CS344 Assignment 3

Group members:

- ★ Saket Kumar Singh 190101081
- ★ Pradnesh Prasad Kalkar 190101103
- ★ Mesharya M Choudhary 190101053
- ★ Parag Panigrahi 190101111

Part A (Lazy Memory Allocation)

A process invokes the sbrk() system call to indicate that it needs extra memory as compared to what is currently assigned. The system call used the growproc() function which calls the allocuvm() function. The latter is responsible for allocating the desired extra memory by allocating pages and also mapping the virtual addresses appropriately to their corresponding physical addresses in the page tables.

So, we started the implementation of part A with the patch already provided to us. The implementation of the sys_sbrk() in sysproc.c is modified. The call to the growproc() function is commented.

```c
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;
  myproc()->sz += n;

  // delaying the memory allcoation by commenting growproc -> lazy allcoation
  //  if(growproc(n) < 0)
  //    return -1;

  return addr;
}
```

The process is just being tricked that it has been allocated the memory because we are just increasing the value of the sz field (the size of the process) using proc->sz statement in sysproc.c and not actually allocating any memory to the process. Hence, when an attempt is made to access the above requested memory, there won't be any page table entry corresponding to the virtual address (miss in the page table) giving rise to a page fault.

So, in lazy memory allocation, whenever these page faults occur, then and only then one page (from the list of free physical memory pages available) is allocated to the process and appropriate page table entries are added/updated corresponding to this allocation. This is also known as demand paging.

```
Booting from Hard Disk...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ ls
pid 3 sh: trap 14 err 6 on cpu 1 eip 0x11c8 addr 0x4004--kill proc
$ _
```

Handling Page Fault in trap.c

We have seen that when the process calls the sbrk() system call, it is being tricked of the memory allocation; i.e. the memory is not actually allocated to the process but we just increase its size attribute. So, when this process tries to access the above requested memory (which it thinks that it has been already bought inside), it encounters a PAGE FAULT, thus generating a trap T_PGFLT to the kernel.
So, to handle the page fault, we call the function allocateMemoryPage() defined in vm.c which actually performs the memory allocation. This function takes 2 arguments -
rcr2() returns the virtual address that caused the page fault.
myproc()->pgdir returns a pointer to the page directory of the process. The page directory of the process is nothing but the outer level of the 2-level page table used.

```
case T_PGFLT:
  // rcr2() is giving the virtual address
  allocateMemoryPage(myproc()->pgdir, rcr2());
  break;
```

allocateMemoryPage() in vm.c - Implementation details:

This function (as the name suggests) is responsible for the actual allocation of pages and corresponding updations in the page table.
Firstly, using PGROUNDDOWN(va), it rounds the virtual address (which was responsible for the page fault) to the start of the page boundary.
Then a call is made to kalloc() to get the list of physical addresses available. If it is available then the memory is filled with 0s using memset. Otherwise, if kalloc() returns 0, then it indicates that no memory is available, and thus memory allocation is not possible.
Then the mappages() function is called with virtual address, page size and physical address V2P(mem) as parameters. Permissions for the pages are also set to writable using PTE_W and accessible by user processes using PTE_U in the parameters.
If at any point mappages causes any error, then all the allocated memory is freed using the kfree() function.

```c
// Allocate a new memory page to the process
void
allocateMemoryPage(pde_t *pgdir, uint va)
{
  char *mem;
  uint a;

  a = PGROUNDDOWN(va);
  mem = kalloc();

  if (mem == 0) {
    cprintf("allocuvm out of memory (3)\n");
    return;
  }

  memset(mem, 0, PGSIZE);
  if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
    cprintf("allocuvm out of memory (4)\n");
    kfree(mem);
  }
}
```

Correctness in output of "ls" and "echo" commands after doing lazy allocation:

After doing all these changes, we can see that the commands "ls" and "echo hi" are working as expected. The pages are getting allocated on demand, when the process actually requires them, and hence the output is working perfectly as can be seen below.

```
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ echo hi
hi
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 16320
echo           2 4 15176
forktest       2 5 9480
grep           2 6 18536
init           2 7 15756
kill           2 8 15204
ln             2 9 15056
ls             2 10 17688
mkdir          2 11 15300
rm             2 12 15280
sh             2 13 27912
stressfs       2 14 16192
usertests      2 15 67296
wc             2 16 17056
zombie         2 17 14868
console        3 18 0
$
```

Part B (Xv6 Memory)

Q1. How does the kernel know which physical pages are used and unused?

Ans: A linked list of free pages called freelist (which is the head of the linked list) is maintained by the kernel in struct kmem in the file kalloc.c. Initially this list is empty. All physical pages are appended to this linked list whenever they are initialized or are freed. The xv6 OS adds 4MB of free pages to the list by calling kinit1 through main.

```c
struct {
  struct spinlock lock;
  int use_lock;
  struct run *freelist;
} kmem;

```

Q2. What data structures are used to answer this question?

Ans: The data structure used for the purpose is the linked list named freelist as shown in the figure above. If we observe the declaration of the linked list above (struct run* freelist), we can see that every node of the linked list is a structure called struct run which is also defined in the file kalloc.c as shown in the image below. With the help of this struct run data structure, the kernel tracks the available free pages. The structure stores a pointer to the next free page thus forming a linked list of free pages. The pointer to the next free page is stored within the page.

```
struct run {
  struct run *next;
};
```

Q-3 Where do these reside?

Ans: This linked list is declared inside the file kalloc.c inside the kmem structure. Every node is of the type struct run whose declaration is also present in the file kalloc.c as shown in the figure above. The structures themselves reside in the kernel memory.

Q-4 Does xv6 memory mechanism limit the number of user processes?

Ans: The size of the process table is limited (NPROC which is 64 by default and is defined in param.h). Due to this the number of user processes is limited as well.

Q-5 If so, what is the lowest number of processes xv6 can have at the same time (assuming the kernel requires no memory whatsoever)?

Ans: When the xv6 operating system boots up, there is only one process named initproc. This initproc process forks the sh(i.e. shell) process which forks other upcoming user processes. We know that the maximum physical memory is 240MB (PHYSTOP). Now, consider any process; it can have a virtual address space of 2GB (KERNBASE) and hence, since 2GB > 240MB, one process can eat up all of the physical memory. Hence, the lowest number of processes present in xv6 at the same time = 1. Moreover, there cannot be zero processes after boot since, all of the user interactions need to be done using user processes which are forked from initproc/sh.

Task 1: kernel processes:

The function create_kernel_process() which is defined in proc.c is responsible for the creation of Kernel process and its addition to the processes queue. It finds an empty slot in the process table and assigns it to the newly created process. This is followed by the allocation of kernel stack for trapframe, putting exit() function which will be called upon return from the context after trapframe in stack, setting up of context and making eip equal to entrypoint function. Next the page table for the new process is created by calling setupkvm() and the name of the process is set as the input parameter name, its parent is set to initproc and its state changed to RUNNABLE.

```c
// This function create a kernel process and add it to the processes queue.
void create_kernel_process(const char *name, void (*entrypoint)())
{
  int i;
  char *sp;
  bool found = false;

  acquire(&ptable.lock);

  for (i = 0; i < NPROC; i++){
    struct proc process = ptable.proc[i];
    if (process.state == UNUSED){
      found = true;
      break;
    }
  }

  if (!found){
    release(&ptable.lock);
    return;
  }
  else{
    ptable.proc[i].state = EMBRYO;
    ptable.proc[i].pid = nextpid;
    nextpid = nextpid + 1;

    release(&ptable.lock);

    // Allocate kernel stack.
    ptable.proc[i].kstack = kalloc();
    if (ptable.proc[i].kstack == 0){
      ptable.proc[i].state = UNUSED;
      return;
```

```c
    ptable.proc[i].state = UNUSED;
    return;
  }
  sp = ptable.proc[i].kstack + KSTACKSIZE;

  // Leave room for trap frame.
  sp -= sizeof *ptable.proc[i].tf;
  ptable.proc[i].tf = (struct trapframe*) sp;

  // Set up new context to start executing at forkret,
  // which returns to trapret.
  sp -= 4;
  *(uint*)sp = (uint)exit; // end the kernel process upon return from entrypoint()

  sp -= sizeof *ptable.proc[i].context;
  ptable.proc[i].context = (struct context*)sp;
  memset(ptable.proc[i].context, 0, sizeof *ptable.proc[i].context);
  (ptable.proc[i].context)->eip = (uint)entrypoint;

  if((ptable.proc[i].pgdir = setupkvm()) == 0){
    panic("kernel process: out of memory?");
  }

  ptable.proc[i].sz = PGSIZE;
  ptable.proc[i].parent = initproc;
  ptable.proc[i].cwd = idup(initproc->cwd);
  // cprintf("%s %d\n",name,sizeof(ptable[i].name));
  safestrcpy(ptable.proc[i].name, name, sizeof(ptable.proc[i].name));

  acquire(&ptable.lock);
  ptable.proc[i].state = RUNNABLE;
  release(&ptable.lock);
```

Once the process returns from the entrypoint() the exit() function terminates it and prevents it from returning to the user mode. The create_kernel_process() function is called by forkret() which is in turn called from initprocess and it creates two kernel processes - swapoutprocess , swapinprocess.

```
// will swtch here.  "Return" to user space.
void
forkret(void)
{
  static int first = 1;
  // Still holding ptable.lock from scheduler.
  release(&ptable.lock);

  if (first) {
    // Some initialization functions must be run in the context
    // of a regular process (e.g., they call sleep), and thus cannot
    // be run from main().
    first = 0;
    iinit(ROOTDEV);
    initlog(ROOTDEV);
    create_kernel_process("swapoutprocess",swapoutprocess);
    create_kernel_process("swapinprocess",swapinprocess);
  }

  // Return to "caller", actually trapret (see allocproc).
}
```

Task 2: Swapping out mechanism

The proc data structure has been modified to include two new fields:

1. satisfied: Indicates whether swap out request has been swapped out for a given process.
2. trapva: Stores virtual address of the page causing page fault for a given process.

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  int satisfied;               // If zero, page request not satisifed
  uint trapva;                 // VA at which pagefault occurred
};
```

The kernel process named swapoutprocess is responsible for swapping out pages from virtual memory of a given process whenever needed. It also supports a requestqueue which is implement as a circular queue ADT with the following features:

1. front, rear: pointers indicating the start and the end of the circular queue.
2. size: represents the size of the circular queue.
3. reqchan: The channel on which all the requesting processes for swapping out of a page wait on.
4. qchan: The channel on which the swapout function waits when there are no processes to serve on the request queue.
5. lock: A spinlock to protect the shared access of the swapqueue among different processes.
6. queue: A queue storing the PCB of the processes queued for swapping out requests.
7. enqueue(): Method to add a process to the queue.
8. dequeue(): Method to remove a process from the queue.

```
struct swapqueue{
    struct spinlock lock;
    int front;
    int size;
    int rear;
    char* reqchan;
    char* qchan;
    struct proc* queue[NPROC+1];
};
```

```c
// Enqueue function for the queues
void enqueue(struct swapqueue* sq, struct proc* np){
  if(sq->size == NPROC){
    return;
  }
  sq->rear = (sq->rear + 1) % NPROC;
  sq->queue[sq->rear] = np;
  sq->size++;
}
```

```c
// Dequeue function for the queues
struct proc* dequeue(struct swapqueue* sq){
  if (sq->size == 0){
    return 0;
  }

  struct proc* next = sq->queue[sq->front];
  sq->front = (sq->front + 1) % NPROC;
  sq->size = sq->size - 1;

  if(sq->size == 0){
    sq->front = 0;
    sq->rear = NPROC - 1;
  }

  return next;
}
```

submitReqToSwapOut() function does the following:

1. Add the requesting process to the queue.
2. Wake the swapoutprocess to swap out a page of the requesting process for accommodation of space of a new page of the requesting process.

3. Make the requesting process sleep until the request is served.
4. Set the satisfied field to 0.

```c
// Submits a request for a free page to the swapout process
void submitReqToSwapOut(){
  struct proc* p = myproc();
  // cprintf("submitReqToSwapOut %d\n",p->pid);
  char my_pid[3];
  my_pid[1] = '0' + p->pid%10;
  my_pid[0] = (p->pid/10 ? '0' + p->pid/10 : ' ');
  my_pid[2] = 0;
  cprintf("| Submit Request to SwapOut |  %s | -  |         Process %s is queued to swapout           |\n", my_pid, my_pid);

  acquire(&ptable.lock);
  acquire(&swap_out_queue.lock);
  p->satisfied = 0;
  enqueue(&swap_out_queue, p);    // Enqueues the process in the Swapout queue
  wakeup1(swap_out_queue.qchan); // Wakes up the Swapout process
  release(&swap_out_queue.lock);

  while(p->satisfied==0)  // Sleep process till not satisfied
    sleep(swap_out_queue.reqchan, &ptable.lock);
  release(&ptable.lock);
  return;
}
```

The swapoutprocess() is a kernel process which keeps running as long as there are requests to serve and sleeps when there are none. To serve a request it does the following:

1. Dequeues a process from the queue.
2. Chooses a victim according to the replacement policy and evicts a frame from it.
3. When a frame is found and allocated, satisfaction is set to 1.
4. Wake Up the corresponding process.
5. Sleep all the processes on the corresponding channel.

```c
// Entry point of the swapout process
void swapoutprocess(){
  sleep(swap_out_queue.qchan, &ptable.lock);

  while(1){
    // cprintf("\n\nEntering swapout\n");
    cprintf("|      Swapout Resumes      |  -  | -  |   Swapout queue is non-empty => start execution   |\n");
    acquire(&swap_out_queue.lock);
    while(swap_out_queue.size){
      while (flimit >= NOFILE)    // Edge case handling
      {
        cprintf("flimit \n");
        wakeup1(swap_out_queue.reqchan);
        release(&swap_out_queue.lock);
        release(&ptable.lock);
        yield();
        acquire(&swap_out_queue.lock);
        acquire(&ptable.lock);
      }

      struct proc *p = dequeue(&swap_out_queue); // Dequeue process from queue

      if(!chooseVictimAndEvict(p->pid)) // Edge case handling
      {
        wakeup1(swap_out_queue.reqchan);
        release(&swap_out_queue.lock);
        release(&ptable.lock);
        yield();
        acquire(&swap_out_queue.lock);
        acquire(&ptable.lock);
      }
      p->satisfied = 1;     // When frame found set satified to true
    }

    wakeup1(swap_out_queue.reqchan);   // The the corresponding process
```

Pseudo LRU replacement policy is used for selecting the victim frame for eviction. To decide the preference order, the accessed bit and the dirty bit are concatenated to form an integer and the following order is maintained:

0(00) < 1(01) < 2(10) < 3(11)

After choosing a victim frame, the present bit is unset for the corresponding PTE and the process is suspended (set to SLEEPING state) until the page has been written on disk. The 7'th bit (which is unset by default) is the set to indicate successful swapping out of the target frame. The function returns 1 on successful eviction of the victim frame and returns 0 otherwise.

```c
// Chooses a victim frame using LRU and evicts it
int chooseVictimAndEvict(int pid){

  struct proc* p;
  struct victim victims[4]={{0,0,0},{0,0,0},{0,0,0},{0,0,0}};
  pde_t *pte;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state == UNUSED|| p->state == EMBRYO || p->state == RUNNING || p->pid < 5|| p->pid == pid)
        continue;

      for(uint i = PGSIZE; i < p->sz; i += PGSIZE){
        pte = (pte_t*)getpte(p->pgdir, (void *) i);
        if( !((*pte) & PTE_U) || !((*pte) & PTE_P) )
          continue;
        int idx =(((*pte)&(uint)96)>>5);
        if(idx>0&&idx<3)
          idx=3-idx;
        victims[idx].pte = pte;
        victims[idx].va = i;
        victims[idx].pr = p;
      }
  }
  for(int i=0;i<4;i++)
  {
    if(victims[i].pte != 0)
    {
      pte = victims[i].pte;
      int origstate = victims[i].pr->state;
      char* origchan = victims[i].pr->chan;
      victims[i].pr->state = SLEEPING;
      victims[i].pr->chan = 0;
      uint reqpte = *pte;
```

```
		}
	}
	for(int i=0;i<4;i++)
	{
		if(victims[i].pte != 0)
		{
			pte = victims[i].pte;
			int origstate = victims[i].pr->state;
			char* origchan = victims[i].pr->chan;
			victims[i].pr->state = SLEEPING;
			victims[i].pr->chan = 0;
			uint reqpte = *pte;
			*pte = ((*pte)&(~PTE_P));
			*pte = *pte | ((uint)1<<7);

			if(victims[i].pr->state != ZOMBIE){
				release(&swap_out_queue.lock);
				release(&ptable.lock);
				writepagetoswapoutfile(victims[i].pr->pid, (victims[i].va)>>12, (void *)P2V(PTE_ADDR(reqpte)));
				acquire(&swap_out_queue.lock);
				acquire(&ptable.lock);
			}
			kfree((char *)P2V(PTE_ADDR(reqpte)));
			lcr3(V2P(victims[i].pr->pgdir));
			victims[i].pr->state = origstate;
			victims[i].pr->chan = origchan;
			return 1;
		}
	}
	return 0;
}
```

The kalloc() function is used to allocate the 4096-byte of physical memory to meet swapping on demand. submitReqToSwapOut is called until a free frame is obtained.

```c
char*
kalloc(void)
{

  if(kmem.use_lock){
    acquire(&kmem.lock);
  }

  struct run *cur = kmem.freelist;

  while (!cur) {
    if (kmem.use_lock){
      release(&kmem.lock);
    }

    submitReqToSwapOut();

    if (kmem.use_lock){
      acquire(&kmem.lock);
    }

    cur = kmem.freelist;
  }

  if (cur){
    kmem.freelist = cur->next;
  }

  if (kmem.use_lock) {
    release(&kmem.lock);
  }

  char* char_cur = (char *)cur;
  return char_cur;
```

write_page() is used to write the frame contents into the disk. The file name is used as PID_VA[20:].swp where PID is the process ID and the VA[20:] are the first 20 bits of the virtual address corresponding to the evicted page. Internal implementation of write_page() uses open_file() to open/create files filewrite() to write the contents.

Task 3: Swapping in mechanism

The first function we encounter while swapping-in is swapinprocess(). This function regularly iterates over the swapin queue and finds a swap-in request to fulfill. It first gets a free frame in main memory using kalloc(). After getting a free frame it reads the swapped-out page from the disk into the newly obtained free frame. Then using swapInMap() it updates the flags and physical page numbers(PPN) in the corresponding Page Table Entry. Then the corresponding process is woken up.

When all the requests in the swapinqueue() are fulfilled, this process goes into SLEEPING state. Before this all the appropriate locks are released.

read_page() functions helps in reading the corresponding page that is swapped out of the process's Page Table Entry into the buffer mem. It computes the filename and then calls fileread() which reads the content of the file.

```c
// Entry point of the swapin process
void swapinprocess(){
  sleep(swap_in_queue.qchan, &ptable.lock);
  while(1){
    // cprintf("\n\nEntering swapin\n");
    cprintf("|      Swapin Resumes       |  -  | -  |   Swapin queue is non-empty => start execution    |\n");
    acquire(&swap_in_queue.lock);
    while(swap_in_queue.size){
      struct proc *p = dequeue(&swap_in_queue);
      flimit--;
      release(&swap_in_queue.lock);
      release(&ptable.lock);

      char* mem = kalloc();
      read_page(p->pid,((p->trapva)>>12),mem);

      acquire(&swap_in_queue.lock);
      acquire(&ptable.lock);
      swapInMap(p->pgdir, (void *)PGROUNDDOWN(p->trapva), PGSIZE, V2P(mem));
      wakeup1(p->chan);
    }
    // cprintf("\n\n");
    release(&swap_in_queue.lock);
    sleep(swap_in_queue.qchan, &ptable.lock);
  }
```

On the event of occurrence of page fault, if the page fault occurs due to the swapping out of an earlier page, the function submitReqToSwapIn() is called. This function does the following:

1. Enqueue the running process into the swapin queue.
2. Suspend the running process.

```c
// Submits a request to the swapin process
void submitReqToSwapIn(){
  struct proc* p = myproc();
  // cprintf("submitReqToSwapIn %d\n",p->pid);
  char my_pid[3];
  my_pid[1] = '0' + p->pid%10;
  my_pid[0] = (p->pid/10 ? '0' + p->pid/10 : ' ');
  my_pid[2] = 0;
  cprintf("| Submit Request to SwapIn  |  %s | -  |         Process %s is queued to swapin            |\n", my_pid, my_pid);

  acquire(&ptable.lock);
  acquire(&swap_in_queue.lock);
    enqueue(&swap_in_queue, p);   // Enqueues the process in the Swapin queue
    wakeup1(swap_in_queue.qchan); // Wake up the Swapin process
  release(&swap_in_queue.lock);

  sleep((char *)p->pid, &ptable.lock);  // Suspend the process
  release(&ptable.lock);
  return;
}
```

After the process execution is done and it is about to exit, it is ensured that the swap out pages which were written earlier onto the disk are now deleted to restore the original state. The deleteSwapoutPageFiles() function is used to perform this task. It does the following:

1. Iterates through the files list of the process to swap out.
2. If the file is not already deleted, then the function deletes it.

```
void deleteSwapoutPageFiles()
{
  acquire(&ptable.lock);
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->state == UNUSED) continue;
    if(p->pid==2||p->pid==3)
    {
      for(int fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
          struct file* f;
          f = p->ofile[fd];

          if(f->ref < 1) {
            p->ofile[fd] = 0;
            continue;
          }
          release(&ptable.lock);
```

Task 4: Sanity Test

To test whether the memory swaps are working correctly or not, we wrote the user program memtest.c. This forks 20 child processes, each of which then iterates 20 times and requests for 4KB using malloc().

Note: We have increased the number of iterations in child processes from 10 to 20 because with 10 loops, the entire memory was not used up. This is because the initialisation requirement of the kernel restricts us so that we can't set the value of PHYSTOP below 4MB, due to which we require more than 10 loops for the memory to be full and the swapper to work.

```c
    for (int i = 1; i <= NUM_CHILD; i++){
        if (!fork()) {
            Child_Function(i);
        }
    }

    for (int i = 1; i <= NUM_CHILD; i++){
        wait();
    }
```

At each memory location, we are using the following function to store the data:

If the child number = i ,

Iteration number = j ,

Byte number = k

Then the value stored in that memory location is (i*j*k)%128

```c
void Child_Function(int i) {
    char *ptr[NUM_ITER];

    // Allocates 4KB pages to each of the char pointer
    for(int j = 0; j < NUM_ITER; j++){
        ptr[j] = (char *)malloc(PAGE_SIZE);
    }

    // Assign values to the allocated memory
    for (int j = 0; j < NUM_ITER; j++) {
        for (int k = 0; k < PAGE_SIZE; k++){
            ptr[j][k] = (i * j * k) % 128;
        }
    }

    // Error detection
    for (int j=0; j < NUM_ITER; j++) {
        for (int k=0; k < PAGE_SIZE; k++) {
            if (ptr[j][k] != (i * j * k) % 128){
                printf(1, "Error at i = %d, j = %d, k = %d, val = %c\n", i, j, k, ptr[j][k]);
            }
        }
    }
```

Then we iterate over again and check whether the value we stored earlier is stored correctly or not. If the value doesn't match then we print an error message in the console.

Output:

```
$ memtest
---------------------------------------------------------------------------------------------------
|             Event             | PID | VA |                       Remark                        |
---------------------------------------------------------------------------------------------------
| Submit Request to SwapOut |  69 |  -  |          Process 69 is queued to swapout             |
| Submit Request to SwapOut |  86 |  -  |          Process 86 is queued to swapout             |
| Submit Request to SwapOut |  87 |  -  |          Process 87 is queued to swapout             |
|      Swapout Resumes      |  -  |  -  |   Swapout queue is non-empty => start execution     |
|     Page File Creation    |  86 |  2  |       Contents of page  2 saved in 86_2.swp          |
|     Page File Creation    |  87 |  2  |       Contents of page  2 saved in 87_2.swp          |
|     Page File Creation    |  85 | 26  |       Contents of page 26 saved in 85_26.swp         |
| Submit Request to SwapOut |  69 |  -  |          Process 69 is queued to swapout             |
| Submit Request to SwapOut |  86 |  -  |          Process 86 is queued to swapout             |
|      Swapout Resumes      |  -  |  -  |   Swapout queue is non-empty => start execution     |
| Submit Request to SwapOut |  87 |  -  |          Process 87 is queued to swapout             |
|     Page File Creation    |  85 | 25  |       Contents of page 25 saved in 85_25.swp         |
|     Page File Creation    |  85 | 24  |       Contents of page 24 saved in 85_24.swp         |
|     Page File Creation    |  85 | 23  |       Contents of page 23 saved in 85_23.swp         |
| Submit Request to SwapOut |  86 |  -  |          Process 86 is queued to swapout             |
| Submit Request to SwapOut |  69 |  -  |          Process 69 is queued to swapout             |
| Submit Request to SwapOut |  87 |  -  |          Process 87 is queued to swapout             |
|      Swapout Resumes      |  -  |  -  |   Swapout queue is non-empty => start execution     |
|     Page File Creation    |  85 | 22  |       Contents of page 22 saved in 85_22.swp         |
|     Page File Creation    |  85 | 21  |       Contents of page 21 saved in 85_21.swp         |
|     Page File Creation    |  85 | 20  |       Contents of page 20 saved in 85_20.swp         |
| Submit Request to SwapOut |  86 |  -  |          Process 86 is queued to swapout             |
| Submit Request to SwapOut |  69 |  -  |          Process 69 is queued to swapout             |
|      Swapout Resumes      |  -  |  -  |   Swapout queue is non-empty => start execution     |
| Submit Request to SwapOut |  87 |  -  |          Process 87 is queued to swapout             |
|     Page File Creation    |  85 | 19  |       Contents of page 19 saved in 85_19.swp         |
|     Page File Creation    |  85 | 18  |       Contents of page 18 saved in 85_18.swp         |
|     Page File Creation    |  85 | 17  |       Contents of page 17 saved in 85_17.swp         |
| Submit Request to SwapOut |  86 |  -  |          Process 86 is queued to swapout             |
| Submit Request to SwapOut |  69 |  -  |          Process 69 is queued to swapout             |
| Submit Request to SwapOut |  87 |  -  |          Process 87 is queued to swapout             |
|      Swapout Resumes      |  -  |  -  |   Swapout queue is non-empty => start execution     |
|     Page File Creation    |  85 | 16  |       Contents of page 16 saved in 85_16.swp         |
|     Page File Creation    |  85 | 15  |       Contents of page 15 saved in 85_15.swp         |
|     Page File Creation    |  85 | 14  |       Contents of page 14 saved in 85_14.swp         |
|        Page Fault         |  -  |  -  | Page fault has occured due to insufficient memory |
```
