0	1	2	3	4	...
...	...	...	...	...	...	...
p						
c						
ys						
os						

Solution

The parent of process 0 is unknown (?).
A dash means no index. Ex: Process 1 has no child (c) and no older siblings (os).

	0	1	2	3	4	5	...
...	...	...	...	...	...	...	...
p	?	0	0	0	2		
c	1	-	4	-	-		
ys	-	2	3	-	-		
os	-	-	1	2	-		

(b) Modify the array to reflect the creation of a new child, 5, of process 2.

Solution

The new child is created at index 5.
Note that the parent process (2) is not modified in any way.

	0	1	2	3	4	5	...
...	...	...	...	...	...	...	...
p	?	0	0	0	2	2	
c	1	-	4	-	-	-	
ys	-	2	3	-	5	-	
os	-	-	1	2	-	4	

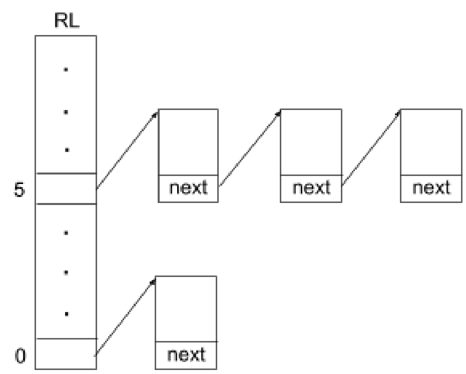
Feedback?

EXERCISE 2.3.2: RL without linked lists.

The RL can be implemented without dynamically managed linked lists by creating a new field, next, in each PCB, which points to the next PCB on the same list. Each entry of the RL then points to the first PCB on the list.

(a) Assume that RL contains 3 processes at level 5 and 1 process at level 0. Draw a diagram showing the RL and the modified PCBs.

Solution



Feedback?

EXERCISE 2.3.3: A modified implementation of PCBs.

Implementing the PCBs as an array is more efficient than using dynamic memory allocation. The drawback is that the array size must be kept relatively small. To overcome this drawback, the PCBs are implemented as fixed size array, PCB[n], where n is large enough most of the time. In the case where more processes need to be created, the array size n is temporarily extended to accommodate the spike. The extension is removed when the number of elements falls again below n. To compare the effectiveness of this scheme with a dynamic linked list implementation, assume the following values:

- s is the time to perform one insert or remove operation in a linked list
- r is the time to perform one insert or remove operation in the array
- o is the overhead time to temporarily extend the array
- p is the probability that any given insert operation will overrun the normal array size n

(a) Derive a formula for computing the value of p, below which the proposed scheme will outperform the linked list implementation.

Solution

- With the linked list implementation, each insert or remove takes s ms.
- With the array implementation, a remove takes r ms and each insert takes r + o*p ms since the overhead of extending the array occurs with probability p.
- Since half of all operations are inserts and half are removes, the time for one operation is (r + r + o*p)/2.
- The break-even point is when the time for the implementations is equal: s = (r + r + o*p)/2.
- Solving for p yields the probability p = (2s - 2r)/o

(b) Compute the value of p when s = 10r and o = 100r.

Solution ^

p = (2*10r - 2r)100r = 18/100 = 0.18

Feedback?

EXERCISE 2.3.4: Trade-offs of PCB reuse.

Reusing PCBs of destroyed processes instead of freeing up the data structures eliminates the overhead of dynamic memory management but is likely to use more memory.

Assume the following:

- The number of processes to be created and destroyed over a period of time is 10,000.
- The average number of processes coexisting at any given time is 100.
- The maximum number of processes coexisting at any given time is 600.
- Allocating and freeing a PCB take the same amount of time.

(a) How many memory operations (allocate or free) will be performed without PCB reuse?

Solution ^

Each of the 10,000 processes will require 1 allocate and 1 free operation for a total of 2*10,000 = 20,000 memory operations.

(b) How many memory operations (allocate or free) will be performed with PCB reuse?

Solution ^

A maximum of 600 PCBs will be needed, all of which will be freed at the end of the run. The total is 2*600 = 1,200 memory operations.

(c) How many PCBs will coexist in memory, on average, during the entire run without PCB reuse?

Solution ^

Since 100 processes will coexist at any given time, 100 PCBs will reside in memory on average.

(d) How many PCBs will coexist in memory, on average, during the entire run with PCB reuse?

Solution ^

The maximum number of PCBs coexisting in memory will reach 600 at some point but the average depends one when this maximum is reached. Without additional information, the midpoint of the run must be assumed. During the first half of the run, the number will steadily rise from 0 to 600, resulting in an average of 600/2 = 300 PCBs. During the second half, the number of PCB will remain at the maximum of 600. Thus the average number of PCBs over the entire run is (300 + 600)/2 = 450.

(e) What could be done to reduce the much larger number of PCBs maintained in memory when PCBs are reused?

Solution

The OS could automatically free some of the unused PCBs, either periodically or when the number of unused PCBs exceeds some threshold.

Feedback?

How was this section?

Provide feedback