CS343 Assignment 3 Report

Group 6

Aditya Trivedi 190101005
Akshat Arun 190101007
Atharva Vijay Varde 190101018
Shashwat Sharma 190123055

Applying changes and executing:

1) Using Patch file:
   - Copy the file patch.txt into “xv6-public” folder
   - `cd xv6_public`
   - `git apply patch.txt`
   - `make clean && make qemu`

2) Using modified files:
   - Copy the files in the “modified files” folder and paste/replace into “xv6-public” folder.
   - `make clean && make qemu`

Part A: Lazy Memory Allocation

In xv6, any process is allocated extra memory whenever it exceeds the allotted memory. This allotment is done by a system call, sbrk(), whenever the process requests it. However Lazy Memory Allocation implies we should allot this extra needed memory whenever accessed, and not when requested.

Whenever system call sbrk() is called, it invokes growproc(). In growproc(), allocuvm function is used to allocate the needed pages and also does the mapping of virtual and physical addresses in the page tables by allocating extra memory.

As we are now allocating lazily, we no longer call growproc() upon a request, and directly return the new address with incremented size. This leads the process to believe we have allocated the extra memory while we actually have not. Now when this extra requested memory is actually accessed, it will lead to page fault, since no such memory is allocated yet. This fault is represented as T_PGFLT in the kernel. We handle this fault by following code :

```
case T_PGFLT:
  ;
  uint va=rcr2();
  va=PGROUNDDOWN(va);

  char *mem;
```

```
   mem = kalloc();

   if(mem == 0){
     cprintf("No page left to be allocated\n");
     panic("trap");
     return;
   }

   memset(mem, 0, PGSIZE);

   int permissions =  (int)(PTE_W|PTE_U);
   uint physAddr = V2P(mem);

   if(mappages(myproc()->pgdir, (char*)va, PGSIZE, physAddr, permissions) <
0){
     cprintf("Page could not be allocated\n");
     panic("trap");
     return;

   }
   break;
```

The rcr2() function returns the virtual address where page fault has occurred. We store this in va variable and then find the address at which this page starts. This is done using inbuilt PGROUNDDOWN macro.

```
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))
```

Using kalloc() we get a pointer to a 4096-Byte of physical memory, i.e. 1 free page. This free page is obtained from kmem which maintains a linked list of free pages.

```
// kalloc.h
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char* kalloc(void)
{
 struct run *r;

 if(kmem.use_lock)
   acquire(&kmem.lock);
 r = kmem.freelist;
 if(r)
   kmem.freelist = r->next;
 if(kmem.use_lock)
   release(&kmem.lock);
 return (char*)r;
}
```

kmem struct :

```
struct {
 struct spinlock lock;
 int use_lock;
 struct run *freelist; // linked list of free pages
} kmem;
```

If the value is 0, it means kalloc could not find any free page. Else, we map this physical memory location to a virtual address calculated using PGROUNDDOWN. This is done by mappages(). It is an inbuilt function in xv6 but to call it in trap.c we must remove the static keyword and declare the prototype in trap.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;
}
```

The function accepts 5 arguments :

1. pgdir : current process' page table
2. va : virtual address of current data
3. size : data size
4. pa : physical address
5. perm : permissions of the page

The physical address is found by V2P macro which stands for virtual to physical (address)

```
// Key addresses for address space layout (see kmap in vm.c for layout)
#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked

#define V2P(a) (((uint) (a)) - KERNBASE)
```

The permissions are Writable and User

```
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
```

Here is how it is called,

```
mappages(myproc()->pgdir, (char*)va, PGSIZE, V2P(mem), PTE_W|PTE_U)
```

We iterate over all the pages of data which is being loaded, the first page is a and the last is last. For each page we call the walkpgdir() function.

```
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)];
}
```

It takes virtual address and the page table and returns the corresponding entry in the table. It uses the PDX and PTX macros to grab the first and last 10 bits of the virtual address. Here two level indexing is used in the page table. The first 10 bits specify the page directory entry and the next 10 bits specify the entry in the page table.

If this page table is present in memory already we make the offset 0 by setting last 12 bits zero. PTE_ADDR macro is used for this.
Else this page is loaded into the directory and it permissions are set.

Lastly, the present-bit (PTE_P) of the entry is checked. If it is set, there is an error, we must remap as the current entry is already mapped to other virtual address. Else we set the page table entry by connecting it to the virtual address, present bit and other permissions (passed as parameter).

Part B :
Few questions answered :

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

```
struct {
 struct spinlock lock;
 int use_lock;
 struct run *freelist; // linked list of free pages
} kmem;
```

kmem struct contains a linked list of free pages. The kernel uses kalloc() function to allocates pages. kalloc uses the kmem linked list to find out unused pages.

What data structures are used to answer this question?

kmem contains a list of pages stored as struct run freelist. It is a linked list.

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

Where do these reside?

Above used data structures reside in kmem struct in kalloc.c

Does xv6 memory mechanism limit the number of user processes?

In param.h the following is defined which limits the number of page table elements, i.e number of processes.

```
#define NPROC        64  // maximum number of processes
```

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

Upon boot up, xv6 runs the initproc function which forks the sh process and forks other processes. Hence lowest number of processes are 1.
As all processes are forked by sh which is initially called by initproc, we cannot have 0 processes.

__**Task 1: Kernel Processes**__

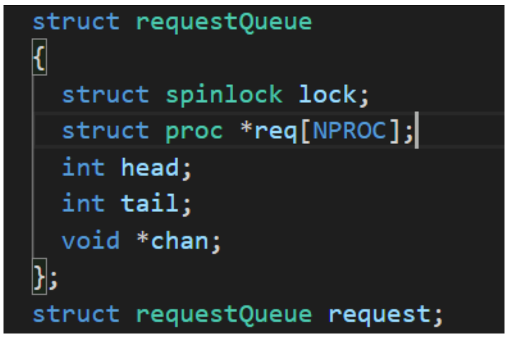
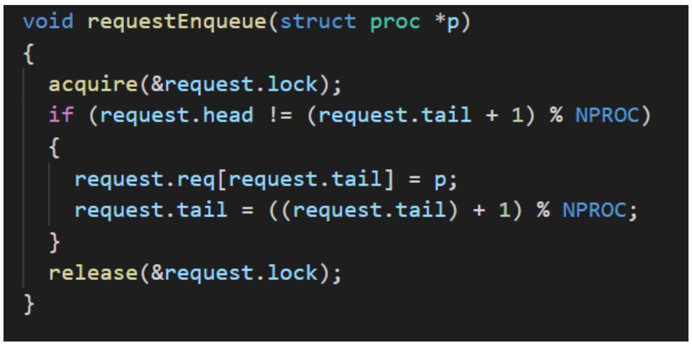
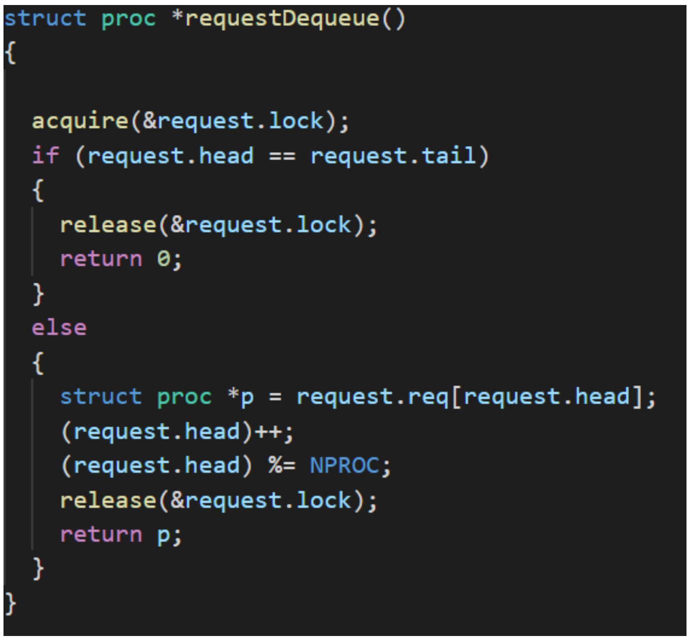
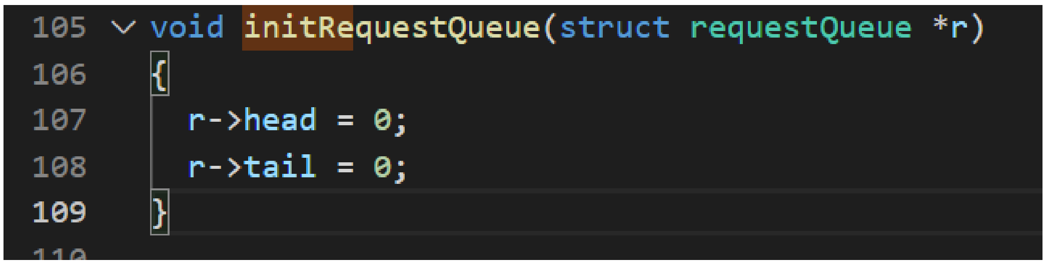
In proc.c we create function create_kernel_process().

```
void create_kernel_process(const char *name, void (*entrypoint)())
{

 struct proc *p;
 p = allocproc();

 if (p == 0)
 {
   panic("Kernel Process not created");
 }

 if ((p->pgdir = setupkvm()) == 0)
 {
   kfree(p->kstack);
   panic("Page table not setup");
 }
 //Since it is a kernel process, trap frame need not be initialised (only for user soace
processes.

 p->sz = PGSIZE;
 p->parent = initproc;
 p->cwd = idup(initproc->cwd);
 //names the process
 safestrcpy(p->name, name, sizeof(p->name));

 acquire(&ptable.lock);
 //entrypoint becomes next instruction to be executed
 p->context->eip = (uint)entrypoint;
 p->state = RUNNABLE;

 release(&ptable.lock);
}
```

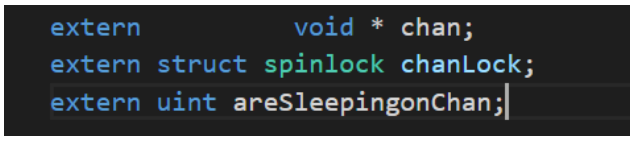
Firstly allocproc is used to return an unused process in the process table. The first unused entry is returned after setting its state to EMBRYO.
Then setupkvm() is used to set up the portion of page table for kernel. Other virtual addresses greater than KERNBASE are mapped to physical addresses.
We need not initialise the trap frame as the process will run in kernel mode. Only user space processes are required to do so.
We set the size as well the parent process - initproc - as it is the parent which invokes other processes. We then set the process name according to the parameter name passed.
Then we set the eip value which stores the address of the next instruction. Here, we set it as the entrypoint passed and set the state as RUNNABLE. This is done after locking the process table.

Task 2: Swapping Out Mechanism

Before starting to analyze where the memory shortage for a particular process will be detected, the question asks for a request queue to support multiple processes having this same problem.
Therefore a queue structure called **requestQueue** must be created to store all the processes which have been denied memory. It is a circular queue. An instance of the requestQueue called **request** is also created.
Enqueue and dequeue functions are also created.


```
struct requestQueue
{
  struct spinlock lock;
  struct proc *req[NPROC];
  int head;
  int tail;
  void *chan;
};
struct requestQueue request;
```

```
void requestEnqueue(struct proc *p)
{
  acquire(&request.lock);
  if (request.head != (request.tail + 1) % NPROC)
  {
    request.req[request.tail] = p;
    request.tail = ((request.tail) + 1) % NPROC;
  }
  release(&request.lock);
}
```

```
struct proc *requestDequeue()
{

  acquire(&request.lock);
  if (request.head == request.tail)
  {
    release(&request.lock);
    return 0;
  }
  else
  {
    struct proc *p = request.req[request.head];
    (request.head)++;
    (request.head) %= NPROC;
    release(&request.lock);
    return p;
  }
}
```

```
void initRequestQueue(struct requestQueue *r)
{
  r->head = 0;
  r->tail = 0;
}
```
The requestQueue is initialised by the initRequestQueue function which is called by initproc.

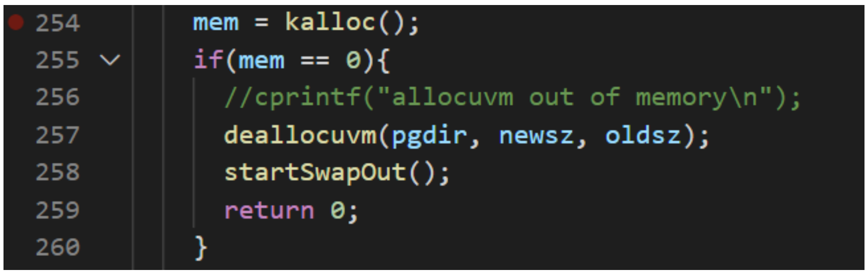
Also, the question asks us to create a mechanism to wake up all processes once a page is freed.
Along with the queue, a sleeping channel **chan** is also declared for all the suspended processes to sleep on so that they can be woken up simultaneously.


```
extern          void * chan;
extern struct spinlock chanLock;
extern uint areSleepingonChan;
```

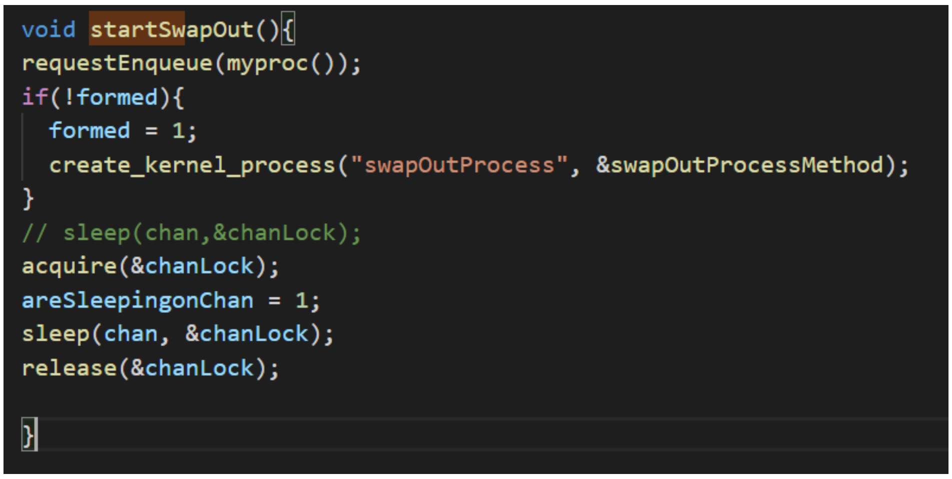
All functions and struct declarations are declared globally in defs.h.

Now we will start to deal with the actual freeing of memory.

Since memory allocation is not lazy, a process memory lack will only be detected whenever the physical memory allocator **kalloc** returns 0 denoting failure.
Thus, a procedure must be invoked if this failure occurs, in **allocuvm()** function in vm.c.


```
mem = kalloc();
if(mem == 0){
  //cprintf("allocuvm out of memory\n");
  deallocuvm(pgdir, newsz, oldsz);
  startSwapOut();
  return 0;
}
```
allocuvm() in vm.c

The **startSwapOut** routine will acknowledge that the current process has experienced a lack of physical memory. Thus, it will change the state of the current process to SLEEPING, place it on our request queue for processes wanting more memory, and invoke the swapOutProcess function.
Note that the current process will sleep on the sleeping channel that we have already created.

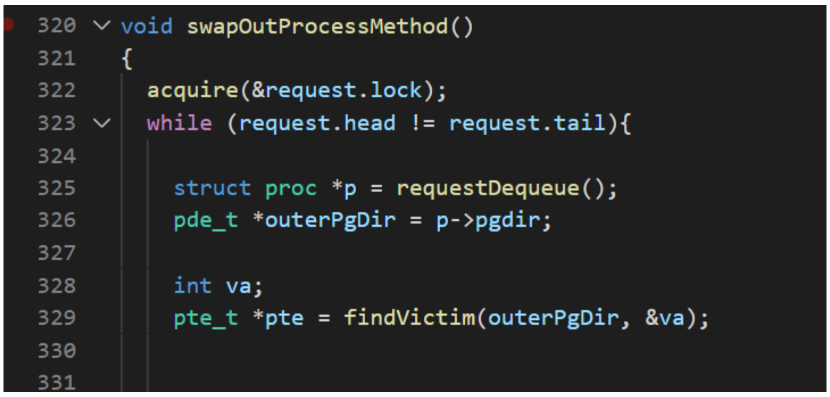

```
void startSwapOut(){
requestEnqueue(myproc());
if(!formed){
  formed = 1;
  create_kernel_process("swapOutProcess", &swapOutProcessMethod);
}
// sleep(chan,&chanLock);
acquire(&chanLock);
areSleepingonChan = 1;
sleep(chan, &chanLock);
release(&chanLock);

}
```
StartSwapOut in vm.c

Now, we will discuss what this swapOutProcess function will do. This has been defined in proc.c and declared globally in defs.h .
Note that swapOutProcess is a kernel process, created using the method we created in Task1.

The SwapOut process is long and has various subroutines. Briefly, what it does is that for each process in the request queue:

- It finds a suitable page from its page table to evict. Here, we have used the second chance LRU algorithm (will be discussed in more detail).
- Next, it writes this page back to the file system, along with marking the page table entry for it (to indicate that a dirty version of this page resides in the disk).

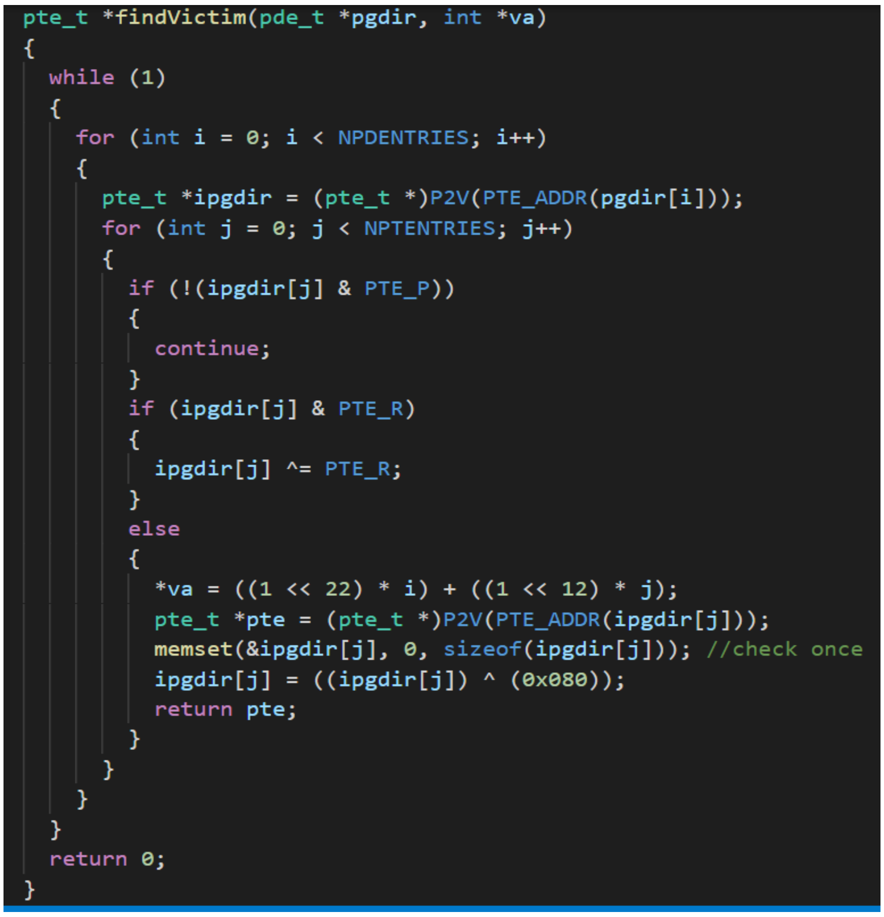
- Now, a page slot is free in this particular processes' page table. Therefore, the SwapOutProcess calls the kfree() function from kalloc.c to free the page, and wake up all processes sleeping in our Sleeping Channel.



```c
void swapOutProcessMethod()
{
  acquire(&request.lock);
  while (request.head != request.tail){

    struct proc *p = requestDequeue();
    pde_t *outerPgDir = p->pgdir;

    int va;
    pte_t *pte = findVictim(outerPgDir, &va);

```

Starting part of swapOutProcessMethod in proc.c.

The for loop is run throughout the queue and the findVictim subroutine is called.

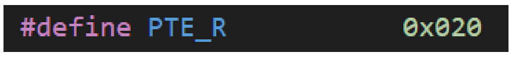


```c
pte_t *findVictim(pde_t *pgdir, int *va)
{
  while (1)
  {
    for (int i = 0; i < NPDENTRIES; i++)
    {
      pte_t *ipgdir = (pte_t *)P2V(PTE_ADDR(pgdir[i]));
      for (int j = 0; j < NPTENTRIES; j++)
      {
        if (!(ipgdir[j] & PTE_P))
        {
          continue;
        }
        if (ipgdir[j] & PTE_R)
        {
          ipgdir[j] ^= PTE_R;
        }
        else
        {
          *va = ((1 << 22) * i) + ((1 << 12) * j);
          pte_t *pte = (pte_t *)P2V(PTE_ADDR(ipgdir[j]));
          memset(&ipgdir[j], 0, sizeof(ipgdir[j])); //check once
          ipgdir[j] = ((ipgdir[j]) ^ (0x080));
          return pte;
        }
      }
    }
  }
  return 0;
}
```

The **findVictim function** finds a victim to be ejected from the page table of the process. Since xv6 uses a hierarchical page table, we iterate through every page in the final page table using two for loops.

Now, to select the page to be ejected out, we use the **Second Chance LRU Approximation Algorithm** :

For the reference bit used by the algorithm, we use the Access Bit (the 6th bit of the page table entry).
This bit is set to 1 whenever a page is referenced by the OS.



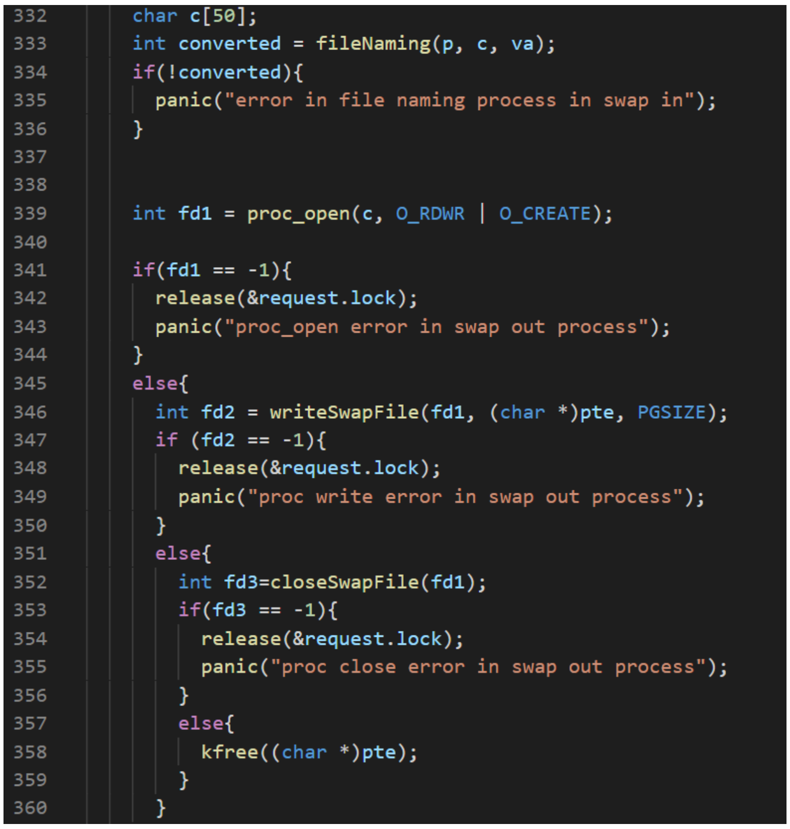
```c
#define PTE_R           0x020
```

The way the second chance algorithm will work is, whenever it encounters a page which has a reference bit set to 1, it gives it a **second chance** and sets it to 0.
However, if it encounters a page with the reference bit as 0, it selects this page as the victim page.

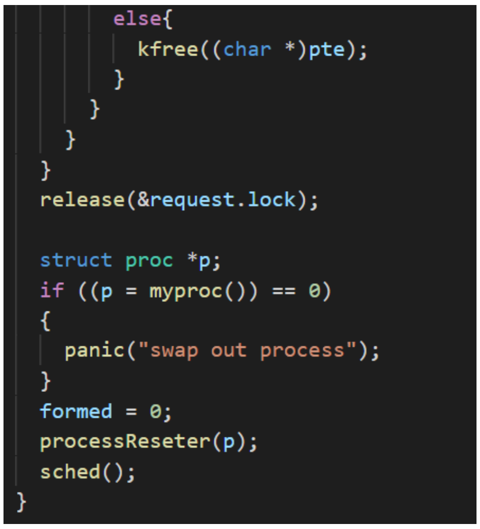
Also, we are using the **0x080 bit of the page table to denote that this page has been fetched by the process but has been ejected**, and currently resides in the file system. This will be used in SwapInProcess.
We now come back to SwapOutProcess in proc.c with our victim page identified.
It is returned in the form of a pointer to that page : **pte.**

We must now write the contents of the file to the disk. For that, we first obtain the name of the file **converted** using the **fileNaming()** Function. Next, we create a file with that name and write the page into that file, using **proc_open()**, **writeSwapFile()** functions.
The **ProcessReseter** function resets all parameters for the SwapOutProcess before passing control to the scheduler.

These functions are all present in proc.c (screenshots are not attached).

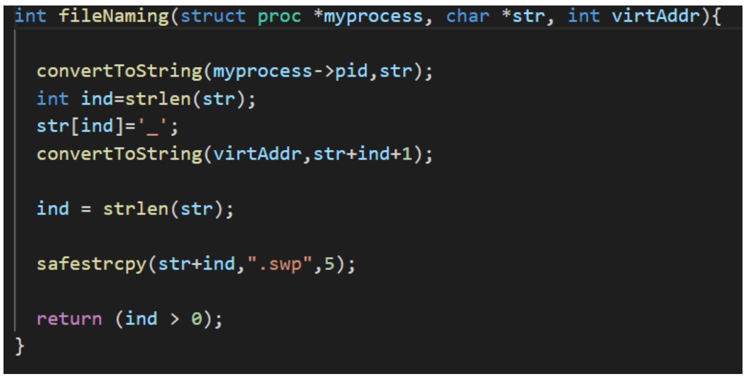


```c
    char c[50];
    int converted = fileNaming(p, c, va);
    if(!converted){
      panic("error in file naming process in swap in");
    }


    int fd1 = proc_open(c, O_RDWR | O_CREATE);

    if(fd1 == -1){
      release(&request.lock);
      panic("proc_open error in swap out process");
    }
    else{
      int fd2 = writeSwapFile(fd1, (char *)pte, PGSIZE);
      if (fd2 == -1){
        release(&request.lock);
        panic("proc write error in swap out process");
      }
      else{
        int fd3=closeSwapFile(fd1);
        if(fd3 == -1){
          release(&request.lock);
          panic("proc close error in swap out process");
        }
        else{
          kfree((char *)pte);
        }
      }
```



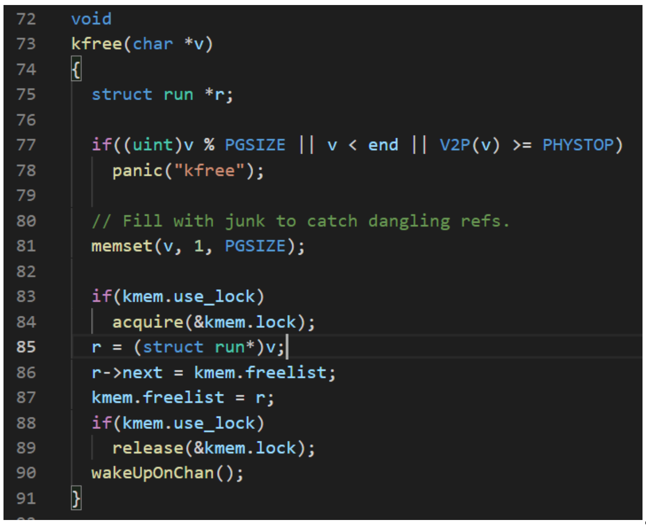
```c
          else{
            kfree((char *)pte);
          }
        }
      }
    }
    release(&request.lock);

    struct proc *p;
    if ((p = myproc()) == 0)
    {
      panic("swap out process");
    }
    formed = 0;
    processReseter(p);
    sched();
}
```

<u>SwapOutProcess - continued.</u>

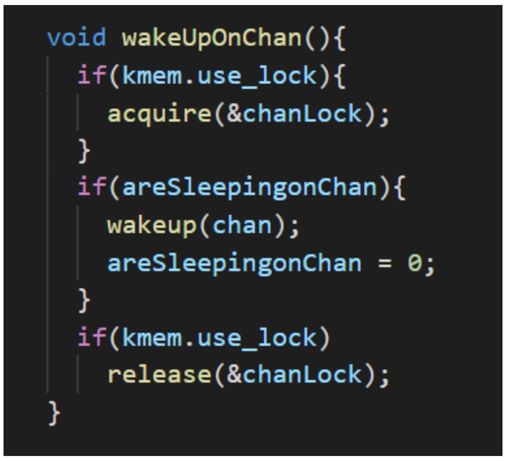


```c
int fileNaming(struct proc *myprocess, char *str, int virtAddr){

  convertToString(myprocess->pid,str);
  int ind=strlen(str);
  str[ind]='_';
  convertToString(virtAddr,str+ind+1);

  ind = strlen(str);

  safestrcpy(str+ind,".swp",5);

  return (ind > 0);
}
```

<u>FileNaming Function :</u> It forms the name of the file in which the dirty page will be written using the **pid** of the process, and the **virtual address** of the ejected page.
Finally, kfree(pte) is called, which actually frees the page .



```c
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
  wakeUpOnChan();
}
```



```c
void wakeUpOnChan(){
  if(kmem.use_lock){
    acquire(&chanLock);
  }
  if(areSleepingonChan){
    wakeup(chan);
    areSleepingonChan = 0;
  }
  if(kmem.use_lock)
    release(&chanLock);
}
```

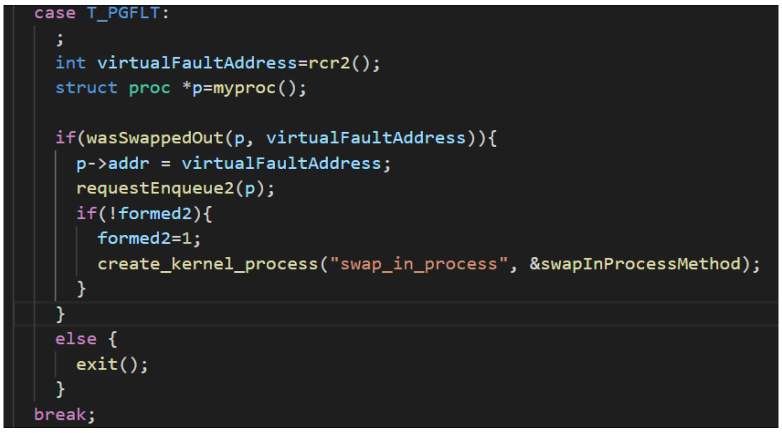
kfree() in kalloc.c **wakeUpOnChan()**

We have added the wakeUpOnChan() function in kfree(). Once the required page has been freed, this function **wakes up all processes** which were sleeping on **chan** (as asked in the question). All these processes were in our request queue.
Note that the processes for whom pages have not been freed yet will simply call kalloc, and sleep on chan again.
This satisfies all requirements of question 2.

Task 3: Swapping In Mechanism

Again for Task3, we must create an instance of our RequestQueue structure so that multiple processes can request for a swap-in.

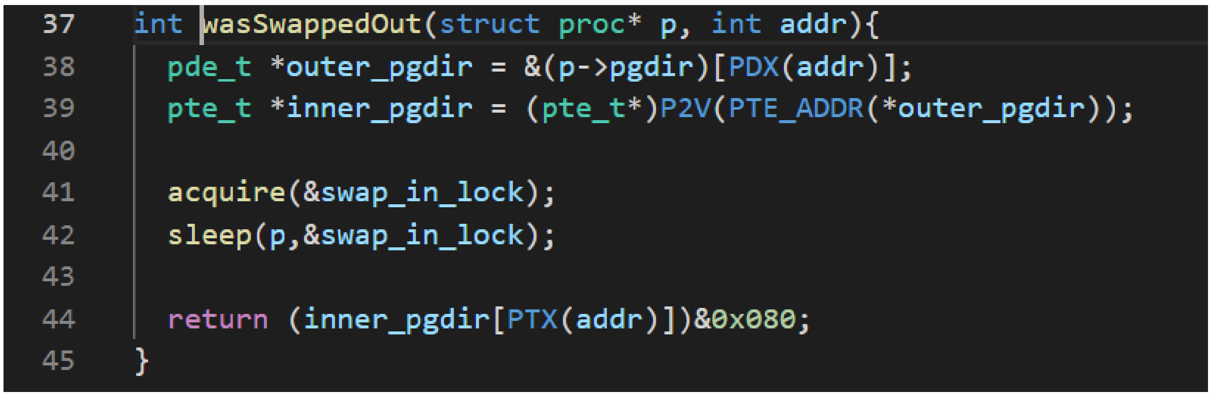
Now, the swapping in mechanism involves swapping in a dirty page from the disk whenever a request is made for it.
Here, the sbrk() and growproc() system calls will not be invoked since the process has already fetched the page once from memory. Thus the OS will think that this page is already in the page table.
However, when it searches through the page table, it will find the accessed bit PTE_A for that page set to 0. Thus, it will generate a page fault trap.
Thus, **trap.c** must be modified to deal with the case of T_PGFLT.



```c
case T_PGFLT:
  ;
  int virtualFaultAddress=rcr2();
  struct proc *p=myproc();

  if(wasSwappedOut(p, virtualFaultAddress)){
    p->addr = virtualFaultAddress;
    requestEnqueue2(p);
    if(!formed2){
      formed2=1;
      create_kernel_process("swap_in_process", &swapInProcessMethod);
    }
  }
  else {
    exit();
  }
break;
```

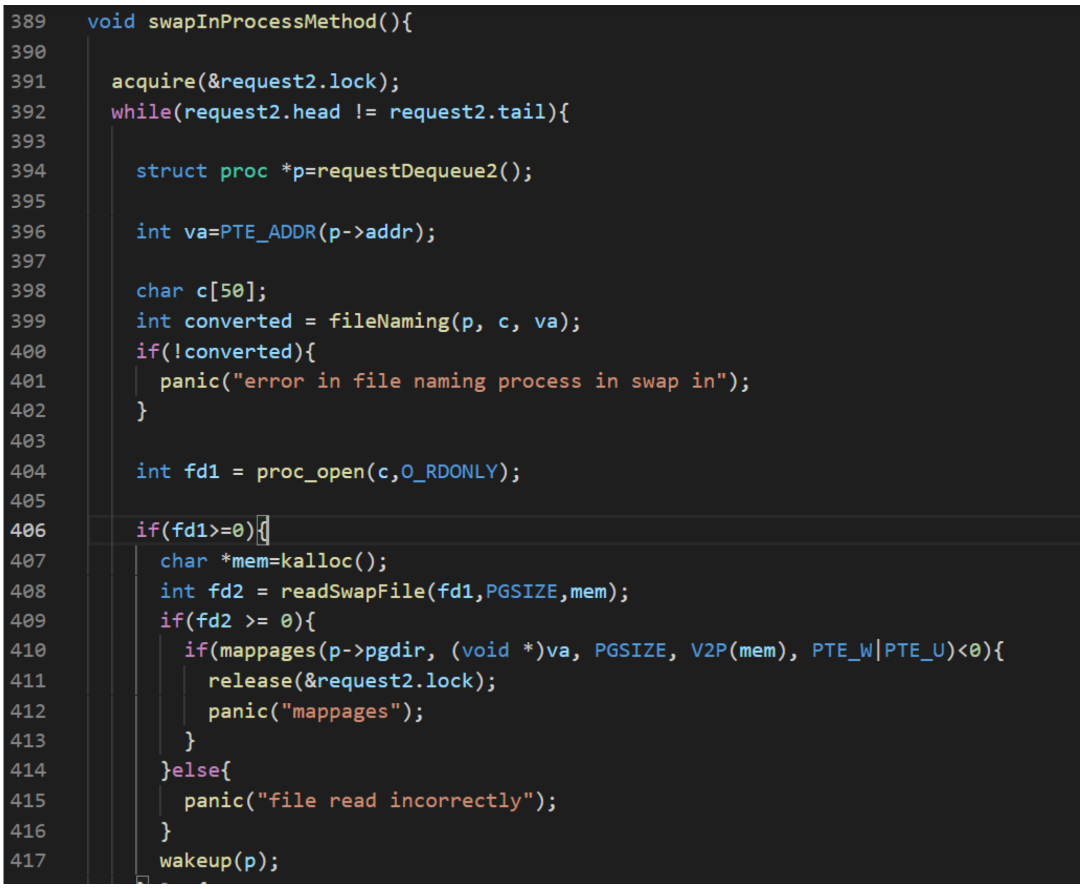
If the page at the current virtual address has been swapped out, we add it to the request queue of processes waiting for swap-in.
Then we create the SwapIn Process if it was not already created.

wasSwappedOut() checks if the page under consideration is present on the file or not. We first obtain a pointer to the inner page table in the hierarchical paging table. Then we check if the 0x080 bit of our page's entry is 1 or not.
Remember that we set this entry to 1 in swapOutProcess!

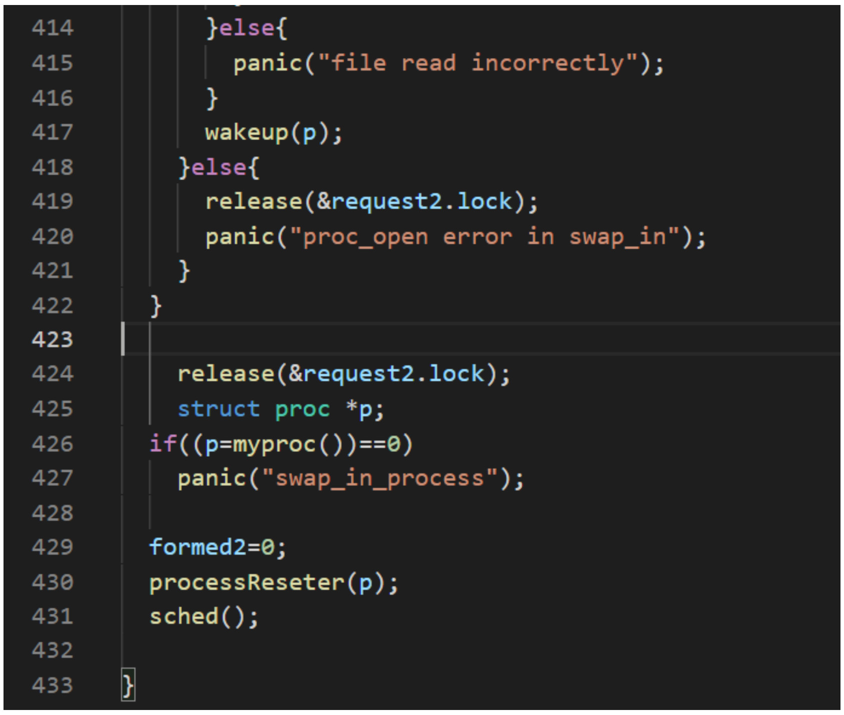


```c
int wasSwappedOut(struct proc* p, int addr){
  pde_t *outer_pgdir = &(p->pgdir)[PDX(addr)];
  pte_t *inner_pgdir = (pte_t*)P2V(PTE_ADDR(*outer_pgdir));

  acquire(&swap_in_lock);
  sleep(p,&swap_in_lock);

  return (inner_pgdir[PTX(addr)])&0x080;
}
```

Now, we discuss the SwapInProcessMethod. Its job is to bring back the page which was swapped out to the disk. It runs a loop for all waiting processes, and uses all the functions created previously to open and read from the required file.
Further, it calls kalloc which allocates a free page frame to load our swapped out page into (line 407).
After calling kalloc, we use readSwapFile to read file and write into mem. Finally we call **mappages** to create a page table entry for this new page in our process.



```c
389  void swapInProcessMethod(){
390
391    acquire(&request2.lock);
392    while(request2.head != request2.tail){
393
394      struct proc *p=requestDequeue2();
395
396      int va=PTE_ADDR(p->addr);
397
398      char c[50];
399      int converted = fileNaming(p, c, va);
400      if(!converted){
401        panic("error in file naming process in swap in");
402      }
403
404      int fd1 = proc_open(c,O_RDONLY);
405
406      if(fd1>=0){
407        char *mem=kalloc();
408        int fd2 = readSwapFile(fd1,PGSIZE,mem);
409        if(fd2 >= 0){
410          if(mappages(p->pgdir, (void *)va, PGSIZE, V2P(mem), PTE_W|PTE_U)<0){
411            release(&request2.lock);
412            panic("mappages");
413          }
414        }else{
415          panic("file read incorrectly");
416        }
417        wakeup(p);
```



```
      }else{
        panic("file read incorrectly");
      }
      wakeup(p);
    }else{
      release(&request2.lock);
      panic("proc_open error in swap_in");
    }
  }

    release(&request2.lock);
    struct proc *p;
  if((p=myproc())==0)
    panic("swap_in_process");

  formed2=0;
  processReseter(p);
  sched();

}
```

Finally, we call wakeup() for the process whose page was swapped in.

Unlike in swapping out, where we woke up all processes when a page became free, here we wake up only the process whose page was swapped in.
Thus, no sleeping channel is required here.

Again, after swapping in is done for all processes, the kernel process resets and gives control to scheduler.

This concludes Task 3.

Task 4: Sanity Tests

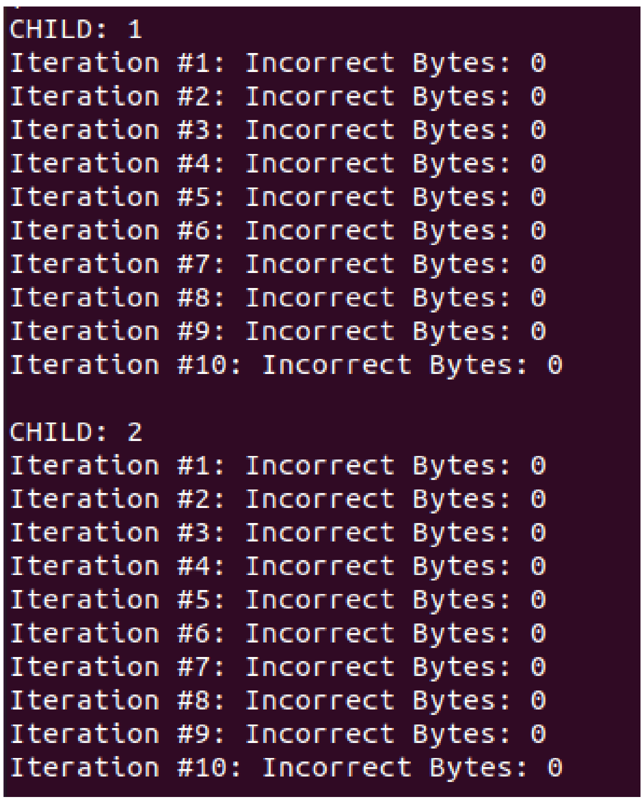
memtest.c is the user program which will test the above implemented features in xv6. We will also modify the Makefile to add memtest in UPROGS (User Programs).

The code is as follows :

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

int numGenerator(int num){
   int smallNum = num%10;
   return (5*smallNum + 7*smallNum*smallNum + 18*smallNum*smallNum*smallNum +
55*smallNum*smallNum*smallNum*smallNum);
   /*
   5, 7, 18 and 55 are our Roll Nums :)
   */
}

void validator(int ind, int num, int* correct){
   if (numGenerator(ind) == num) *correct+=4;
   return;
}
```

```
int main(int argc, char *argv[])
{
   for (int i=1; i<=20; i++)
       if (!fork()){
           int *ptr[10];
           printf(1, "CHILD: %d\n", i);
           for (int j=0; j<10; j++) {
               ptr[j] = (int *)malloc(4096);
               for (int k=0; k<1024; k++)
                   ptr[j][k] = numGenerator(k);
           }

           for (int j=0; j<10; j++){
               int correctBytes=0;
               for (int k=0; k<1024; k++){
                   validator(k,ptr[j][k],&correctBytes);
               }
               printf(1,"Iteration #%d: Incorrect Bytes:
%d\n",j+1,4096-correctBytes);
           }
           printf(1,"\n");
           exit();
       }
//children waiting and parent exit() beyond this
```

<u>Output (shown for the first 2 children, same for all 20):</u>



```
CHILD: 1
Iteration #1: Incorrect Bytes: 0
Iteration #2: Incorrect Bytes: 0
Iteration #3: Incorrect Bytes: 0
Iteration #4: Incorrect Bytes: 0
Iteration #5: Incorrect Bytes: 0
Iteration #6: Incorrect Bytes: 0
Iteration #7: Incorrect Bytes: 0
Iteration #8: Incorrect Bytes: 0
Iteration #9: Incorrect Bytes: 0
Iteration #10: Incorrect Bytes: 0

CHILD: 2
Iteration #1: Incorrect Bytes: 0
Iteration #2: Incorrect Bytes: 0
Iteration #3: Incorrect Bytes: 0
Iteration #4: Incorrect Bytes: 0
Iteration #5: Incorrect Bytes: 0
Iteration #6: Incorrect Bytes: 0
Iteration #7: Incorrect Bytes: 0
Iteration #8: Incorrect Bytes: 0
Iteration #9: Incorrect Bytes: 0
Iteration #10: Incorrect Bytes: 0
```

Explanation :

We test our code as follows:

1. The parent process memtest forks 20 identical child processes, where each of them run a loop of iterations. In each iteration, The child calls malloc(4096) to request additional memory and uses it as an int array. We iterate over this array and fill it with an integer calculated as a function of the index i using `numGenerator(i).`
2. After all the iterations are completed, meaning 10 arrays allocated, we then run a validation loop. The validation loop iterates again over the array, and matches the array value stored at index i with the value assigned earlier (numGenerator(i)).
3. We maintain a counter variable correctBytes for each array to track the number of bytes correctly matched. Finally, we report the number of incorrectly matched bytes (4096 - correctBytes).

We also ran the code against lower values of PHYSTOP. In particular, we also set it to 0x0400000 (meaning 4MB of memory), which is the least needed by xv6 to execute kinit1 (as seen in main.c and xv6 documentation).

In all cases, the output obtained is identical as shown above. The number of mismatched bytes are 0 in all iterations for all children, which means our implementation works correctly.