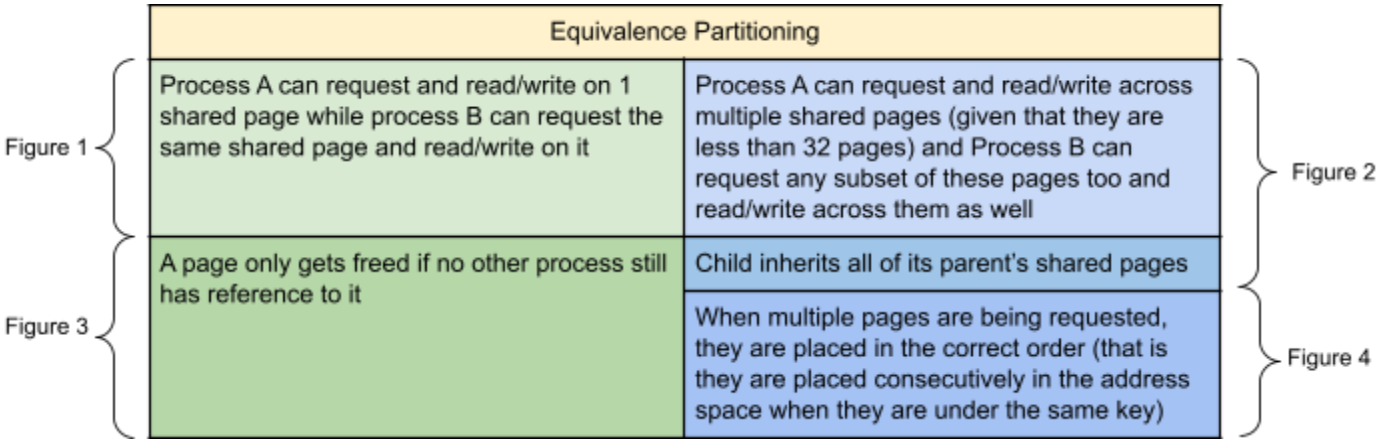
Test Data Explanation



`shpg1test` & `shpg2test`

```
$ shpg1test
Process 1 wrote: CS450 is very easy!
Process 2 sees that process 1 wrote: CS450 is very easy!
Process 2 wrote: CS450 is very hard!
Process 1 sees that process 2 wrote: CS450 is very hard!
```

- Process A requests 1 shared page under key `111` and writes `"CS450 is very easy!\0"`
- Process A forks B who execs `shpg2test`
- In `shpg2test`, B requests the same shared page under key 111 and overwrites it with `"CS450 is very hard!\0"`
- Process A reaps B and reads what B has overwritten the page with

Figure 1 demonstrates that a single page is properly shared between two user processes.

`shpg3test`

```
$ shpg3test
Parent says: at a[0] on page 0 is 0
Parent says: at a[1] on page 1024 is 1024
Parent says: at a[2] on page 2048 is 2048
Parent says: at a[3] on page 3072 is 3072
Parent says: at a[4] on page 4096 is 4096
Child says: at a[0] on page 0 is 0
Child says: at a[1] on page 1024 is 1024
Child says: at a[2] on page 2048 is 2048
Child says: at a[3] on page 3072 is 3072
Child says: at a[4] on page 4096 is 4096
Parent says: at a[0] on page 0 is 0
Parent says: at a[1] on page 1024 is 1024
Parent says: at a[2] on page 2048 is 4096
Parent says: at a[3] on page 3072 is 6144
Parent says: at a[4] on page 4096 is 8192
```

- First process A requests 5 shared pages with key `500` and gets pages with PFNs of a, b, c, d, e
- Process A writes on all 5 pages using 5120 integers
    - each integer is 4 bytes so a 4096 byte page can hold 1024 integers
    - 5 pages can hold 5120 integers
- Process A forks process B who still has access to the shared pages of PFNs a, b, c, d, e (inherited by A)
- Process B reads from these pages
- Process B then requests 3 mores shared pages with the same key and thus gets back pages with PFNS c, d, e
    - B's virtual address space has 8 shared pages with PFNs of a, b, c, d, e, c, d, e (the virtual pages with the same PFNs are synced)
- Process B then writes on these 3 pages, multiplying the integer values by 2
- Process A reaps process B then reads its 5 shared pages (PFNS a, b, c, d, e) with 3 of them (PFNS c, d, e) having values overwritten by process B

Figure 2 demonstrates that a process can request multiple shared pages and write across all of them using the returned virtual address. It also shows that a child inherits its parent process's shared pages. By having the child also request a subset of the pages under the same key, writing in them, then having the parent read those pages shows that multiple pages can be properly shared.

`shpg4test` & `shpg5test`

```
$ shpg4test
a now points to 20
a now points to 19
a now points to 18
a now points to 17
a now points to 16
a now points to 15
a now points to 14
a now points to 13
a now points to 12
a now points to 11
a now points to 10
a now points to 9
a now points to 8
a now points to 7
a now points to 6
a now points to 5
a now points to 4
a now points to 3
a now points to 2
a now points to 1
From first process: a now points to 0
```

A "silly loop"

- Process A requests 1 shared page under key `313`, writes integer value 20 in it
- A forks B who exec's `shpg5test`
- In `shpg2test`, B requests the same shared page under key `313` then decrements the integer value, prints, and frees its shared page
- B execs `shpg2test` again and loops until the integer value is 0
- A reaps B and prints the 0 integer value

- To prove the pages are being freed by `FreeSharedPage()`, the original integer value 20 is greater than 16
- `exec()` does not free the page so it does not remove the procs from `procs_sh_page` in `all_shared_pages`
- Thus, if they were not being freed, we'd get an error returned as such:

```
$ shpg4test
a now points to 20
a now points to 19
a now points to 18
a now points to 17
a now points to 16
a now points to 15
a now points to 14
a now points to 13
a now points to 12
a now points to 11
a now points to 10
a now points to 9
a now points to 8
a now points to 7
ERROR: too many processes sharing this page
pid 5 shpg5test: trap 14 err 5 on cpu 1 eip 0x27 addr 0xffffffff--kill proc
From first process: a now points to 6
```

Figure 3 demonstrates that when calling `FreeSharedPage()`, the process is removed from the array holding the list of processes accessing the page, but `kfree()` is not called on the physical until no other process is accessing it.

---

shpg6test

```
$ shpg6test
At c[0] on page 0 is 333
At c[1] on page 1024 is 333
```

- This test makes sure it is mapping the pages in the right spots in the address space
- The process requests 1 page under key `111` so `sh_bit_map` looks as 'A 0 0 0 0 ...' where 'A' is the 1 bit belonging to key `111`. Then it writes 111 across the page.
- Next, it requests 1 more page under key `222` so `sh_bit_map` looks as 'A B 0 0 0 ...' where 'B' is the 1 bit belonging to key `222`. Then it writes 222 across the page.
- Next, it calls `FreeSharedPage(111)` so `sh_bit_map` looks as '0 B 0 0 0 ...'
- Lastly, it requests 2 more pages under key `333` so `sh_bit_map` looks as '0 B C C 0 ...' where 'C' are the bits belonging to key `333`. Then it writes 333 across the page.
- To prove they are placed correctly and not as 'C B C 0 0 ...' the process reads from the pages using pointer arithmetic

Figure 4 demonstrates that when multiple pages are being requested, they are placed consecutively in the user address space such that they can be written on using the return virtual address.