Implementation:

Data structure :

-To maintain page reference count

```
struct {
  struct spinlock lock;
  uchar cnt[PHYSTOP/PGSIZE];
} pgrefcnt;
```

-To maintain number of free pages

```
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
  uint sz;                    // this field was added
} kmem;
```

Modified files:

vm.h

-kvmalloc() : initialize pgrefcnt.lock and pgrefcnt.cnt[]
-inituvm(..) and allocuvm(..) : initialize pgrefcnt.cnt for corresponding allocated pages to 1
-deallocuvm(..) : decrement pgrefcnt.cnt for corresponding pages by one and calling kfree if pgrefcnt.cnt for page is 0.
-copyuvm(..) : copying the pgdir of parent process and backing the write permissions on the page in pte itself and resetting PTE_W bit. If anything goes wrong we reset the pgdir for parent. In the end we call lcr3() to reload the page table which might have changed.
-pagefault() : check it the proc exists and pte exists and user has correct priviege(PTE_U) and that the virtual address is below KERNBASE.
Then if page ref cnt > 1 the allocate new page to the process and edit the pte accordingly also restore the write permissions on the page and flush the PTE_COW and PTE_WCOW bits.
In the end reload page table with lcr3() or kill the process if error was detected.

kalloc.c

- added sz(uint) to struct kmem
- sz is incremented in kfree and decremented in kalloc
- sz is initialized to 0 in kvmalloc

trap.c

- added a new case for handling page fault which calls pagefault() of vm.c

mmu.h

- added helper macros for indexing in pgrefcnt.cnt[] given pa
- added new flag bits PTE_COW and PTE_WCOW for page table entries denoting page is not yet serviced for CoW and the write permission before fork resp.

defs.h
- added knumfreepages(), pagefault() function declarations.

sysfile.h for a #define for syscall getnumfreepages()

---

sysproc.c for syscall getnumfreepages()
usys.S for syscall getnumfreepages()
Makefile for adding testcow

Testing the implementation:
testcow.c forks two children. Note that a printf call will write to user stack and hence will itself cause pagefault and pages are used in creating page tables also.
First child makes small update in 'pid' and does not cause new page to be allocated.
Second child makes a big update on array of size 4*PGSIZE hence causing 5 new pages to be allocated.(4-5 pages can be spanned by the array)
Hence pages are allocateed only when process tries to write into them as required by CoW.

Output:
#free-pages [in parent(before forks)]= 56771
#free-pages [in child-1 (before)]= 56635
#free-pages [in child-1 (after)]= 56635
#free-pages [in child-2 (before)]= 56635
#free-pages [in child-2 (after)]= 56630
#free-pages [in parent(after all reaps)]= 56771