An Example of Reasoning in Extended Pointer Logic: AIO Remove Request Function in Glibc

Hongjin Liang[1,3] Yu Zhang[1,3] Yiyun Chen[1,3] Zhaopeng Li[1,3] Baojian Hua[2,3]

[1]School of Computer Science, University of Science and Technology of China

[2]School of Software Engineering, University of Science and Technology of China

[3]Software Security Lab., Suzhou Institute for Advanced Study, University of Science and Technology of China

In subsection 3.2 of our paper *A Pointer Logic Dealing with Uncertain Equality of Pointers*[1], we show how the extended pointer logic can be used in practice by illustrating the proof of a function taken from GNU C Library[2]. This report gives details of the proof.

In GNU C Library, the asynchronous I/O operations are implemented using a request queue. The type of the request queue node is:

```
struct requestlist {
        int runnning;
        struct requestlist *last_fd;
        struct requestlist *next_fd;
        struct requestlist *next_prio;
        struct requestlist *next_run;
        // …
};
```

Every node holds an I/O request for a file descriptor. A doubly-linked list is formed by nodes in order of the file descriptors via the fields last_fd and next_fd. In the list, last_fd of the head node and next_fd of the tail node are all **NULL**. If there are several I/O requests for one file descriptor, then the corresponding nodes form a singly-linked list in order of priority via the field next_prio. The head nodes of these singly-linked lists are in the doubly-linked list. Obviously, equality of these pointers is certain. And then a singly-linked list via next_run is formed by some nodes in the request queue (especially, in the doubly-linked list) of which the corresponding requests are ready, so this singly-linked list is called ready queue. The equality between the pointers in ready queue and those in the doubly-linked list of request queue is uncertain.

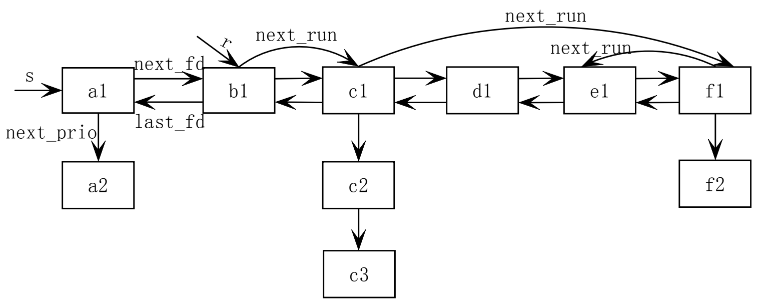

Figure 1. Request Queue and Ready Queue

Assume the numbers of nodes in request queue and ready queue are respectively n+1 and m+1, and s and r are respectively the head pointers of the queues (in **Figure 1**). The structure can be defined in the pointer logic as follows:

rq(s, r, n, m) = /* The doubly-linked list of request queue: */

($\forall$k: 0..n-1. [s(->next_fd)^{k}, s(->next_fd)$^{k+1}$->last_fd])

$\wedge$ [s(->next_fd)^{n}]

$\wedge$ {s->last_fd, s(->next_fd)$^{n+1}$}$_{N}$

/* Every node in the doubly-linked list links a singly-linked list: */

$\wedge$ ($\forall$k: 0..n. **plist**(s(->next_fd)^{k}->next_prio))

/* Ready queue: */

$\wedge$ **rlist**(r, m)

$\wedge$ ($\forall$k: 0..n. $\forall$l: 0..n. (k!=l $\Rightarrow$ s(->next_fd)^{k}$\notin$[s(->next_fd)^{l}, s(->next_fd)$^{l+1}$->last_fd]))

where

plist(p) = {p}$_{N}$ $\vee$ ({p} $\wedge$ **plist**(p->next_prio) $\wedge$ {p->next_fd, p->last_fd, p->next_run}$_{N}$)

rlist(r, m) = {r(->next_run)$^{m+1}$}$_{N}$ $\wedge$ ($\forall$k: 0..m. [r(->next_run)^{k}])

$\wedge$ ($\forall$k: 0..m. $\forall$l: 0..m. (k!=l $\Rightarrow$ r(->next_run)^{k}$\notin$[r(->next_run)^{l}]))

In the definition, ($\forall$k:0..n.$\forall$l:0..n. (k!=l $\Rightarrow$ s(->next_fd)^{k}$\notin$[s(->next_fd)^{l}, s(->next_fd)$^{l+1}$->last_fd]) is a non-membership assertion[1] which expresses the pointer fields next_fd of nodes in the doubly-linked list are not equal to each other. The non-membership assertion in the definition of **rlist**(r, m) has similar meanings. Due to the limited space, some **NULL** pointers are ignored, such as the fields next_run of nodes which are in the doubly-linked list but not in **rlist**(r, m), because they are not involved in the verification.

In the function

void __aio_remove_request(struct requestlist *last, struct requestlist *req, int all),

two global variables will be used:

```
struct requestlist *runlist;    // Ready queue
struct requestlist *requests;    // Request queue
```

So we have **rq**(requests, runlist, n, m). The numbers of nodes in the doubly-linked list of requests and in runlist are respectively *n*+1 and *m*+1.

The function has two pointer parameters. Intuitively, the pointer last points to the predecessor of the node pointed to by req in the singly-linked list. The possible relations between them can be classified into 7 cases (in **Figure 2**).

case1: (last = **NULL**) $\wedge$ ($\exists$i: 0..n-1.req = requests(->next_fd)^{i}) $\wedge$ ($\exists$j: 0..m. req = runlist(->next_run)^{j})

case2: (last = **NULL**) $\wedge$ (req = requests(->next_fd)^{n}) $\wedge$ ($\exists$j: 0..m. req = runlist(->next_run)^{j})

case3: (last = **NULL**) $\wedge$ ($\exists$i: 0..n-1. req = requests(->next_fd)^{i}) $\wedge$ req $\notin$ **rlist**(runlist, m)

case4: (last = **NULL**) $\wedge$ (req = requests(->next_fd)^{n}) $\wedge$ req $\notin$ **rlist**(runlist, m)

case5: ($\exists$i: 0..n-1. last = requests(->next_fd)^{i}) $\wedge$ (req = last->next_prio) $\wedge$ req $\notin$ **rlist**(runlist, m)

case6: (last = requests(->next_fd)^{n}) $\wedge$ (req = last->next_prio) $\wedge$ req $\notin$ **rlist**(runlist, m)

case7: ($\exists$i: 0..n. last = requests(->next_fd)^{i}(->next_prio)^{ip}) $\wedge$ (req = last->next_prio) $\wedge$ req $\notin$ **rlist**(runlist, m)

Figure 2. Relations between Pointer Parameters

0<=i<=n-1

$\wedge$ ($\forall$k: 0..i-1. [requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd])

$\wedge$ {requests(->next_fd)^{i}, requests(->next_fd)$^{i+1}$->last_fd, req, runlist(->next_run)^{j}}

$\wedge$ ($\forall$k: i+1..n-1. [requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd])

$\wedge$ [requests(->next_fd)^{n}]

$\wedge$ {requests->last_fd, requests(->next_fd)$^{n+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$

$\wedge$ ($\forall$k: 0..n. **plist**(requests(->next_fd)^{k}->next_prio))

$\wedge$ ($\forall$k: 0..j-1. [runlist(->next_run)^{k}]) $\wedge$ ($\forall$k: j+1..m. [runlist(->next_run)^{k}])

$\wedge$ ($\forall$k: 0..m. $\forall$l: 0..m. (k!=l $\Rightarrow$ runlist(->next_fd)$^{k}\notin$[runlist(->next_fd)^{l}]))

$\wedge$ ($\forall$k: 0..n. $\forall$l: 0..n. (k!=l $\Rightarrow$ requests(->next_fd)$^{k}\notin$[requests(->next_fd)^{l}, requests(->next_fd)$^{l+1}$->last_fd]))

Figure 3. Relations between Pointer Parameters

In case1 and case2, the node pointed to by req is in the doubly-linked list of requests and in runlist. In case3 and case4, the node pointed to by req is in the doubly-linked list but not in runlist. In other cases, the node pointed to by req is in a singly-linked list linked by a node in the doubly-linked list, thus it will not be in runlist. The pointer last points to the predecessor of the node pointed to by req. So case5, case6 and case7 are distinguished by which node last points to. In order to reason in the pointer logic, the definition of **rq**(requests, runlist, n, m) should be expanded to include the pointers req and last, as case1 (in **Figure 3**).

Here we only take case1 and case7 as examples to show the proof of the AIO Remove Request Function. Briefly, the function removes the node *N* pointed to by req in all the linked relations including request queue and ready queue. Yet the function does not free the node. If all=1, all the nodes linked by req in the singly-linked list will be removed along with the node *N*. Otherwise, if all=0, only the node *N* will be removed and its successor in the singly-linked list will take its place. Note that when the node *N* is in the doubly-linked list of requests and the singly-linked list linked by req is empty (that is, req->next_prio=**NULL**), removing the node *N* has the same effect as removing it when all=1.

Since it's unsupported in **PointerC**, a loop with a break statement in the original code is modified. We only show the code and assertions at main points of the function below:

{ (0<=i<=n-1

$\wedge$ ($\forall$k:0..i-1.[requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd])

$\wedge$ {requests(->next_fd)^{i}, requests(->next_fd)$^{i+1}$->last_fd, req, runlist(-> next_run)^{j}}

$\wedge$ ($\forall$k:i+1..n-1.[requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd])

$\wedge$ [requests(->next_fd)^{n}]

$\wedge$ {requests->last_fd, requests(->next_fd)$^{n+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$

$\wedge$ ($\forall$k:0..n.**plist**(requests(->next_fd)^{k}->next_prio))

$\wedge$ ($\forall$k:0..j-1.[runlist(->next_run)^{k}]) $\wedge$ ($\forall$k:j+1..m.[runlist(->next_run)^{k}])

$\wedge$ ($\forall$k:0..m. $\forall$l:0..m. (k!=l $\Rightarrow$ runlist(->next_fd)$^{k}\notin$[runlist(->next_fd)^{l}]))

$\wedge$ ($\forall$k:0..n. $\forall$l:0..n. (k!=l $\Rightarrow$ requests(->next_fd)$^{k}\notin$[requests(->next_fd)^{l}, requests(->next_fd)$^{l+1}$->last_fd])))

$\vee$ (($\forall$k:0..n-1.[requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd]) $\wedge$ [requests(->next_fd)^{n}]

$\wedge$ {requests->last_fd, requests(->next_fd)$^{n+1}$}$_{N}$

$\wedge$ ($\forall$k:0..i-1.**plist**(requests(->next_fd)^{k}->next_prio)) $\wedge$ ($\forall$k:i+1..n.**plist**(requests(->next_fd)^{k}->next_prio))

$\wedge$ ($\forall$l:0..ip-1.{requests(->next_fd)^{i}(->next_prio)^{l}}) $\wedge$ {requests(->next_fd)^{i}(->next_prio)^{ip}, last}

$\wedge$ {last->next_prio, req} $\wedge$ **plist**(req->next_prio) $\wedge$ **rlist**(runlist)

$\wedge$ ($\forall$k:0..n. $\forall$l:0..n. (k!=l $\Rightarrow$ requests(->next_fd)$^{k}\notin$[requests(->next_fd)^{l}, requests(->next_fd)$^{l+1}$->last_fd]))) }

```
/*----precondition*/
void __aio_remove_request(struct requestlist *last, struct requestlist *req, int all)
{
        if (last != NULL)
        {
```

{ ($\forall$k:0..n-1.[requests(->next_fd)^{k}, requests(->next_fd)$^{k+1}$->last_fd]) $\wedge$ [requests(->next_fd)^{n}]

$\wedge$ {requests->last_fd, requests(->next_fd)$^{n+1}$}$_{N}$

$\wedge$ ($\forall$k:0..i-1.**plist**(requests(->next_fd)^{k}->next_prio))

$\wedge$ ($\forall$k:i+1..n.**plist**(requests(->next_fd)^{k}->next_prio))

```
    ∧ (∀l:0..ip-1.{requests(->next_fd)$^{i}$(->next_prio)$^{l}$}) ∧ {requests(->next_fd)$^{i}$(->next_prio)$^{ip}$, last}
    ∧ {last->next_prio, req} ∧ **plist**(req->next_prio) ∧ **rlist**(runlist)
    ∧ (∀k:0..n. ∀l:0..n. (k!=l
        ⇒ requests(->next_fd)$^{k}$∉[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd])) }
    if (all == 1)
        last->next_prio = **NULL**;
    else
        last->next_prio = req->next_prio;
    { (∀k:0..n-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
    ∧ [requests(->next_fd)$^{n}$] ∧ {requests->last_fd, requests(->next_fd)$^{n+1}$}$_{N}$
    ∧ (∀k:0..i-1.**plist**(requests(->next_fd)$^{k}$->next_prio))
    ∧ (∀k:i+1..n.**plist**(requests(->next_fd)$^{k}$->next_prio))
    ∧ (∀l:0..ip-1.{requests(->next_fd)$^{i}$(->next_prio)$^{l}$}) ∧ {requests(->next_fd)$^{i}$(->next_prio)$^{ip}$, last}
    ∧ {req} ∧ **plist**(last->next_prio)
    ∧ **rlist**(runlist) ∧ (∀k:0..n. ∀l:0..n. (k!=l
                ⇒ requests(->next_fd)$^{k}$∉[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd])) }
}
else
{
    if (all == 1 ∨ req->next_prio == **NULL**)
    {
        { 0<=i<=n-1 ∧ (∀k:0..i-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ {requests(->next_fd)$^{i}$, requests(->next_fd)$^{i+1}$->last_fd, req, runlist(->next_run)$^{j}$ }
        ∧ (∀k:i+1..n-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd]) ∧ [requests(->next_fd)$^{n}$]
        ∧ {requests->last_fd, requests(->next_fd)$^{n+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..n.**plist**(requests(->next_fd)$^{k}$->next_prio))
        ∧ (∀k:0..j-1.[runlist(->next_run)$^{k}$]) ∧ (∀k:j+1..m.[runlist(->next_run)$^{k}$])
        ∧ (∀k:0..m. ∀l:0..m. (k!=l ⇒ runlist(->next_fd)$^{k}$∉[runlist(->next_fd)$^{l}$]))
        ∧ (∀k:0..n. ∀l:0..n. (k!=l
            ⇒ requests(->next_fd)$^{k}$∉[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd])) }
        if (req->last_fd != **NULL**)
            req->last_fd->next_fd = req->next_fd;
        else
            requests = req->next_fd;
        { 0<=i<=n-1 ∧ (∀k:0..i-2.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ [requests(->next_fd)$^{i-1}$, req->last_fd] ∧ {req->next_fd->last_fd, req, runlist(->next_run)$^{j}$ }
        ∧ [req->next_fd, requests(->next_fd)$^{i}$ , req(->next_fd)$^{2}$->last_fd]
        ∧ (∀k:i+2..n-1.[req(->next_fd)$^{k-i}$, req(->next_fd)$^{k-i+1}$->last_fd]) ∧ [req(->next_fd)$^{n-i}$]
        ∧ {requests->last_fd, req(->next_fd)$^{n-i+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..i-1.**plist**(requests(->next_fd)$^{k}$->next_prio))
        ∧ (∀k:i..n.**plist**(req(->next_fd)$^{k-i}$->next_prio))
        ∧ (∀k:0..j-1.[runlist(->next_run)$^{k}$]) ∧ (∀k:j+1..m.[runlist(->next_run)$^{k}$]) ∧ ... }
        if (req->next_fd != **NULL**)
```

```
                req->next_fd->last_fd = req->last_fd;
        { 0<=i<=n-1 ∧ (∀k:0..n-2.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ {req, runlist(->next_run)$^{j}$ } ∧ [requests(->next_fd)$^{n-1}$]
        ∧ {requests->last_fd, requests(->next_fd)$^{n}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..n-1.**plist**(requests(->next_fd)$^{k}$->next_prio)) ∧ **plist**(req->next_prio)
        ∧ (∀k:0..j-1.[runlist(->next_run)$^{k}$]) ∧ (∀k:j+1..m.[runlist(->next_run)$^{k}$])
        ∧ (∀k:0..m. ∀l:0..m. (k!=l ⇒ runlist(->next_fd)$^{k}$∉[runlist(->next_fd)$^{l}$]))
        ∧ (∀k:0..n-1. ∀l:0..n-1. (k!=l
                ⇒ requests(->next_fd)$^{k}$∉[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd])) }
}
else
{
        { 0<=i<=n-1 ∧ (∀k:0..i-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ {requests(->next_fd)$^{i}$, requests(->next_fd)$^{i+1}$->last_fd, req, runlist(->next_run)$^{j}$ }
        ∧ (∀k:i+1..n-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd]) ∧ [requests(->next_fd)$^{n}$]
        ∧ {requests->last_fd, requests(->next_fd)$^{n+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..n.k!=i ∧ **plist**(requests(->next_fd)$^{k}$->next_prio))
        ∧ {req->next_prio} ∧ **plist**(req->next_prio->next_prio)
        ∧ (∀k:0..j-1.[runlist(->next_run)$^{k}$]) ∧ (∀k:j+1..m.[runlist(->next_run)$^{k}$])
        ∧ (∀k:0..m. ∀l:0..m. (k!=l ⇒ runlist(->next_fd)$^{k}$∉[runlist(->next_fd)$^{l}$]))
        ∧ (∀k:0..n. ∀l:0..n. (k!=l
                ⇒ requests(->next_fd)$^{k}$∉[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd])) }
        if (req->last_fd != **NULL**)
                req->last_fd->next_fd = req->next_prio;
        else
                requests = req->next_prio;
        { 0<=i<=n-1 ∧ (∀k:0..i-2.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ [requests(->next_fd)$^{i-1}$, req->last_fd] ∧ {req->next_fd->last_fd, req, runlist(->next_run)$^{j}$ }
        ∧ {req->next_prio, requests(->next_fd)$^{i}$}
        ∧ (∀k:i+1..n-1.[req(->next_fd)$^{k-i}$, req(->next_fd)$^{k-i+1}$->last_fd]) ∧ [req(->next_fd)$^{n-i}$]
        ∧ {requests->last_fd, req(->next_fd)$^{n-i+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..i-1.**plist**(requests(->next_fd)$^{k}$->next_prio)) ∧ **plist**(req->next_prio->next_prio)
        ∧ (∀k:i+1..n.**plist**(req(->next_fd)$^{k-i}$->next_prio))
        ∧ (∀k:0..j-1.[runlist(->next_run)$^{k}$]) ∧ (∀k:j+1..m.[runlist(->next_run)$^{k}$]) ∧ ... }
        if (req->next_fd != **NULL**)
                req->next_fd->last_fd = req->next_prio;
        { 0<=i<=n-1 ∧ (∀k:0..i-2.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd])
        ∧ [requests(->next_fd)$^{i-1}$, req->last_fd] ∧ {req, runlist(->next_run)$^{j}$ }
        ∧ {req->next_fd->last_fd, req->next_prio, requests(->next_fd)$^{i}$}
        ∧ (∀k:i+1..n-1.[req(->next_fd)$^{k-i}$, req(->next_fd)$^{k-i+1}$->last_fd]) ∧ [req(->next_fd)$^{n-i}$]
        ∧ {requests->last_fd, req(->next_fd)$^{n-i+1}$, last, runlist(->next_run)$^{m+1}$}$_{N}$
        ∧ (∀k:0..i-1.**plist**(requests(->next_fd)$^{k}$->next_prio)) ∧ **plist**(req->next_prio->next_prio)
        ∧ (∀k:i+1..n.**plist**(req(->next_fd)$^{k-i}$->next_prio))
        ∧ (∀k:0..j-1.[runlist(->next_run)^{k}]) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}]) ∧ ... }
        req->next_prio->last_fd = req->last_fd;
        req->next_prio->next_fd = req->next_fd;
        { 0<=i<=n-1 ∧ (∀k:0..i-1.[requests(->next_fd)^{k}, requests(->next_fd)^{k+1}->last_fd])
        ∧ {req, runlist(->next_run)^{j}}
        ∧ {req->next_prio, requests(->next_fd)^{i} , requests(->next_fd)^{i+1}->last_fd}
        ∧ (∀k:i+1..n-1.[requests(->next_fd)^{k}, requests(->next_fd)^{k+1}->last_fd]) ∧ [requests(->next_fd)^{n}]
        ∧ {requests->last_fd, requests(->next_fd)^{n+1}, last, runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..n.plist(requests(->next_fd)^{k}->next_prio))
        ∧ (∀k:0..j-1.[runlist(->next_run)^{k}]) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ (∀k:0..m. ∀l:0..m. (k!=l ⇒ runlist(->next_fd)^{k}∉[runlist(->next_fd)^{l}]))
        ∧ (∀k:0..n. ∀l:0..n. (k!=l
                ⇒ requests(->next_fd)^{k}∉[requests(->next_fd)^{l}, requests(->next_fd)^{l+1}->last_fd])) }
        req->next_prio->running = yes;
}
if (req->running == yes)
{
        /*Using Frame Rule*/
        { {req, runlist(->next_run)^{j}}
        ∧ (∀k:0..j-1.[runlist(->next_run)^{k}]) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ {last, runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])) }
        struct requestlist *runp = runlist;
        last = NULL;
        { (0<j<=m ∧{req, runlist(->next_run)^{j}} ∧ [runlist, runp]
        ∧ (∀k:1..j-1.[runlist(->next_run)^{k}]) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ {last, runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..n-1.∀l:0..n-1. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])))
        ∨ (j=0 ∧{req, runlist, runp} ∧ (∀k:1..m.[runlist(->next_run)^{k}]) ∧ {last, runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}]))) }
        /*loop invariant----*/
        { (0<j<=m ∧ (∃t:0..j-2.((∀k:0..t-1.[runlist(->next_run)^{k}]) ∧ (∀k:t+2..j-1.[runlist(->next_run)^{k}]))
        ∧ [runlist(->next_run)^{t}, last] ∧ [runlist(->next_run)^{t+1}, runp]
        ∧ {runlist(->next_run)^{j}, req} ∧ (∀k:j+1..m.[runlist(->next_run)^{k}]) ∧ {runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])))
        ∨ (0<j<=m ∧ (∀k:0..j-2.[runlist(->next_run)^{k}]) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ [runlist(->next_run)^{j-1}, last] ∧ {runlist(->next_run)^{j}, req, runp} ∧ {runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])))
        ∨ (j=0 ∧ {req, runlist, runp} ∧ (∀k:1..m.[runlist(->next_run)^{k}]) ∧ {last, runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}]))) }
        while (runp != NULL ∧ runp != req)
        {
                { 0<j<=m ∧ (∃t:0..j-2.((∀k:0..t-1.[runlist(->next_run)^{k}])
        ∧ [runlist(->next_run)^{t}, last] ∧ [runlist(->next_run)^{t+1}, runp]
        ∧ (∀k:t+2..j-1.[runlist(->next_run)^{k}])) ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ {runlist(->next_run)^{j}, req} ∧ {runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])) }
        last = runp;
        runp = runp->next_run;
        { (0<j<=m ∧ (∃t:0..j-2.((∀k:0..t-1.[runlist(->next_run)^{k}])
        ∧ [runlist(->next_run)^{t}, last] ∧ [runlist(->next_run)^{t+1}, runp]
        ∧ (∀k:t+2..j-1.[runlist(->next_run)^{k}]))
        ∧ {runlist(->next_run)^{j}, req} ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ {runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])))
        ∨ (0<j<=m ∧ (∀k:0..j-2.[runlist(->next_run)^{k}])
        ∧ [runlist(->next_run)^{j-1}, last] ∧ {runlist(->next_run)^{j}, req, runp}
        ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
        ∧ {runlist(->next_run)^{m+1}}_{N}
        ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}]))) }
    }
    { (0<j<=m ∧ (∀k:0..j-2.[runlist(->next_run)^{k}]) ∧ [runlist(->next_run)^{j-1}, last]
    ∧ {runlist(->next_run)^{j}, req, runp}  ∧ (∀k:j+1..m.[runlist(->next_run)^{k}])
    ∧ {runlist(->next_run)^{m+1}}_{N}
    ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])))
    ∨ (j=0 ∧ {req, runlist, runp} ∧ (∀k:1..m.[runlist(->next_run)^{k}])    ∧ {last, runlist(->next_run)^{m+1}}_{N}
    ∧ (∀k:0..m.∀l:0..m. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}]))) }
    if (runp == req)
    {
        if (last == NULL)
            runlist = runp->next_run;
        else
            last->next_run = runp->next_run;
    }
    { 0<=j<=m ∧ (∀k:0..j-1.[runlist(->next_run)^{k}]) ∧ {req, runp}
    ∧ [runlist(->next_run)^{j}, req->next_run]
    ∧ (∀k:j+1..m-1.[runlist(->next_run)^{k}]) ∧ {runlist(->next_run)^{m}}_{N}
    ∧ (∀k:0..m-1.∀l:0..m-1. (k!=l ⇒ runlist(->next_run)^{k}∉[runlist(->next_run)^{l}])) }
}
{ (0<=i<=n-1 ∧ (∀k:0..n-2.[requests(->next_fd)^{k}, requests(->next_fd)^{k+1}->last_fd])
∧ {req} ∧ [requests(->next_fd)^{n-1}] ∧ {requests->last_fd, requests(->next_fd)^{n}, runlist(->next_run)^{m}}_{N}
∧ (∀k:0..n-1.plist(requests(->next_fd)^{k}->next_prio)) ∧ plist(req->next_prio)
∧ (∀k:0..m-1.[runlist(->next_run)^{k}])
∧ (∀k:0..m-1. ∀l:0..m-1. (k!=l ⇒ runlist(->next_fd)^{k}∉[runlist(->next_fd)^{l}]))
∧ (∀k:0..n-1. ∀l:0..n-1. (k!=l
    ⇒ requests(->next_fd)^{k}∉[requests(->next_fd)^{l}, requests(->next_fd)^{l+1}->last_fd])))
```

```
            $\vee$ (0<=i<=n-1 $\wedge$ ($\forall$k:0..i-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd]) $\wedge$ {req}
            $\wedge$ {req->next_prio, requests(->next_fd)$^{i}$ , requests(->next_fd)$^{i+1}$->last_fd}
            $\wedge$ ($\forall$k:i+1..n-1.[requests(->next_fd)$^{k}$, requests(->next_fd)$^{k+1}$->last_fd]) $\wedge$ [requests(->next_fd)$^{n}$]
            $\wedge$ {requests->last_fd, requests(->next_fd)$^{n+1}$, runlist(->next_run)$^{m}$}$_{N}$
            $\wedge$ ($\forall$k:0..n.**plist**(requests(->next_fd)$^{k}$->next_prio)) $\wedge$ ($\forall$k:0..m-1.[runlist(->next_run)$^{k}$])
            $\wedge$ ($\forall$k:0..m-1. $\forall$l:0..m-1. (k!=l $\Rightarrow$ runlist(->next_fd)$^{k}$$\notin$[runlist(->next_fd)$^{l}$]))
            $\wedge$ ($\forall$k:0..n. $\forall$l:0..n. (k!=l
                $\Rightarrow$ requests(->next_fd)$^{k}$$\notin$[requests(->next_fd)$^{l}$, requests(->next_fd)$^{l+1}$->last_fd]))) }
        }
}
{ (**rq**(requests, runlist, n, m) $\wedge$ {req}) $\vee$ (**rq**(requests, runlist, n, m-1) $\wedge$ {req})
$\vee$ (**rq**(requests, runlist, n-1, m-1) $\wedge$ {req}) } /*----postcondition*/
```

References:

[1] Hongjin Liang, Yu Zhang, Yiyun Chen, Zhaopeng Li, Baojian Hua. A pointer logic dealing with uncertain equality of pointers (Chinese Version). Available at: http://kyhcs.ustcsz.edu.cn/lss.

[2] The GNU C Library. http://www.gnu.org/software/libc/.