Assignment 3

Part A: Lazy Memory Allocation

- When a process needs extra memory, it indicates this requirement using the **sbrk system call** in the xv6 OS.
- **sbrk** uses **growproc** (defined in proc.c) to allocate extra memory by mapping virtual addresses to physical addresses in the page table.
- The assignment's goal is to implement **Lazy Memory Allocation**, where memory is allocated when accessed, not when requested.
- To achieve this, the **growproc** call inside sbrk is commented out, giving the process the illusion of allocated memory.
- Upon accessing the "allocated" memory, a **page fault** occurs, generating a **T_PGFLT trap** to the kernel.
- The trap is handled in **trap.c** by calling **handlePageFault()**, which uses **rcr2()** to get the virtual address of the fault.
- The virtual address is rounded to the nearest page boundary and then a free physical page is allocated using **kalloc** from a list of free pages.
- The physical page is mapped to the virtual address using **mappages**, which is declared by removing its static keyword in trap.c.
- **mappages** takes the page table, virtual address, physical memory address (converted using **V2P**), and permission bits.

- **walkpgdir()** is used to find the page table entry for a given virtual address, operating with a two-level page table structure.
- **mappages** checks if the page table entry is already mapped (by checking the PRESENT bit) and maps the physical page if not already mapped.
- If an error occurs (such as a remap), an error is raised by **mappages**.

```
case T_PGFLT:
  if(handlePageFault()<0){
    cprintf("Could not allocate page. Sorry.\n");
    panic("trap");
  }
break;
```

(trap.c)

```
int handlePageFault(){
  int addr=rcr2();
  int rounded_addr = PGROUNDDOWN(addr);
  char *mem=kalloc();
  if(mem!=0){
    memset(mem, 0, PGSIZE);
    if(mappages(myproc()->pgdir, (char*)rounded_addr, PGSIZE, V2P(mem),
    PTE_W|PTE_U)<0)
      return -1;
    return 0;
  } else
    return -1;
}
```

(trap.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;
}
```

(vm.c)

```c
// create any required page table pages.
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)];
}
```
(vm.c)

PART B : Question and answer

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

xv6 maintains a linked list of free pages in kalloc.c called kmem. Initially, the list is empty, so xv6 calls kinit1 through main() which adds 4MB of free pages to the list.

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

A linked list named freelist as shown in the above image. Every node of the linked list is a structure defined in kalloc.c namely struct run (pages are typecast to (struct run *) when inserting into freelist in kfree(char *v)).

Q3: Where do these reside?

This linked list is declared inside kalloc.c inside a structure kmem. Every node is of the type struct run which is also defined inside kalloc.c.

Q4: Does xv6 memory mechanism limit the number of user processes?

Due to a limit on the size of ptable (a max. of NPROC elements which is set to 64 by default), the number of user processes are limited in xv6. NPROC is defined in param.h.

Q5: If so, what is the lowest number of processes xv6 can 'have' at the same time (assuming the kernel requires no memory whatsoever)?

The lowest number of processes xv6 can "have" at the same time, assuming the kernel requires no memory at all (a theoretical assumption), would still be **2 processes**. Here's why:

1. Process 0 (the "scheduler" process):

- xv6 always starts with an initial process called **process 0** or the **scheduler process**. This process is responsible for managing CPU scheduling and process switching. It is created during the system initialization in main().
- Even though the scheduler process doesn't run user code, it is necessary for xv6 to function because it coordinates all other processes.

2. Process 1 (the "init" process):

- After initializing the kernel and creating the scheduler process, xv6 creates a special process called **init** (process 1). The **init process** is the first user-level process in xv6 and is responsible for starting the shell and other user-level programs.
- This process also serves as a parent for any orphaned processes (i.e., processes whose parent has exited).

Part B Assignment.

Task 1:

- **The create_kernel_process() function:** This function is where the kernel process is created.
- **Kernel mode:** The kernel process will always remain in kernel mode, meaning it has direct access to the system's hardware and resources.
- **Trapframe:** The kernel process doesn't need to initialize its trapframe because it won't be switching to user mode. Trapframes store user-space register values, which are only relevant for user processes.
- **User space and page table:** Similarly, the kernel process doesn't need a user space or a user section of its page table since it operates exclusively in kernel mode.
- **EIP register:** The EIP (instruction pointer) register is set to the entry_point address. This is the starting address of the code that the kernel process will execute.
- **Allocproc and setupkvm:** These functions are used to allocate a process slot in the process table (ptable) and set up the kernel part of the process's page table, respectively. The kernel page table maps virtual addresses above KERNBASE (the base address of the kernel) to physical addresses between 0 and PHYSTOP (the end of physical memory).

This code creates the kernel process, sets up its initial context, and configures its memory management. The kernel process will then start executing the code at the specified entry_point.

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

  struct proc *p = allocproc();

  if(p == 0)
    panic("create_kernel_process failed");

  //Setting up kernel page table using setupkvm
  if((p->pgdir = setupkvm()) == 0)
    panic("setupkvm failed");

  //This is a kernel process. Trap frame stores user space registers. We don't need to initialise tf.
  //Also, since this doesn't need to have a userspace, we don't need to assign a size to this process.

  //eip stores address of 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);

}
```
(proc.c)

Task 2:

This task has various parts. First, we need a process queue that keeps track of the processes that were refused additional memory since there were no free pages available. We created a circular queue struct called rg. And the specific queue that holds processes with swap out requests is rqueue. We have also created the functions corresponding to rg, namely rpush() and rpop(). The queue needs to be accessed with a lock that we have initialized in pinit. We have also initialized the initial values of s and e to zero in userinit. Since the queue and the functions relating to it are needed in other files too, we added prototypes in defs.h too.

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

//circular request queue for swapping out requests.
struct rq rqueue;

struct proc* rpop(){

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

  return p;
}
```

(proc.c)

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

(proc.c)

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

  acquire(&rqueue2.lock);
  rqueue2.s=0;
  rqueue2.e=0;
  release(&rqueue2.lock);

  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();

  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  // this assignment to p->state lets other cores
  // run this process. the acquire forces the above
  // writes to be visible, and the lock is also needed
  // because the assignment might not be atomic.
  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);

}
```

(proc.c)

```c
int rpush(struct proc *p){

  acquire(&rqueue.lock);
  if((rqueue.e+1)%NPROC==rqueue.s){
    release(&rqueue.lock);
    return 0;
  }
  rqueue.queue[rqueue.e]=p;
  rqueue.e++;
  (rqueue.e)%=NPROC;
  release(&rqueue.lock);

  return 1;
}
```

(proc.c)

```c
extern struct rq rqueue;
extern struct rq rqueue2;
int rpush(struct proc *p);
struct proc* rpop();
struct proc* rpop2();
int rpush2(struct proc* p);
```

(defs.h)

Now, whenever kalloc is not able to allocate pages to a process, it returns zero. This notifies allocuvm that the requested memory wasn't allocated (mem=0). Here, we first need to change the process state to sleeping.

(*Note: The process sleeps on a special sleeping channel called sleeping_channel that is secured by a lock called sleeping_channel_lock. sleeping_channel_count is used for corner cases when the system boots)

Then, we need to add the current process to the swap out request queue, rqueue:

```c
struct spinlock sleeping_channel_lock;
int sleeping_channel_count=0;
char * sleeping_channel;
```

(vm.c)

```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(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;
}
```

(vm.c)

*Note: create_kernel_process here creates a swapping out kernel process to allocate a page for this process if it doesn't already exist. When the swap out process ends, the swap_out_process_exists (declared as extern in defs.h and initialized in proc.c to 0) variable is set to 0. When it is created, it is set to 1 (as seen above). This is done so multiple swap out processes are not created. swap_out_process is explained later.

Next, we create a mechanism by which whenever free pages are available, all the processes sleeping on sleeping_channel are woken up. We edit kfree in kalloc.c in the following way:

Basically, all processes that were preempted due to lack of availability of pages were sent sleeping on the sleeping channel. We wake all processes currently sleeping on sleeping_channel by calling the wakeup() system call.

```
void
kfree(char *v)
{

  struct run *r;
  // struct proc *p=myproc();

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

  // Fill with junk to catch dangling refs.
  // memset(v, 1, PGSIZE);
  for(int i=0;i<PGSIZE;i++){
    v[i]=1;
  }

  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);

  //Wake up processes sleeping on sleeping channel.
  if(kmem.use_lock)
    acquire(&sleeping_channel_lock);
  if(sleeping_channel_count){
    wakeup(sleeping_channel);
    sleeping_channel_count=0;
  }
  if(kmem.use_lock)
    release(&sleeping_channel_lock);

}
```

(kalloc.c)

Now, I will explain the swapping out process. The entry point for the swapping out process in swap_out_process_function. Since the function is very long, I have attached two screenshots:

```c
void swap_out_process_function(){

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

    pde_t* pd = p->pgdir;
    for(int i=0;i<NPDENTRIES;i++){

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

        //Skip if found
        if((pgtab[j]&PTE_A) || !(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]));

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

        break;
      }
    }

  }

  release(&rqueue.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();
}
```

(proc.c)

The process runs a loop until the swap out requests queue (rqueue1) is non-empty. When the queue is empty, a set of instructions are executed for the termination of swap_out_process. The loop starts by popping the first process from rqueue and uses the LRU policy to determine a victim page in its page table. We iterate through each entry in the process' page table (pgdir) and extracts the physical address for each secondary page table. For each secondary page table, we iterate through the page table and look at the accessed bit (A) on each of the entries (The accessed bit is the sixth bit from the right. We check if it is set by checking the bitwise & of the entry and PTE_A (which we defined as 32 in mmu.c)).

Important note regarding the Accessed flag: Whenever the process is being context switched into by the scheduler, all accessed bits are unset. Since we are doing this, the accessed bit seen by swap_out_process function will indicate whether the entry was accessed in the last iteration of the process:

```
for(int i=0;i<NPDENTRIES;i++){
  //If PDE was accessed

  if(((p->pgdir)[i])&PTE_P && ((p->pgdir)[i])&PTE_A){

    pte_t* pgtab = (pte_t*)P2V(PTE_ADDR((p->pgdir)[i]));

    for(int j=0;j<NPTENTRIES;j++){
      if(pgtab[j]&PTE_A){
        pgtab[j]^=PTE_A;
      }
    }

    ((p->pgdir)[i])^=PTE_A;
  }
}

// Switch to chosen process.  It is the process's job
// to release ptable.lock and then reacquire it
// before jumping back to us.
c->proc = p;
switchuvm(p);
p->state = RUNNING;

swtch(&(c->scheduler), p->context);
switchkvm();
```

(proc.c)

This code resides in the scheduler and it basically unsets every accessed bit in the process' page table and its secondary page tables.

Now, back to swap_out_process function. As soon as the function finds a secondary page table entry with the accessed bit unset, it chooses this entry's physical page number (using macros mentioned in part A report) as the victim page. This page is then swapped out and stored to drive.

We use the process' pid (pid, line 267 in image) and virtual address of the page to be eliminated (virt, line 268 in image) to name the file that stores this page. We have

created a new function called int to_string that copies an integer into a given string. We use this function to make the filename using integers pid and virt. Here is that function declared in proc.c:

```c
void int_to_string(int x, char *c){
    if(x==0)
    {
        c[0]='0';
        c[1]='\0';
        return;
    }
    int i=0;
    while(x>0){
        c[i]=x%10+'0';
        i++;
        x/=10;
    }
    c[i]='\0';

    for(int j=0;j<i/2;j++){
        char a=c[j];
        c[j]=c[i-j-1];
        c[i-j-1]=a;
    }

}
```

(proc.c)

We need to write the contents of the victim page to the file with the name <pid>_<virt>.swp. But we encounter a problem here. We store the filename in a string called c. File system calls cannot be called from proc.c. The solution was that we copied the open, write, read, close etc. functions from sysfile.c to proc.c, modified them since the sysfile.c functions used a different way to take arguments and then renamed them to proc_open, proc_read, proc_write, proc_close etc. so we can use them in proc.c. Some examples:

```c
int
proc_write(int fd, char *p, int n)
{
  struct file *f;
  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;
  return filewrite(f, p, n);
}

```

```c
int
proc_close(int fd)
{
  struct file *f;

  if(fd < 0 || fd >= NOFILE || (f=myproc()->ofile[fd]) == 0)
    return -1;

  myproc()->ofile[fd] = 0;
  fileclose(f);
  return 0;
}

```

There are many more functions (proc_open, proc_fdalloc etc.) and you can check them out in proc.c. I can't paste all of them here.

Now, using these functions, we write back a page to storage. We open a file (using proc_open) with O_CREATE and O_RDWR permissions (we have imported fcntl.h with these macros). O_CREATE creates this file if it doesn't exist and O_RDWR refers to read/write. The file descriptor is stored in an integer called fd. Using this file descriptor, we write the page to this file using proc_write. Then, this page is added to the free page queue using kfree so it is available for use (remember we also wake up all processes sleeping on sleeping_channel when kfree adds a page to the free queue). We then clear the page table entry too using memset.

After this, we do something important: for Task 3, we need to know if the page that caused a fault was swapped out or not. In order to mark this page as swapped out, we set the 8th bit from the right (2^7) in the secondary page table entry. We use xor to accomplish this task (LINE 295 in image).

Suspending kernel process when no requests are left:

When the queue is empty, the loop breaks and suspension of the process is initiated. While exiting the kernel processes that are running, we can't clear their kstack from within the process because after this, they will not know which process to execute next. We need to clear their kstack from outside the process. For this, we first preempt the process and wait for the scheduler to find this process. When the scheduler finds a kernel process in the UNUSED state, it clears this process' kstack and name. The scheduler identifies the kernel process in unused state by checking its name in which the first character was changed to '*' when the process ended.

The ending of kernel processes has two parts:

1. From within process:

```
    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();
}
```

(proc.c)

2. From Scheduler

```
        name.
        if(p->state==UNUSED && p->name[0]=='*'){

          kfree(p->kstack);
          p->kstack=0;
          p->name[0]=0;
          p->pid=0;
        }
```

(proc.c)

TASK 3

We first needed to create a swap in request queue. We used the same struct (rg) as in Task 2 to create a swap in request queue called rqueue2 in proc.c. We also declare an extern prototype for rqueue2 in defs.h. Along with declaring the queue, we also created the corresponding functions for rqueue2 (rpop2() and rpush2()) in proc.c and declared their prototype in defs.h. We also initialized its lock in pinit. We also initialized its s and e variables in userinit.

Next, we add an additional entry to the **struct proc** in **proc.h** called **addr (int).** This entry will tell the swapping in function at which virtual address the page fault occurred:

Proc.h(in struct proc):

```
char name[16];               // Process name (debugging)
int addr;                    // ADDED: Virtual address of pagefault
```

Next, we need to handle page fault (**T_PGFLT**) traps raised in trap.c. We do it in a function called handlePageFault():

Trap.c:

```
case T_PGFLT:
  handlePageFault();
break;
//PAGEBREAK: 13
```

```c
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;
    rpush2(p);
    if(!swap_in_process_exists){
      swap_in_process_exists=1;
      create_kernel_process("swap_in_process", &swap_in_process_function);
    }
  } else {
    exit();
  }
}
```
(trap.c)

In handlePageFault, just like Part A, we find the virtual address at which the page fault occurred by using rcr2(). We then put the current process to sleep with a new lock called swap_in_lock (initialized in trap.c and with extern in defs.h). We then obtain the page table entry corresponding to this address (the logic is identical to walkpgdir). Now, we need to check whether this page was swapped out. In Task 2, whenever we swapped out a page, we set its page table entry's bit of 7th order (2^7). This is mentioned at the beginning of the 5th page of this report. Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using bitwise & with 0x080. If it is set, we initiate swap_in_process (if it doesn't already exist - check using swap_in_process_exists). Otherwise, we safely suspend the process using exit() as the assignment asked us to do.

Now, we go through the swapping in process. The entry point for the swapping out process is swap_in_process_function (declared in proc.c) as you can see in handlePageFault.

Note: swap_in_process_function is shown on the next page since it is long. Refer to the next page for the actual function.

I have already mentioned how we have implemented file management functions in proc.c in the Task 2 part of the report. I will just mention which functions I used and how I used them here. The function runs a loop until rqueue2 is not empty. In the loop, it pops a process from the queue and extracts its pid and addr value to get the file name. Then, it creates the filename in a string called "c" using int to string .Then, it used proc_open to open this file in read only mode (O_RDONLY) with file descriptor fd. We then allocate a free frame (mem) to this process using kalloc. We read from the file with the fd file descriptor into this free frame using proc_read. We then make mappages available to proc.c by removing the static keyword from it in vm.c and then declaring a prototype in proc.c. We then use mappages to map the page corresponding to addr with the physical page that got using kalloc; and read into (mem). Then we wake up, the process for which we allocated a new page to fix the page fault using wakeup. Once the loop is completed, we run the kernel process termination instructions.

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

(proc.c)

```c
void swap_in_process_function(){

  acquire(&rqueue2.lock);
  while(rqueue2.s!=rqueue2.e){
    struct proc *p=rpop2();

    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(&rqueue2.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(&rqueue2.lock);
        panic("mappages");
      }
      wakeup(p);
  }

    release(&rqueue2.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();

}
```

16 ⚠ 0 0 Ln 325, Col 30 (24 selected) Spaces: 2

(proc.c)

TASK 4: Sanity Test

In this part, our aim is to create a testing mechanism in order to test the functionalities created by us in the previous parts. We will implement a user-space program named memtest that will do this job for us. The implementation of memtest is given below.

```c
// memtest.c > ...
#include "types.h"
#include "stat.h"
#include "user.h"

int compute(int val){
return val*val - 4*val + 1;
}

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

for(int idx=0; idx<20; idx++){
if(!fork()){
printf(1, "child number: %d\n", idx+1);
printf(1, "Iter | Matched Bytes | Different Bytes\n");
printf(1, "--------------------------------------\n\n");

for(int iteration=0; iteration<10; iteration++){
    int *buffer = malloc(4096);
    for(int index=0; index<1024; index++){
        buffer[index] = compute(index);
    }

    int countMatch = 0;
    for(int index=0; index<1024; index++){
        if(buffer[index] == compute(index))
            countMatch += 4;
    }

    if(iteration < 9)
        printf(1, "  %d  |    %dB    |    %dB\n", iteration+1, countMatch,
        4096-countMatch);
    else
        printf(1, "  %d  |    %dB    |    %dB\n", iteration+1, countMatch,
        4096-countMatch);
}
    printf(1, "\n");
    exit();
}
}

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

We can make the following observations by looking at the implementation:

- The main process creates 20 child processes using fork() system call.
- Each child process executes a loop with 10 iterations
- At each iteration, 4096B (4KB) of memory is being allocated using malloc()
- The value stored at index i of the array is given by the mathematical expression $i^2 - 4i + 1$ which is computed using compute().

- A counter named countMatch is maintained which stores the number of bytes that contain the right values. This is done by checking the value stored at every index with the value returned by the function for that index.

In order to run memtest, we need to include it in the Makefile under UPROGS and EXTRA to make it accessible to the xv6 user.

On running memtest, we obtain the following output.

```
$ memtest
child number: 1
Iter | Matched Bytes | Different Bytes
--------------------------------------

  1  |    4096B    |    0B
  2  |    4096B    |    0B
  3  |    4096B    |    0B
  4  |    4096B    |    0B
  5  |    4096B    |    0B
  6  |    4096B    |    0B
  7  |    4096B    |    0B
  8  |    4096B    |    0B
  9  |    4096B    |    0B
  10  |    4096B    |    0B

child number: 2
Iter | Matched Bytes | Different Bytes
--------------------------------------

  1  |    4096B    |    0B
  2  |    4096B    |    0B
  3  |    4096B    |    0B
  4  |    4096B    |    0B
  5  |    4096B    |    0B
  6  |    4096B    |    0B
  7  |    4096B    |    0B
  8  |    4096B    |    0B
  9  |    4096B    |    0B
  10  |    4096B    |    0B

child number: 3
Iter | Matched Bytes | Different Bytes
--------------------------------------

  1  |    4096B    |    0B
  2  |    4096B    |    0B
  3  |    4096B    |    0B
  4  |    4096B    |    0B
  5  |    4096B    |    0B
  6  |    4096B    |    0B
  7  |    4096B    |    0B
  8  |    4096B    |    0B
  9  |    4096B    |    0B
  10  |    4096B    |    0B

child number: 4
Iter | Matched Bytes | Different Bytes
--------------------------------------

  1  |    4096B    |    0B
  2  |    4096B    |    0B
  3  |    4096B    |    0B
  4  |    4096B    |    0B
  5  |    4096B    |    0B
  6  |    4096B    |    0B
  7  |    4096B    |    0B
  8  |    4096B    |    0B
  9  |    4096B    |    0B
  10  |    4096B    |    0B

child number: 5
Iter | Matched Bytes | Different Bytes
--------------------------------------

  1  |    4096B    |    0B
  2  |    4096B    |    0B
  3  |    4096B    |    0B
  4  |    4096B    |    0B
  5  |    4096B    |    0B
  6  |    4096B    |    0B
  7  |    4096B    |    0B
  8  |    4096B    |    0B
  9  |    4096B    |    0B
  10  |    4096B    |    0B
```

As can be seen in the output, our implementation passes the sanity test as all the indices store the correct value.