Protected Service Processes

Peter J. Denning

Protected Service Processes

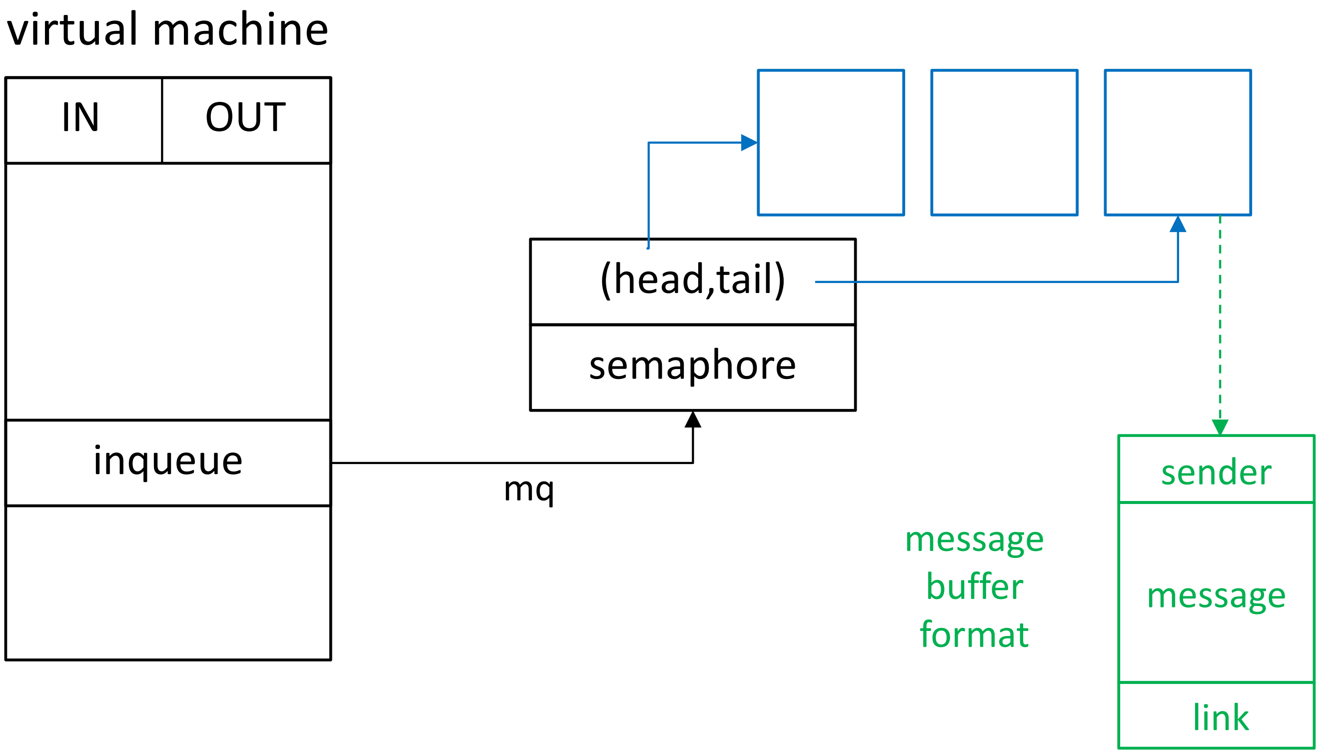
- All processes have private address spaces
  - Basis of security and privacy guarantees of an OS
  - Supports reliability – for example, disk controller owns disk (and driver) as private objects, inaccessible to other processes
- Service processes receive requests and make responses via a message system outside their private address spaces
  - For example, a process sends a read request message to disk controller process, which responds by sending back the record

- Simple structure of service process used here:
  - A service process operates in a cycle that begins with a “homing position”, which is the only place it listens for incoming requests
  - When a request arrives the process services it, sends reply, and returns to homing position
  - During its service it may make requests of other service processes for subtasks; when this happens, it stops and waits for a response
- A process can only be waiting for one reply at a time
  - Multiple incoming requests are possible
  - They are queued and serviced one at a time
  - Only one return reply is possible to a service process sub-request

- Every process has input queue to receive requests for service
  - Pointer in inqueue slot of the process's VM
- Inqueue is a linked list of message buffers, each holding a request from an identified sender
  - Queues and message buffers are stored in kernel space, not in the private address spaces of processes
  - Reply returned in same message buffer that contained the request

- This architecture is a “sandbox” that protects
  - devices and other components from misuse
  - low level protocols for using devices
  - rest of system from service process errors and bugs

- Message queue structure
  - message buffer mb = (sender, message)
  - mq = CREATE_MQ( ) returns pointer mq to queue descriptor that contains (head,tail) for the queue and a queue-length counting semaphore; mq is placed in VM inqueue slot

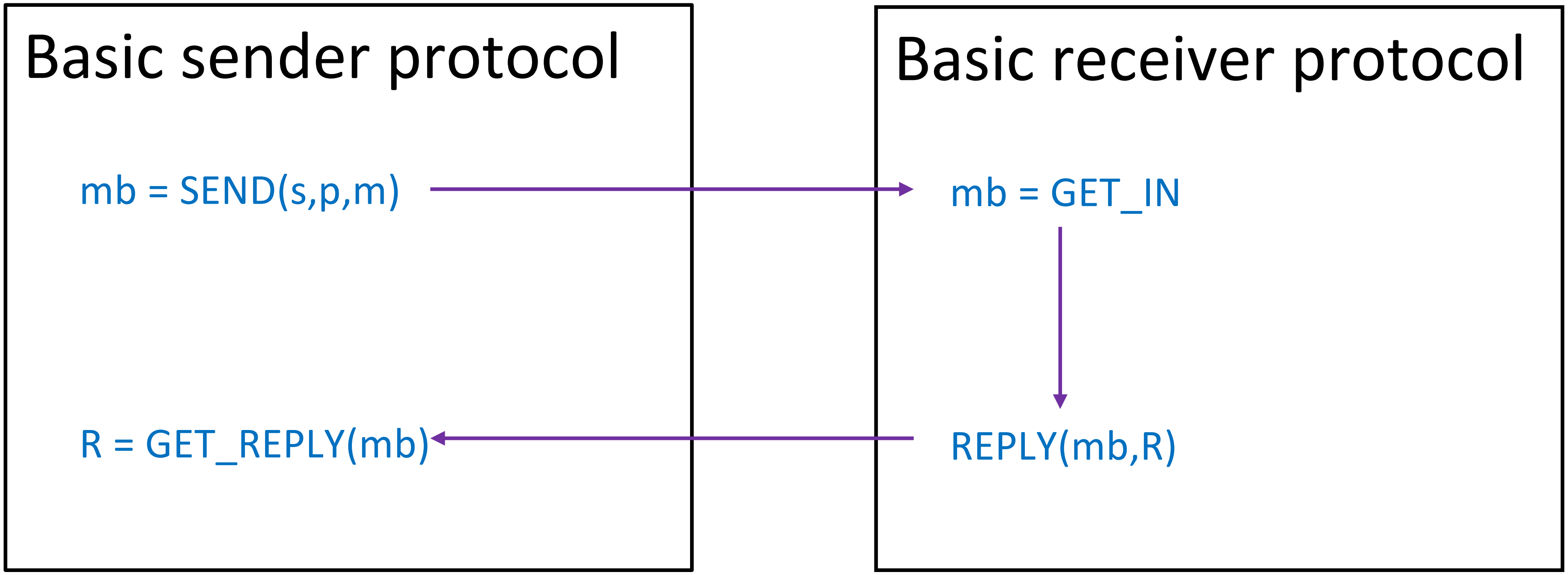


Guaranteed Return Policy: response returned in same message buffer as request

  - Simplifies responding
  - Guarantees response goes to original sender
  - Reduces chance of deadlock from running out of buffers

Policy implemented with the protocol below. SEND gets empty message buffer from OS pool, sets mb.sender=p and mb.message=m, and links mb to the tail of inqueue[s]. GET_REPLY goes to SLEEP until the receiver returns the response R = mb.message to that request.

In receiver, GET_IN does WAIT(mq[self].semaphore) and then returns a pointer to the mb taken from its inqueue. It then processes the request in mb.message. When response R is ready it uses REPLY to insert R into mb.message and then issues WAKEUP(mb.sender)



Basic sender protocol

mb = SEND(s,p,m)

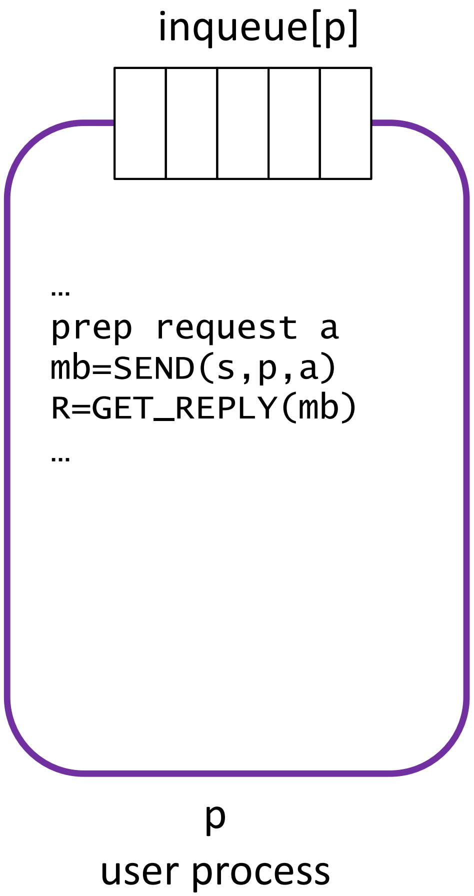
R = GET_REPLY(mb)

Basic receiver protocol

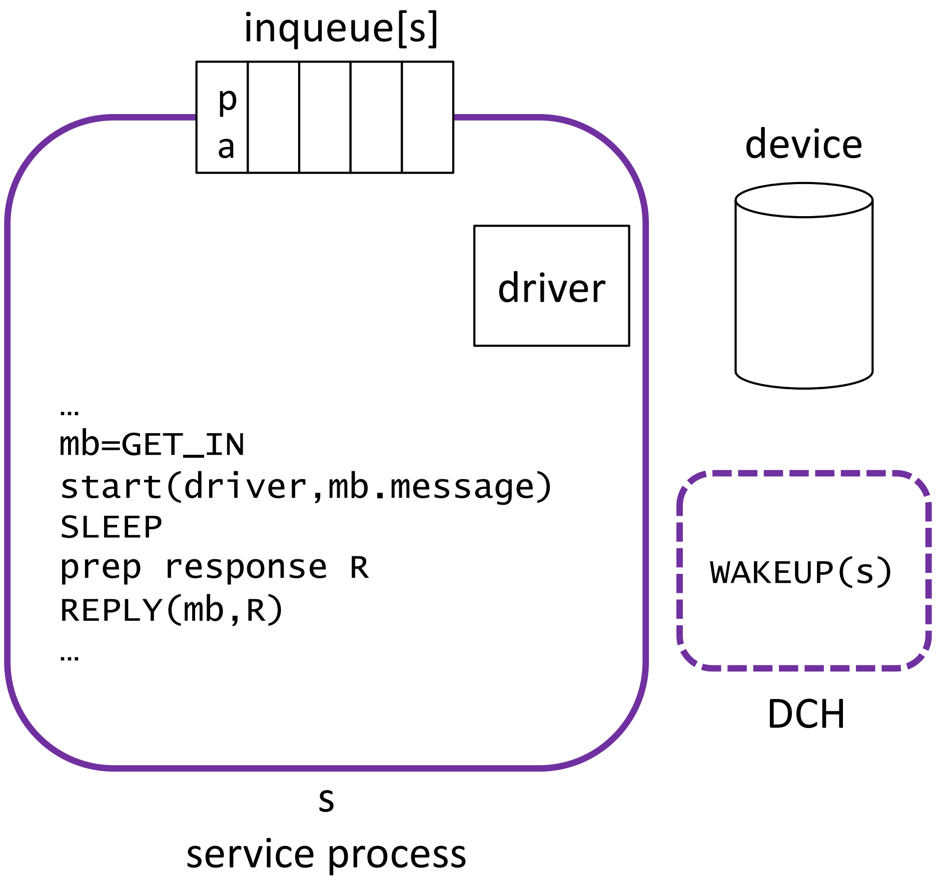
mb = GET_IN

REPLY(mb,R)

This example shows a user process p ready to interact with a server process s that controls a device. The device and its driver are private to s. The Device Completion Interrupt (DCH) signals that the device has completed a previous request.

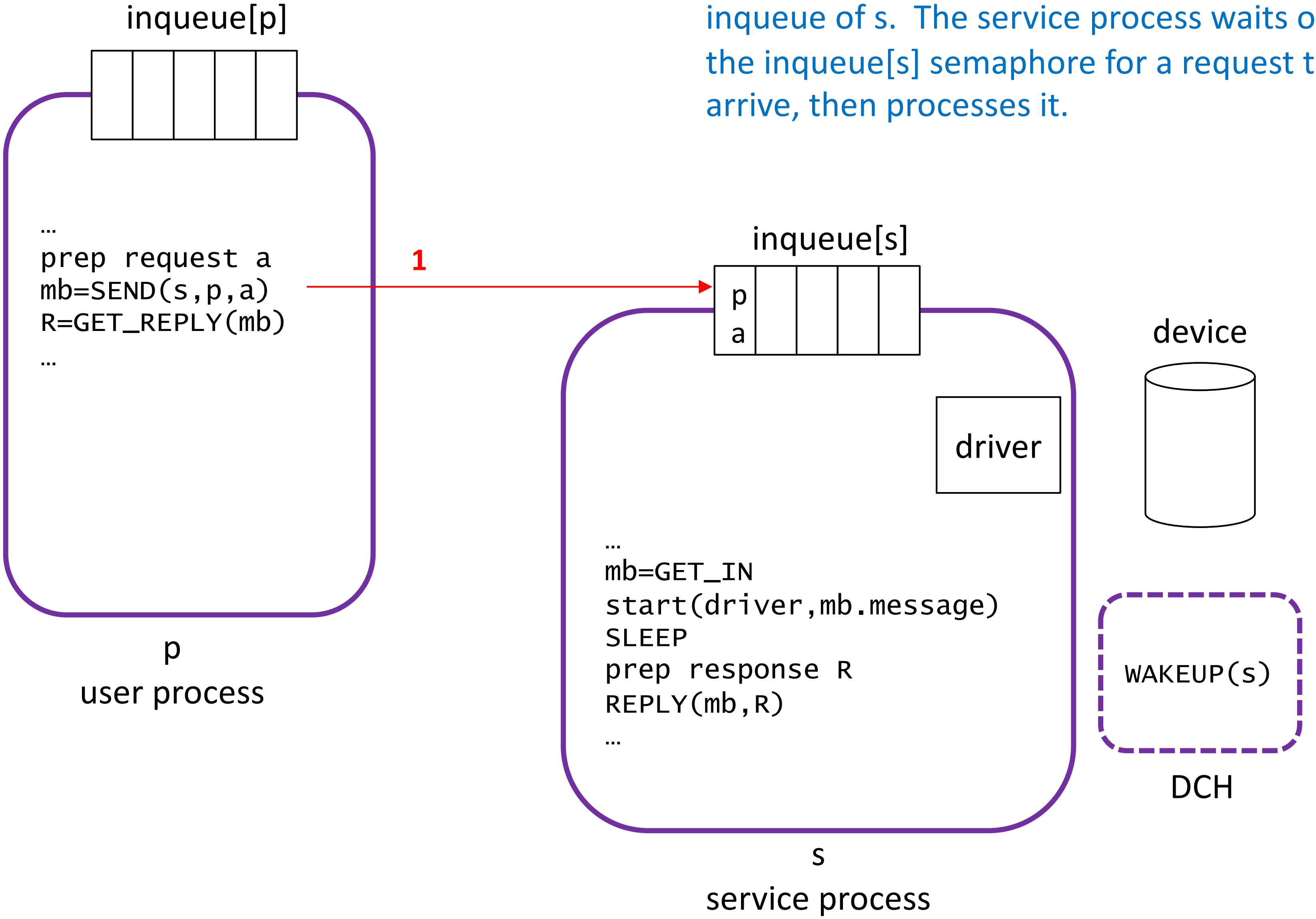


```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```



```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

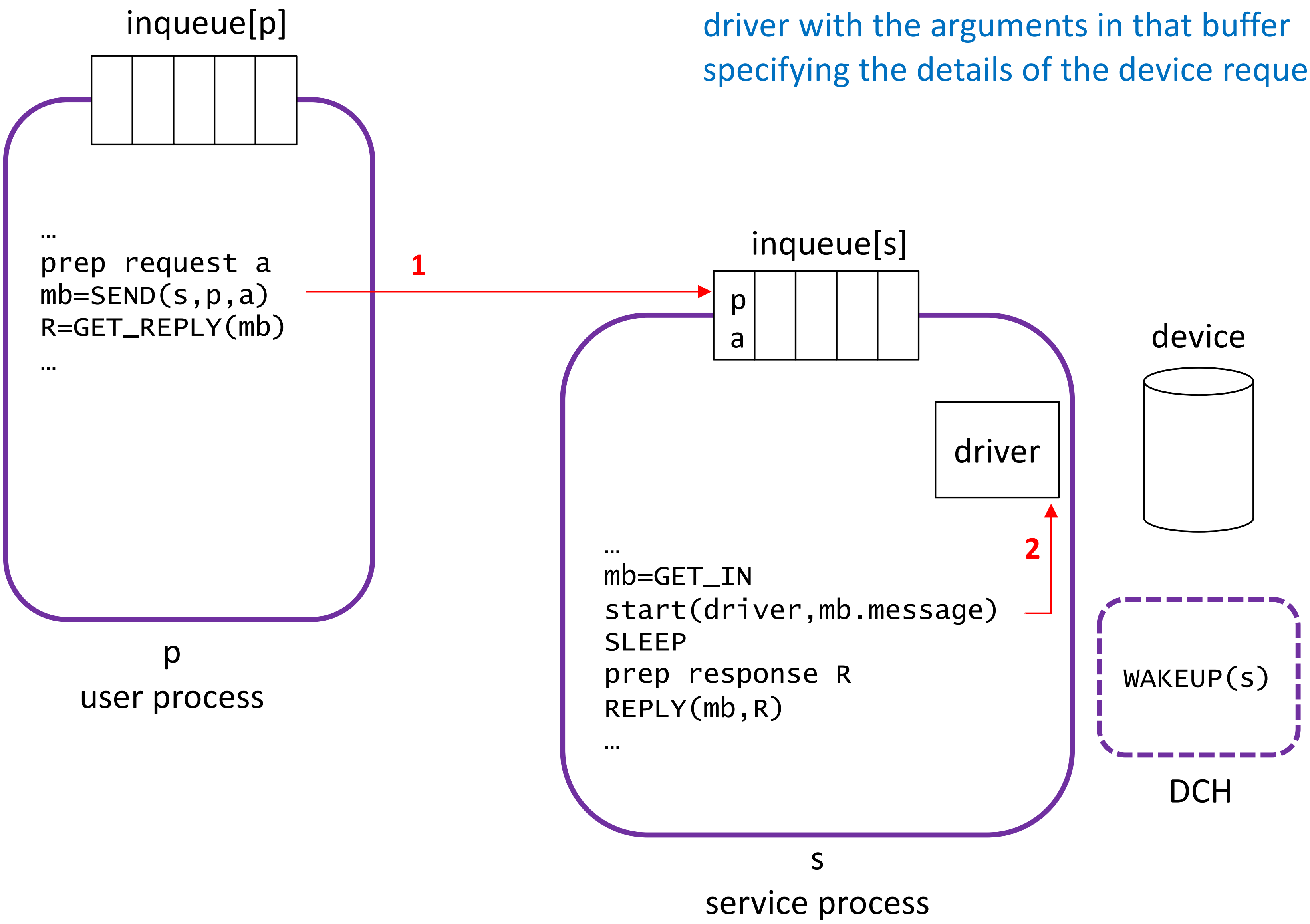
(1) The protocol begins with p generating a request for s and inserting it into the inqueue of s. The service process waits on the inqueue[s] semaphore for a request to arrive, then processes it.



```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

(2) After the service process removes a message buffer from its inqueue, it calls the driver with the arguments in that buffer specifying the details of the device request.



inqueue[p]

```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

p
user process

1

inqueue[s]

p
a

driver

2

```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

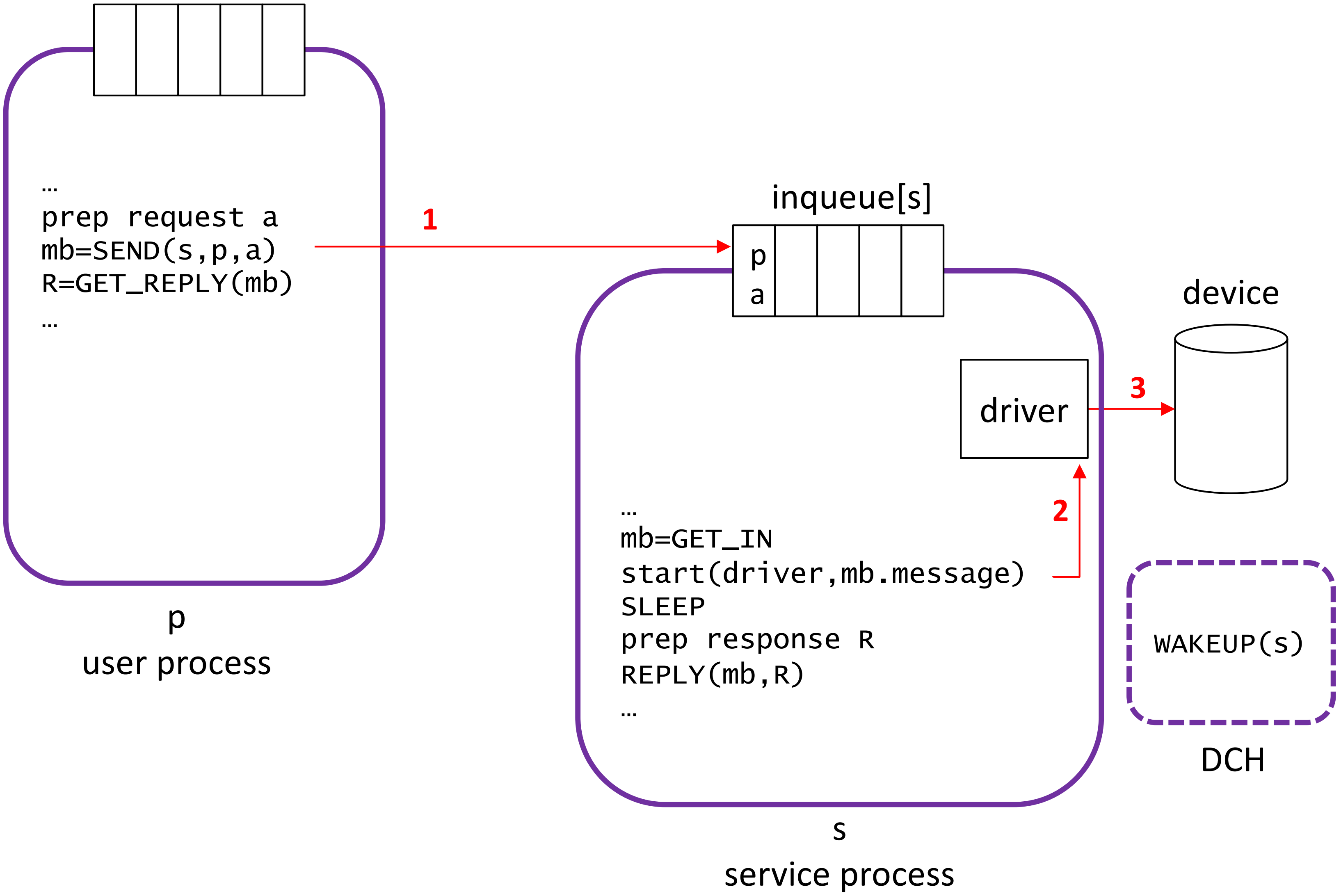
s
service process

device

WAKEUP(s)

DCH

(3) The driver sends a start signal (plus arguments) to the device, which begins processing the specified transaction.



inqueue[p]

```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

p
user process

1

inqueue[s]

p
a

device

driver

3

2

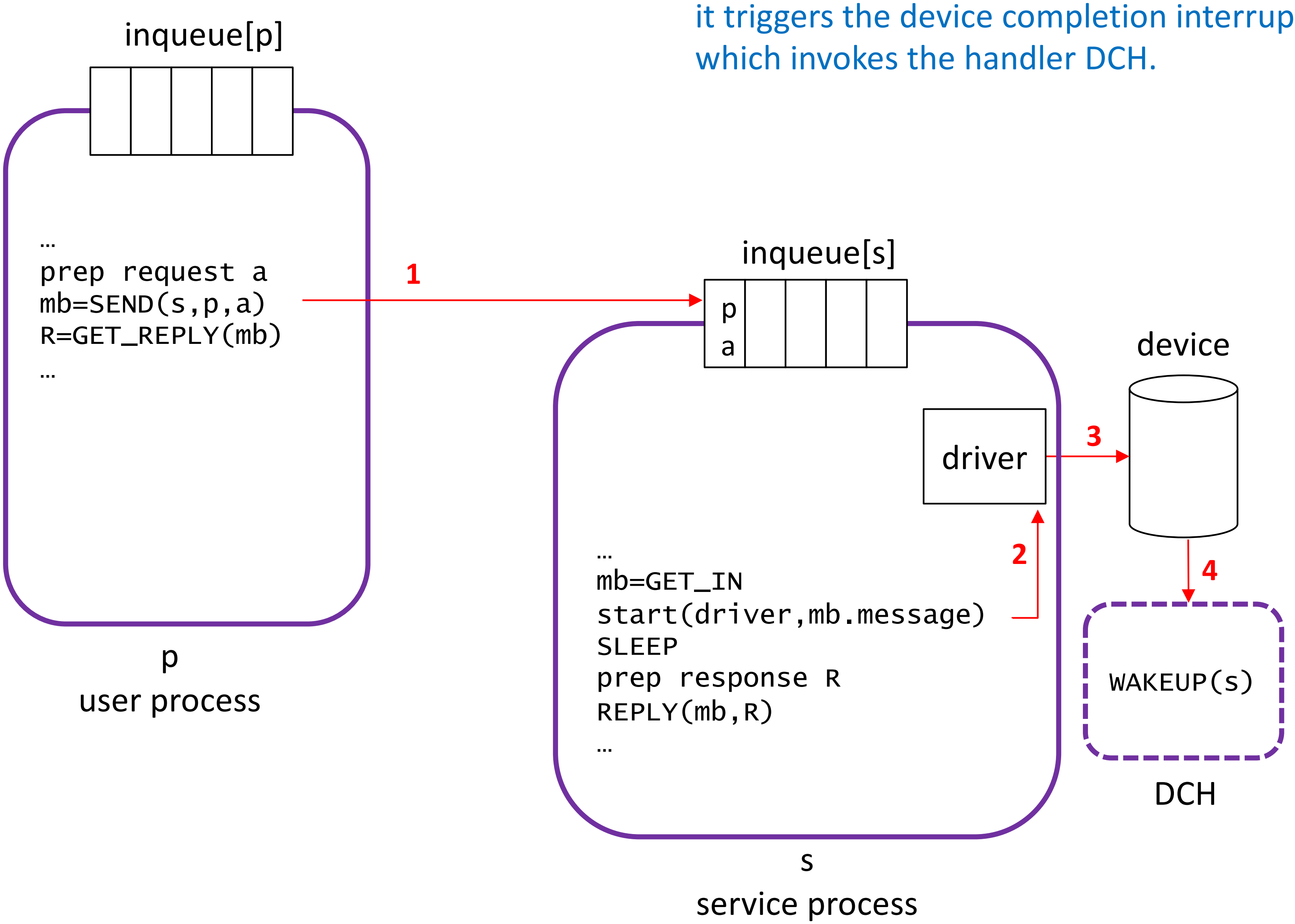
```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

WAKEUP(s)

DCH

s
service process

(4) When the device finishes the transaction, it triggers the device completion interrupt, which invokes the handler DCH.



inqueue[p]

```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

p
user process

1

inqueue[s]

p
a

driver

3

device

2

4

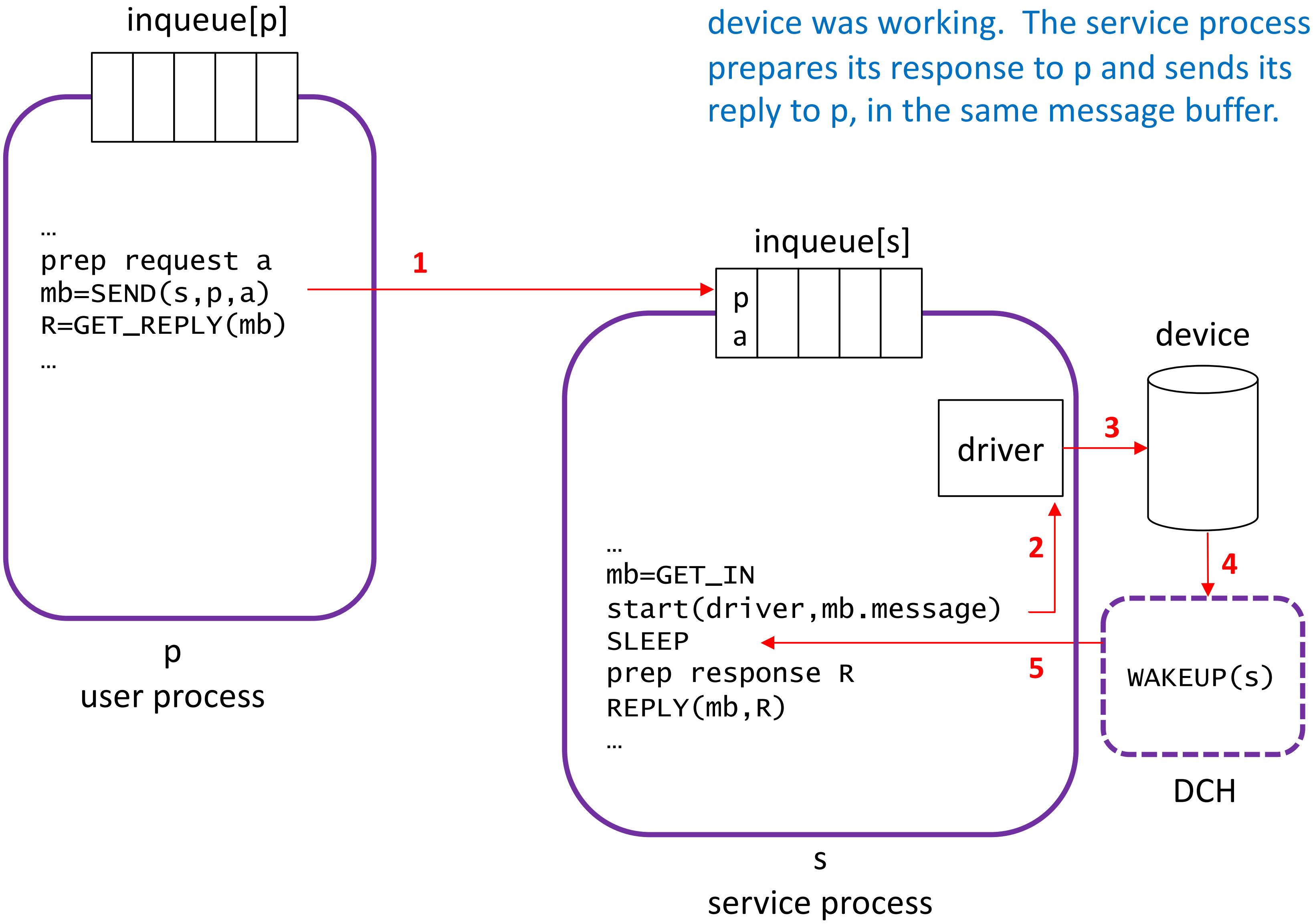
```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

WAKEUP(s)

DCH

s
service process

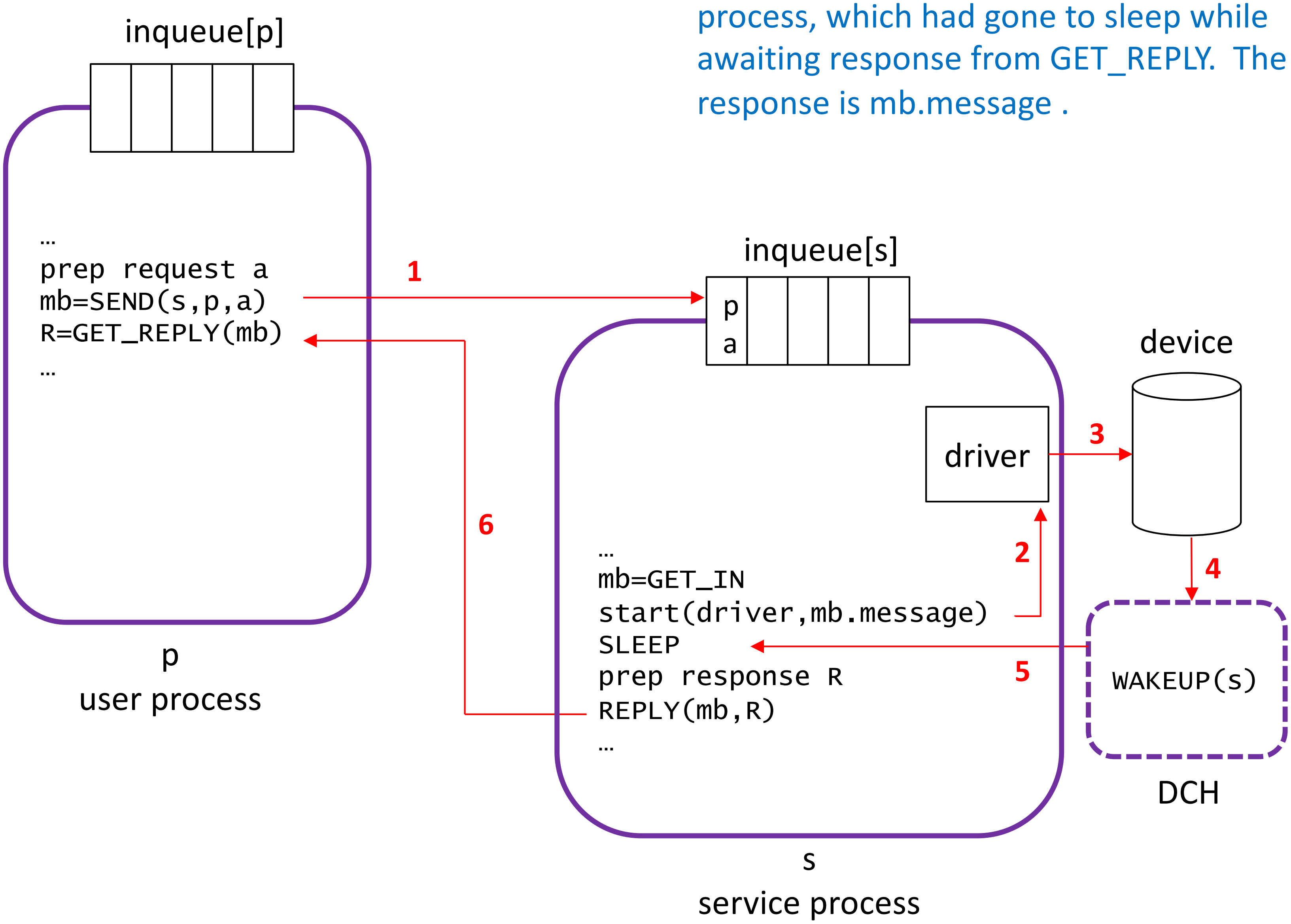
(5) The handler wakes up the service process, which had gone to sleep while the device was working. The service process prepares its response to p and sends its reply to p, in the same message buffer.



```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

(6) The service process awakens the user process, which had gone to sleep while awaiting response from GET_REPLY. The response is mb.message .



inqueue[p]

```
…
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
…
```

p

user process

inqueue[s]

p

a

driver

device

```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

WAKEUP(s)

DCH

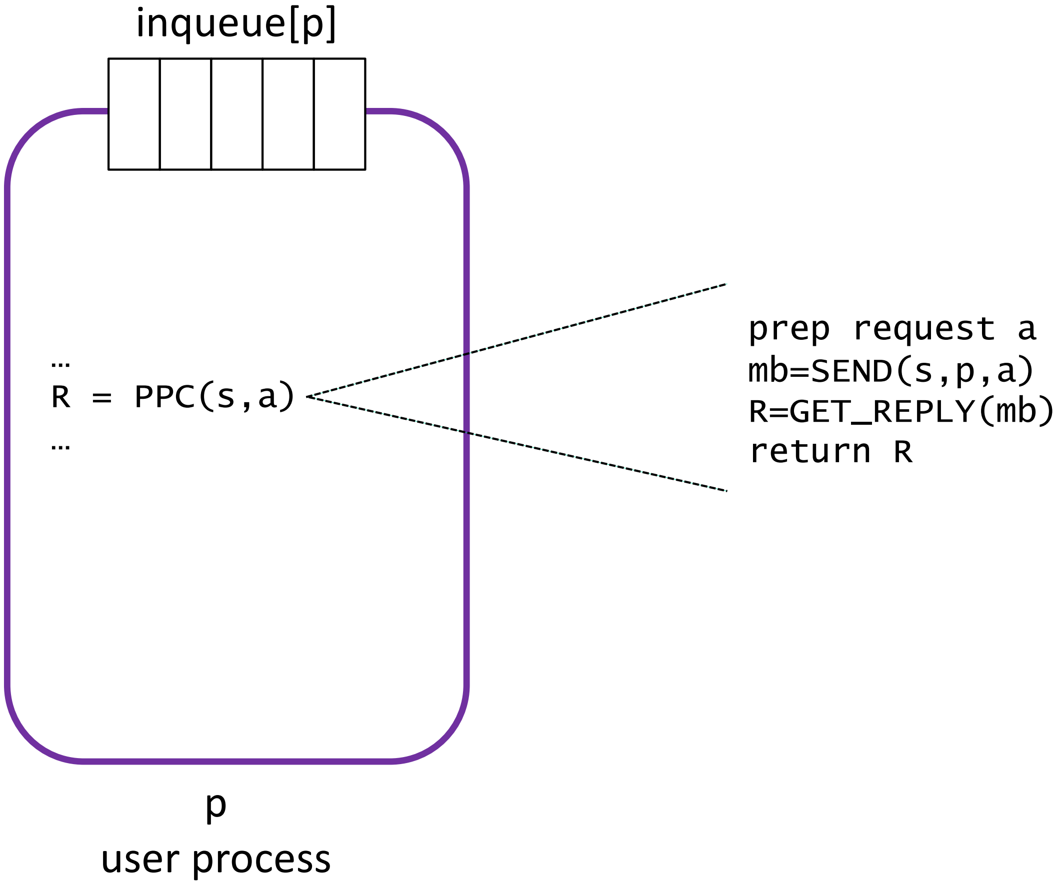
s

service process

Messages Model

- The instructions on the client side (in process p) can be grouped together behind a common interface
- The server protocol already provides for the server to return a response
- Looks like a procedure call on a protected procedure on the same machine
  - Protected means called procedure is in a separate process with its own address space
- Let's call it PPC = protected procedure call

This is similar to the remote procedure call (RPC). RPC differs by sending messages across the network to a remote server (and OS) to call the procedure on that server. RPC is discussed in a separate chapter.



inqueue[p]

```
…
R = PPC(s,a)
…
```

```
prep request a
mb=SEND(s,p,a)
R=GET_REPLY(mb)
return R
```

p
user process

```
…
mb=GET_IN
start(driver,mb.message)
SLEEP
prep response R
REPLY(mb,R)
…
```

S

service process

Two Possible Message Deadlocks

- Service processes in a request loop
  - Arrange all services processes in a tree hierarchy and allow only downward-request-upward-reply (same principle as in kernel)
- Service process chain waiting for buffer pool empty
  - Make buffer pool larger than the maximum number of service processes

Scenario for message pool deadlock: pool size is K, and in the process tree is a path of downward calls longer than K. Each call invokes a new sleep and when there are no more buffers the last caller waits on an empty pool. All the processes that could return a buffer to the pool are asleep.