# Assignment 3

Group C25

Anindya Vijayvargeeya - 200101015
Gunjan Dhanuka - 200101038
Pranjal Singh - 200101084

## Part A: Lazy Memory Allocation

In the original version of xv6, whenever an application asks the kernel for heap memory using the **sbrk()** system call, it allocates physical memory and maps it into the process's virtual address space. To add support for the lazy allocation feature we need to delay the memory requested by sbrk() until the process actually uses it.

sbrk uses the **growproc()** function to increase the space given to a process, which in turn calls **allocuvm()** to allocate the desired extra memory by allocating extra pages and mapping virtual addresses to the physical addresses in the page tables.

So first of all, we comment out the growproc() inside the sbrk() system call in the sysproc.c file.

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

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

 // if(growproc(n) < 0)
 //   return -1;
 return addr;
}
```

After commenting out the growproc(), we change the size variable for the current process to essentially “fool” it into believing that extra memory has been allocated to it, while actually it is not.

When this process tries to access this memory, it will encounter a Page Fault and generate a **T_PGFLT** trap to the kernel. We handle this in the **trap.c** by calling **handlePageFault().**

```c
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);

int handlePageFault(){
 char *mem;
 uint start_addr;

 start_addr = PGROUNDDOWN(rcr2());
 mem = kalloc();
 if(mem == 0){
   cprintf("kalloc couldn't allocate\n");
   return -1;
 }
 memset(mem, 0, PGSIZE);
 if(mappages(myproc()->pgdir, (char*)start_addr, PGSIZE, V2P(mem),
PTE_W|PTE_U) < 0){
   cprintf("mappages couldn't map\n");
   kfree(mem);
   return -1;
 }

 return 0;
}
```

The **rcr2()** gives us the value of the Page Fault Linear Address (PFLA) which is basically the address the process tried to access. **PGROUNDDOWN** gives us the start address of the page and then we can use **kalloc()** to allocate one page of physical memory from a linked list of free pages and returns the pointer. Then we set all the values of that page to 0 using **memset**.
Now we need to map the page we obtained from **kalloc()** to the start_addr which can be done using **mappages()**. [We needed to remove the *static* keyword from before the function definition in vm.c and then declare the prototype of mappages() in trap.c to use it there.]

We pass the page table of the current process, the virtual start address of the data, the size of the data (here equal to one page size), physical memory at which the physical page is actually located (which can be obtained using the **V2P macro**, which basically subtracts **KERNBASE** from mem) and permissions applicable to the page table entries (**PTE_W** means writable and **PTE_U** stands for User access).

**vm.c:**

```c
int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
 char *a, *last;
 pte_t *pte;

 a = (char*)PGROUNDDOWN((uint)va);
 last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
 for(;;){
   if((pte = walkpgdir(pgdir, a, 1)) == 0)
     return -1;
   if(*pte & PTE_P)
     panic("remap");
   *pte = pa | perm | PTE_P;
   if(a == last)
     break;
   a += PGSIZE;
   pa += PGSIZE;
 }
 return 0;
}
```

Let’s understand what goes on in mappages() in the vm.c file. We have two variables a and last, which correspond to the start and end addresses respectively. Then we have a loop which iterates from the start to the last and loads pages one-by-one. For every page, we use **walkpgdir()** to load it into the page table.

**vm.c:**

```c
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
 pde_t *pde;
 pte_t *pgtab;

 pde = &pgdir[PDX(va)];
 if(*pde & PTE_P){
   pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
 } else {
   if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
     return 0;
   // Make sure all those PTE_P bits are zero.
   memset(pgtab, 0, PGSIZE);
   // The permissions here are overly generous, but they can
   // be further restricted by the permissions in the page table
   // entries, if necessary.
   *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
 }
 return &pgtab[PTX(va)];
}
```

**walkpgdir()** takes in a page directory and a virtual address as input and returns the page table entry of that virtual address after placing it inside the page table. Since we have a two-level page table, there are two values we must get to reach the entry. The **macro PDX** gives the Page Directory Index from the first 10 bits of the virtual address. This gives the entry which points to the relevant page table that contains the page for this virtual address. Then, the **macro PTX** gives the next 10 bits to get the corresponding entry in the page table obtained above and returns it.

If the page table corresponding to the page directory entry is already present in the memory, we then store the pointer to its first entry in pgtab ( **PTE_ADDR** is used to zero the last 12 bits thereby making the offset zero). If it isn't already present, we create a page table and then set all the bits to zero. Then it returns a pointer to the page table entry corresponding to the virtual address.

Now the mappages() knows the page table entry to where the current virtual address has to be mapped. It checks the **PTE_P** bit (which implies Present if set) which means that the entry is already mapped to some virtual address. Thus, it raises an error that a remap will occur here. In case of no error, it maps the page table entry to the virtual address. It then also sets the permission bits and **PTE_P** bit indicating that the current page table entry is now mapped to a virtual address.

## Part B : Swapping

Answers to the questions :

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

Kernel maintains a linked list of free pages to do this.

➔ What data structures are used to answer this question?

A linked list named freelist.

➔ Where do these reside?

Its declared in kalloc.c file.

➔ Does xv6 memory mechanism limit the number of user processes

The process table has atmax NPROC (=64) entries, which limit the number of processes to 64.

➔ If so, what is the lowest number of processes xv6 can ‘have’ at the same time (assuming the kernel requires no memory whatsoever)?
Since all user interactions are to be done by user processes only, there will be atleast one process constantly running in the OS from the time it boots. Hence, lowest number of processes will be 1.

## Task 1 : Kernel process

**proc.c:**

```
// to create a kernel process and add it to the process queue
void create_kernel_process(const char *name, void (*entrypoint)())
{

 // creating a new process and assign it a spot in the ptable
 struct proc *p = allocproc();

 if (p == 0)
 {
   panic("create_kernel_process failed.");
 }

 // Setup the kernel part of the process's page table that maps virtual
addresses above KERNBASE to physical addresses between 0 and PHYSTOP

 p->pgdir = setupkvm();
 if (p->pgdir == 0)
 {
   panic("setupkvm failed and kernel page table not created.");
 }

 // eip - pointer of the next instruction to be executed

 p->context->eip = (uint)entrypoint;

 safestrcpy(p->name, name, sizeof(p->name));

 acquire(&ptable.lock);
 p->state = RUNNABLE;
 release(&ptable.lock);
}
```

In this task we had to create a **create_kernel_process()** function in proc.c. First we create a new process using the **allocproc()** function which looks in the process table for an UNUSED proc, and changes its state to EMBRYO and initializes it. If it returns 0, then it means that no such process could be created. Then we set up the kernel part of the process's page table using **setupkvm()** that will map the virtual addresses above **KERNBASE** to physical addresses between 0 and **PHYSTOP**. If the page directory of the process remains 0, it means that the page table couldn't be created.

We do not need to initialize a trapframe for a kernel level process, since it cannot be evicted out anyways. Trapframes are used to store user space registers, and since there is no userspace no size assignment is required.

Then we set the **eip** (pointer of the next instruction to be executed) to the input function, and then copy the input name to the name of the process using **safestrcpy** (which is just like strcpy but guarantees NULL-termination).

Finally, we set the state of the process as **RUNNABLE** by first acquiring the lock of the process table and then releasing it after the state is changed.

## Task 2 : Swapping out mechanism

Here we need to maintain a request queue for the processes that failed to be allocated, and create a function to handle swapping out of processes, that will take out some pages and write it to disk, and thus generate a free page.

We first start with implementing a queue:
**proc.c**:

```
struct rq{
 struct spinlock lock;
 struct proc* queue[NPROC];
 int s;
 int e;
};
```

```
int rpush( struct rq* q, struct proc *p){

 acquire(&q->lock);
 if((q->e+1)%NPROC==q->s){
   release(&q->lock);
   return 0;
 }
 q->queue[q->e]=p;
 q->e++;
 (q->e)%=NPROC;
 release(&q->lock);
  return 1;
}
```

```
struct proc* rpop( struct rq* q){

 acquire(&q->lock);
 if(q->s==q->e){
   release(&q->lock);
   return 0;
 }
 struct proc *p=q->queue[q->s];
 (q->s)++;
 (q->s)%=NPROC;
 release(&q->lock);

 return p;
}
```

```
void
pinit(void)
{
 initlock(&ptable.lock, "ptable");
 initlock(&rqueue_out.lock, "rqueue");
 initlock(&sleeping_channel_lock, "sleeping_channel");
 initlock(&rqueue_in.lock, "rqueue2");
}
```

```
void
userinit(void)
{
 acquire(&rqueue_in.lock);
 rqueue_in.s=0;
 rqueue_in.e=0;
 release(&rqueue_in.lock);

 acquire(&rqueue_out.lock);
 rqueue_out.s=0;
 rqueue_out.e=0;
 release(&rqueue_out.lock);
```

This process queue will store the processes that couldn't be given extra memory since there were no free pages available. **Struct rq** is a circular queue structure that we'll be using here. To add functionality to the queue, we have also added the **rpop** and **rpush** functions that add and remove processes from the queue respectively. Both of these processes make sure to acquire the lock before making changes to the queue. This lock was initialized in **pinit**. Also we initialised the s (start) and e (end) to zero in **userinit** to make the queue empty. These prototypes were added in defs.h too.
**defs.h:**

```
struct rq;
```

```
int             rpush( struct rq* , struct proc *p);
struct proc*    rpop(struct rq *);
extern struct   rq rqueue_out;
extern struct   rq rqueue_in;
```

Now this **global struct rqueue_out and rqueue_in** will store the processes which are to be swapped out and swap in. Now when kalloc returns zero (not able to allocate pages to a process), it informs allocuvm that requested memory could not be allocated.

**vm.c:**

```
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
 char *mem;
 uint a;

 if(newsz >= KERNBASE)
   return 0;
 if(newsz < oldsz)
   return oldsz;

 a = PGROUNDUP(oldsz);
 for(; a < newsz; a += PGSIZE){
   mem = kalloc();
   if(mem == 0){
     // cprintf("allocuvm out of memory\n");
     deallocuvm(pgdir, newsz, oldsz);

```

```
      //SLEEP
      myproc()->state=SLEEPING;
      acquire(&sleeping_channel_lock);
      myproc()->chan=sleeping_channel;
      sleeping_channel_count++;
      release(&sleeping_channel_lock);

       rpush(&rqueue_out,myproc());
      if(!swap_out_process_exists){
        swap_out_process_exists=1;
        create_kernel_process("swap_out_process",
&swap_out_process_function);
      }

      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
 return newsz;
}
```

Now we first send the process to **SLEEPING** state, where the process sleeps on a special **sleeping_channel** ( this is done so that we can wake up all the processes on the queue together by using a single wakeup later on). The **sleeping_channel_count** is used in **kfree** to check if there is any process at all that is sleeping on the sleeping_channel. We do this while acquiring a lock to make sure there are no ambiguities.

Then we push the current process on the rqueue_out (swap out queue). Then we create the swap_out_process using the create_kernel_process function we created earlier. This will allocate a page for this process if it doesn't already exist. When it is created, the **swap_out_process_exists** flag is set to 1 so that we do not accidentally create multiple swap out processes.

**proc.c:**

```
void swap_out_process_function(){
  int swapped_out;
  acquire(&rqueue_out.lock);
  while(rqueue_out.s!=rqueue_out.e){
    struct proc *p = rpop(&rqueue_out);
    swapped_out = 0;

    pde_t* pd = p->pgdir;
    for(int lvl = 0; lvl < 4 ; lvl ++) {
      if(swapped_out) break;

      for(int i=0;i<NPDENTRIES;i++){
        if(swapped_out) break;

        //skip page table if accessed. chances are high, not every page
table was accessed.
        if( ( pd[i] & ( PTE_A | PTE_D ) ) !=  ( lvl << 5) )
          continue;
        //else
        pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(pd[i]));
        for(int j=0;j<NPTENTRIES;j++){

          //Skip if found
          if(( (pd[i] & ( PTE_A | PTE_D )) != (lvl <<5) ) ||
!(pgtab[j]&PTE_P))
            continue;
          pte_t *pte=(pte_t*)P2V(PTE_ADDR(pgtab[j]));

          //for file name
          int pid=p->pid;
          int virt = ((1<<22)*i)+((1<<12)*j);

          //file name
          char c[50];
          int_to_string(pid,c);
          int x=strlen(c);
          c[x]='_';
          int_to_string(virt,c+x+1);
          safestrcpy(c+strlen(c),".swp",5);
          // file management
          int fd=proc_open(c, O_CREATE | O_RDWR);
          if(fd<0){
            cprintf("error creating or opening file: %s\n", c);
            panic("swap_out_process");
          }

          if(proc_write(fd,(char *)pte, PGSIZE) != PGSIZE){
            cprintf("error writing to file: %s\n", c);
            panic("swap_out_process");
          }
          proc_close(fd);

          kfree((char*)pte);
          // memset(&pgtab[j],0,sizeof(pgtab[j]));
          pgtab[j]^=PTE_P;

          //mark this page as being swapped out.
          pgtab[j]=((pgtab[j])^(0x080));

          swapped_out = 1; break;
        }
      }
    }

  }

  release(&rqueue_out.lock);

  struct proc *p;
  if((p=myproc())==0)
    panic("swap out process");

  swap_out_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

First of all we run the loop till the out queue becomes empty. We pop out the process at the head of the queue and then go over all the levels one by one to implement the LRU policy. We have a two-level page table structure here. Firstly, we iterate through all the entries in the page directory and extract the physical address for each second-level page table. We also make an optimisation to skip the page tables that were already accessed. Now for each secondary page table, we iterate through the page table and look at the **accessed bit (A) and dirty bit (D)** on each of the entries to determine the level. We use the PTE_A and PTE_D bits to determine how last used this particular page was.

Then we get the physical address of the page table entry using the PTE_ADDR macro and then convert it to a virtual address using the P2V macro. Now this page is swapped out and stored to the drive

The name of the file is created using the pid of the process and the virtual address of the page to be eliminated. Then we need to write the contents of the victim page to the file with name **<pid>_<virtual_addr>.swp** . However file system calls cannot be called from the proc.c file . To solve this, we copied the open, write, read, close functions from sysfile.c to proc.c and modified them and renamed them to proc_open, proc_read, proc_write and proc_close.

After writing to the file, we close the file descriptor and then use kfree to free up the page table entry. In the kfree function, there is a modification made to accommodate the sleeping channel. We wakeup all the processes on the **sleeping_channel** we created specially for this purpose. We also unset the **PTE_P** bit to indicate that this page table entry is now vacant. We also update the flag that indicates swapping out of the page table entry. Once we get a page that was swapped, we break out of all the loops since our task is done.

Finally we release the lock of the queue and then do the cleanup as was asked in the question.

## Task 3 : Swapping in mechanism

When the kernel throws a page fault, it must check if the cause is the swapping out mechanism. For that we modify trap.c to deal with page faults separately. First we add a case in the switch statement

```
  case T_PGFLT:
    handlePageFault();
```

This function is implemented in the same file:

```
void handlePageFault(){
  int addr=rcr2();
  struct proc *p=myproc();
  acquire(&swap_in_lock);
  sleep(p,&swap_in_lock);
  pde_t *pde = &(p->pgdir)[PDX(addr)];
  pte_t *pgtab = (pte_t*)P2V(PTE_ADDR(*pde));

  if((pgtab[PTX(addr)])&0x080){
    //This means that the page was swapped out.
    //virtual address for page
    p->addr = addr;
    rpush(&rqueue_in,p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```

We first get the address that threw the page fault using rcr2,  then put the current process to sleep with a new lock called swap_in_lock (initialised in trap.c and with extern in defs.h), use that address to locate the corresponding entry in page table of the process and check if the swapped flag is on. Then the corresponding entry was swapped out and we need to swap back in that page. We push the page in the swap-in queue “rqueue_in”, then create the corresponding kernel process if it hasn’t already been created.

Code of swap_in_process_function’s implementation is pasted below.

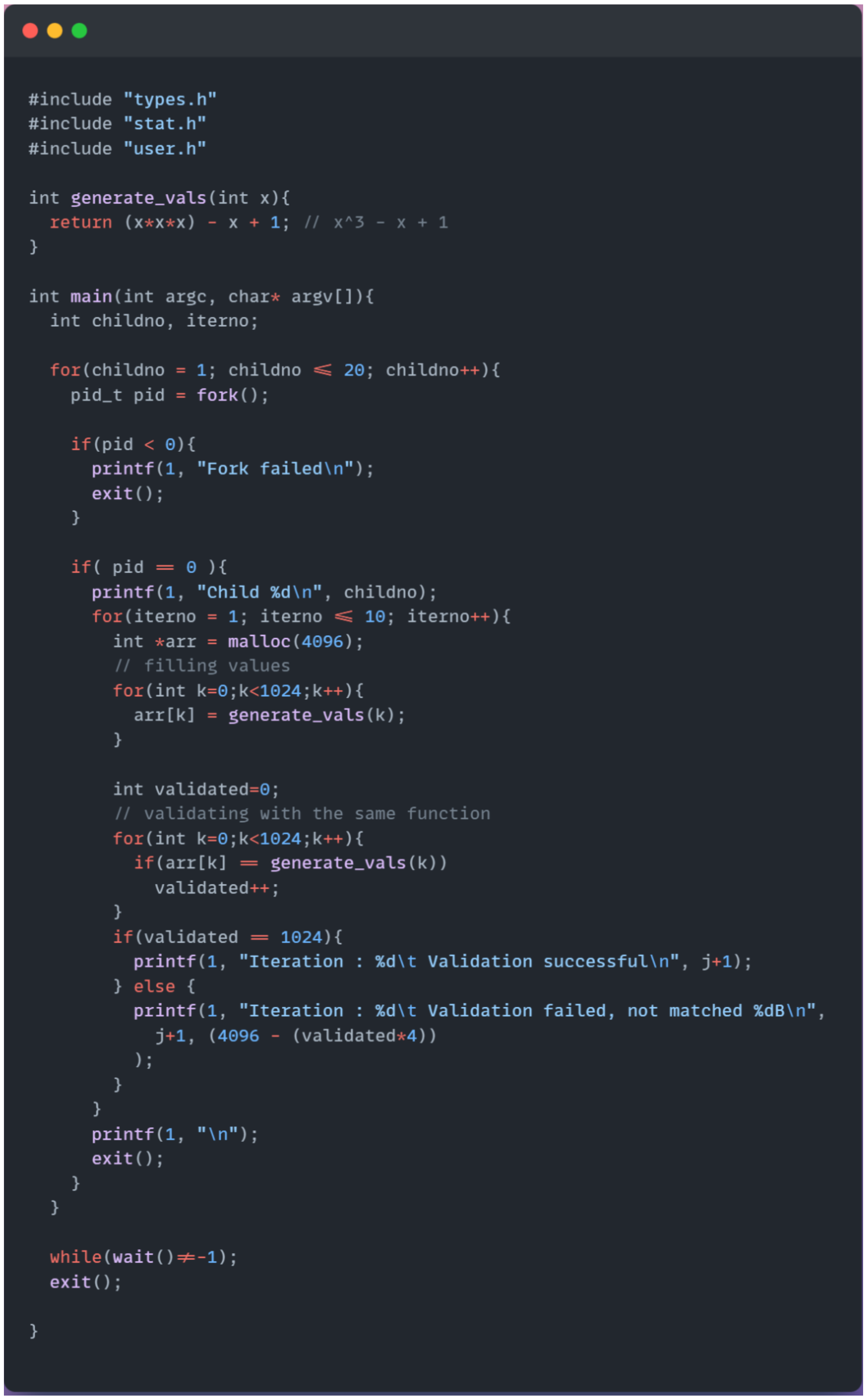
```
void swap_in_process_function(){

  acquire(&rqueue_in.lock);
  while(rqueue_in.s!=rqueue_in.e){
    struct proc *p = rpop(&rqueue_in);

    int pid=p->pid;
    int virt=PTE_ADDR(p->addr);

    char c[50];
    int_to_string(pid,c);
    int x=strlen(c);
    c[x]='_';
    int_to_string(virt,c+x+1);
    safestrcpy(c+strlen(c),".swp",5);

    int fd=proc_open(c,O_RDONLY);
    if(fd<0){
      release(&rqueue_in.lock);
      cprintf("could not find page file in memory: %s\n", c);
      panic("swap_in_process");
    }
    char *mem=kalloc();
    proc_read(fd,PGSIZE,mem);

    if(mappages(p->pgdir, (void *)virt, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
      release(&rqueue_in.lock);
      panic("mappages");
    }
    wakeup(p);
  }

  release(&rqueue_in.lock);
  struct proc *p;
  if((p=myproc())==0)
    panic("swap_in_process");

  swap_in_process_exists=0;
  p->parent = 0;
  p->name[0] = '*';
  p->killed = 0;
  p->state = UNUSED;
  sched();
}
```

Here we run a while loop, and while the queue is not empty, we pop out a process, get its address ( we have modified the proc struct in proc.h to include an extra field for this ) then get the name of the corresponding .swp file (created at the time of swapping-out), open it, then allocate a page using kalloc, write the content of the swp file on the allocated page, then close the file. The file access calls are as described in task 2. We then use mappages to map the allocated page to the corresponding virtual address. Finally we wakeup the process (we had put it to sleep in handlePageFault function).

When the swap-in queue becomes empty, we clean up the swap_in_process_function, just like in task 2. The name has been again set to ‘*’, which would be cleared up in the scheduler.

## Task 4 : sanity test

Here we've written a user level program named memtest, as asked in the assignment. The implementation of this



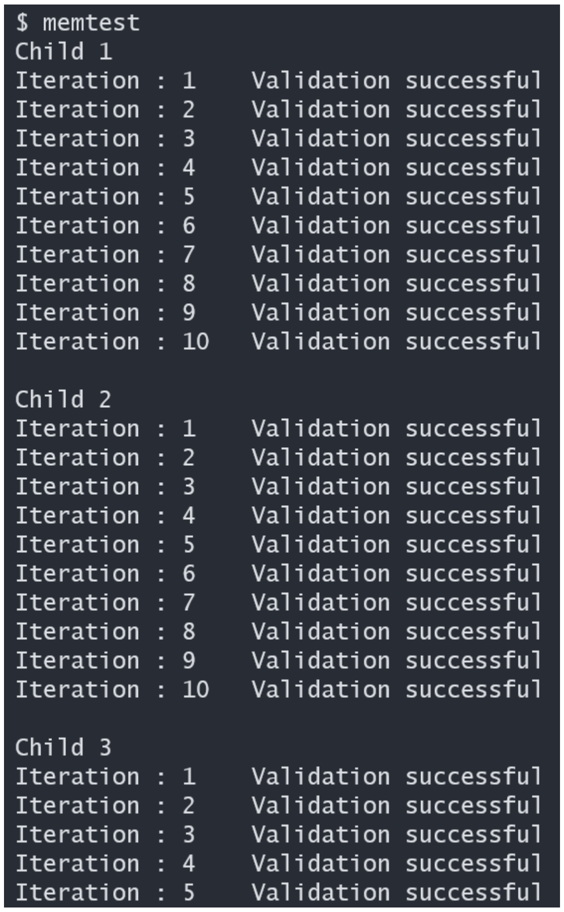
```c
#include "types.h"
#include "stat.h"
#include "user.h"

int generate_vals(int x){
  return (x*x*x) - x + 1; // x^3 - x + 1
}

int main(int argc, char* argv[]){
  int childno, iterno;

  for(childno = 1; childno <= 20; childno++){
    pid_t pid = fork();

    if(pid < 0){
      printf(1, "Fork failed\n");
      exit();
    }

    if( pid == 0 ){
      printf(1, "Child %d\n", childno);
      for(iterno = 1; iterno <= 10; iterno++){
        int *arr = malloc(4096);
        // filling values
        for(int k=0;k<1024;k++){
          arr[k] = generate_vals(k);
        }

        int validated=0;
        // validating with the same function
        for(int k=0;k<1024;k++){
          if(arr[k] == generate_vals(k))
            validated++;
        }
        if(validated == 1024){
          printf(1, "Iteration : %d\t Validation successful\n", j+1);
        } else {
          printf(1, "Iteration : %d\t Validation failed, not matched %dB\n",
            j+1, (4096 - (validated*4))
          );
        }
      }
      printf(1, "\n");
      exit();
    }
  }

  while(wait()!=-1);
  exit();

}
```

The function used to generate the values is the following:

$$f(x) = x^3 - x + 1$$

Here we first create 20 child processes, and for each child process, we run 10 iterations where we allocate 4096 bytes of memory and then first write to the array the values generated by generate_vals function, and then read them back to validate.

We have included it in Makefile under UPROGS and EXTRA to make the process accessible from xv6 terminal. On running it, we see our implemenation passes all tests, cropped screenshot below.



```
$ memtest
Child 1
Iteration : 1    Validation successful
Iteration : 2    Validation successful
Iteration : 3    Validation successful
Iteration : 4    Validation successful
Iteration : 5    Validation successful
Iteration : 6    Validation successful
Iteration : 7    Validation successful
Iteration : 8    Validation successful
Iteration : 9    Validation successful
Iteration : 10   Validation successful

Child 2
Iteration : 1    Validation successful
Iteration : 2    Validation successful
Iteration : 3    Validation successful
Iteration : 4    Validation successful
Iteration : 5    Validation successful
Iteration : 6    Validation successful
Iteration : 7    Validation successful
Iteration : 8    Validation successful
Iteration : 9    Validation successful
Iteration : 10   Validation successful

Child 3
Iteration : 1    Validation successful
Iteration : 2    Validation successful
Iteration : 3    Validation successful
Iteration : 4    Validation successful
Iteration : 5    Validation successful
```

To further test our implementation we decrease the value of **PHYSTOP** to reduce the available physical memory and force the test process to rely on swapping mechanism. We changed PHYSTOP to **0x0400000** and we still see the identical output, with successful validation in all iterations.