Supplement: Scheduling with Linked Lists

Leonard König, April 12, 2021

Linked Lists

In this exercise you are asked to select the next process to be scheduled from a list of given processes. This list may be arbitrarily long and thus an array won't suffice to hold it. Instead, we introduce a new data structure, called *linked list*. Spoken more precisely, the structure used is a *circular doubly-linked list*.

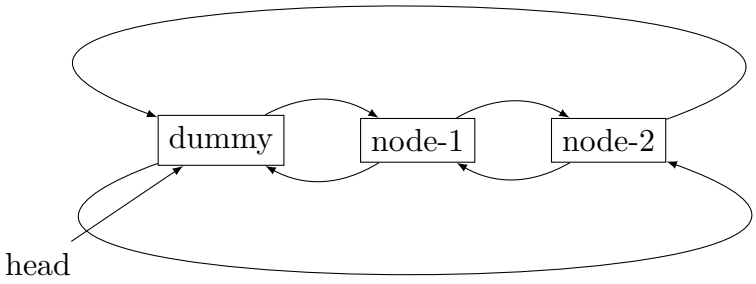

Figure 1 Linked List with two elements

In order to build a linked list of such nodes, we need to define a new type. This is already done by us, but, just for reference, a possible definition of a type named `struct node` for use in a doubly linked list is given in **Listing 2**.

```
struct node {
    struct node *next;
    struct node *prev;

    int value;
};
```

Listing 2 A node in a doubly linked list

However, this only defines a new type but doesn't define a new variable *of* this type. Again, for reference only, we give a definition of the dummy node.

```
struct process dummy = {
    .next = &dummy,
    .prev = &dummy,
    /* we don't care about the dummys value(s) */
};
```

Listing 3 Defining a linked list node

We need the dummy, as otherwise we wouldn't be able to model an empty list with our data structure (cf. **Listing 4**).

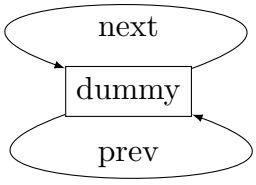

Figure 4 An empty list

When we later want to give this list to another function, we don't want to give it a copy of the dummy value, so we create a pointer to our dummy value instead and will call it 'head' as it *points to the head of the list*.

This will be the variable that is passed to your functions!

Iterating over Linked Lists

Luckily, in this exercise you are not asked to manage the linked list (i.e., add new elements), but only to iterate over its elements, finding the best candidate to schedule next. We provide you with a wrapper which simply calls your function (e.g., `spn()`) each time for *every tick* with the *current* list of processes. It is then the task of your function to select the process that is to run next by setting its `state` to `PS_RUNNING` and making sure that all other processes are `PS_READY` or `PS_DEAD` (no two processes can be running at the same time). You then end your function, the simulation will 'run' your process and your function will be called again.

We start by creating a pointer which will, after we iterated through the whole list, point to the process that is supposed to run. However, before iteration we haven't found such a process, thus it's reasonable to set it to `NULL`.

```
struct process *selected = NULL;
for (struct process *c = head->next;
        c != head;
        c = c->next) {
    //TODO: Update `selected'
}
```

Listing 5 Iterating over a linked list

We then setup an iteration pointer `c` which is initialized to the first element of our list (remember, `head` points to the dummy element). We want to terminate iteration, when `c` points to the same element as `head` points (i.e., the dummy), and after each iteration we update `c` to point to its following node.

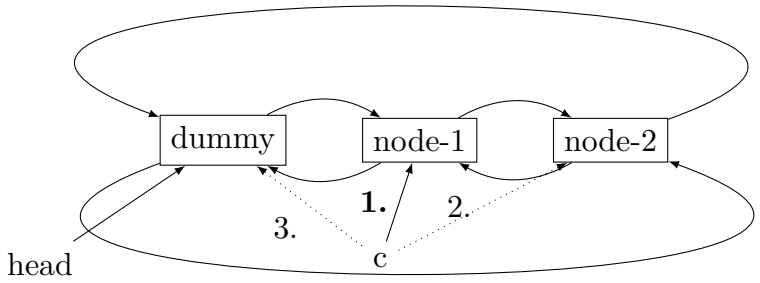

Figure 6 Iterating over a linked list

In order to find the process with the shortest runtime, we simply check, wether we have a currently selected process, and that its runtime (`cycles_todo`, `cycles_done`) is indeed shorter than the runtime of `c`. If not, we update `selected` to be `c`. As we go, we also set the currently running process back to be `PS_READY` or to `PS_DEAD` if it has no ticks left to run.

After iteration, we simply set the currently selected process to be running (this may be the same process as before), and exit our function.