OS Lab Assignment 3

Group - 21

Tanmay Mittal	220101099
Sarthak Kapoor	220101116
Sushant Kumar	220101096
Tanay Goenka	220101098

File Link: https://github.com/Tanmay7404/CS344_OsLab_2024/tree/main/Assignment%203

Part - A

Lazy Memory Allocation:

- **Lazy allocation** (also known as **demand allocation**) is a memory management strategy where physical memory is **not actually allocated** to a process until it is needed. Instead of allocating memory as soon as a process requests it, the operating system waits until the process attempts to access that memory (e.g., when it tries to read from or write to that address). At that moment, the operating system allocates the required memory page.
- Xv6 uses `sys_sbrk()` system call to allocate memory to the process as soon as the process requests it.
- `sys_sbrk()` calls growproc() ,which in turns call `allocuvm()` which in turn is responsible for allocating required pages to process and mapping process virtual address to its physical address in its process table.

To delay the allocation of memory for its pages until it is accessed , we instead move these functions in trap.c.

- We do this by first commenting out growproc() in sys_sbrk() and **changing the process size variable** to make it feel the desired memory has been allocated.

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

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

int
```

- Now , when the process tries to access this memory , it will encounter a **T_PGFLT.**

```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap st
t 58
init: starting sh
$ echo hi
pid 3 sh: trap 14 err 6 on cpu 0 eip 0x1220 addr 0x4004--kill proc
$
```

- To handle this page fault , we introduce a new case in the trap handler such that if the trap occurs due to a page fault , we can handle it accordingly.

```
    lapiceoi();
    break;
  case T_PGFLT:
    if(PageFaultFunction()<0){
      cprintf("Could not allocate page. Sorry.\n");
      panic("trap");
    }
  break;
  //PAGEBREAK: 13
  default:
```

- So, to handle above page fault , we will make a function **PageFaultFunction()**, and also declare mappages so its scope can be accessible inside **trap.c**

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

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

- Inside PageFaultFunction() , **rcr2()** gives the virtual address at which the page fault occurs, nearest_addr variable points to the just next address, which is multiple of 4096(page size), thus **internal fragmentation** occurs here.
- Now, we call kalloc(), which chooses a free frame on physical memory using freelist data structure(**linked list**) and returns its address and **store in memory**.
- Now , we use mappages() which create PTE for virtual address nearest_addr and map it with V2P(memory). Kernel returns **address of available free frame** in virtual space ,

so we need to **convert it in physical space to map it**, and this is what **V2P** does by subtracting KERNBASE from it.

- Inside mappage, ‘a’ denotes the first page and ‘last’ denotes the last page of the data that has to be loaded. It then runs a loop until all the pages from the first to last have been loaded successfully. For every page, it loads it into the page table using walkpgdir()
- The `walkpgdir()` function in xv6 navigates a two-level page table structure to locate the page table entry (PTE) for a given virtual address. It takes a page directory and the virtual address as input. Using the first 10 bits of the virtual address (extracted with the `PDX` macro), it identifies the page directory entry, which points to the corresponding page table. Then, it uses the next 10 bits of the virtual address (extracted with the `PTX` macro) to locate the specific PTE within that page table. If the page table already exists in memory, it retrieves the pointer to its base address using the `PTE_ADDR` macro. If the page table isn't present, `walkpgdir()` allocates a new page table, sets the appropriate permissions in the page directory, and then returns a pointer to the desired PTE for the virtual address.
- After successfully allocating memory, trap returns to the user space, so **the instruction is re run**, which now runs successfully as **present bit (P)** is set now

```
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
$ _
```

Part - B

Answers to part-B Questions

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

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

1) How does the kernel know which physical pages are used and unused?
 Ans – xv6 maintains a list of free pages in **free list** data structure in kalloc.c.
2) What data structures are used to answer this question?
 Ans – linked list
3) Where do these reside?
Ans) 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**
4) Does xv6- memory mechanism limit the number of user processes.
Ans) Due to the limit in size of ptable(**NPROC**), no of processes has a limit.
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, it starts with a single process called initproc. This process plays a crucial role as it forks the sh (shell) process, which in turn creates other user processes. In xv6, each process can have a virtual address space of up to 2 GB (starting from KERNBASE), while the maximum physical memory is limited to 128 MB (PHYSTOP). This means that a single process has the potential to use up all of the available physical memory.

Address Translation:

- A virtual address in the x86 architecture is broken into three parts: **Page Directory Index (PDX)**, **Page Table Index (PTX)**, and **Offset within Page**.
  - Macros like `PDX(va)` and `PTX(va)` extract the page directory and table indexes from a virtual address.
  - `PGADDR(d, t, o)` constructs a virtual address using the directory, table, and offset values.

- Constants like `NPDENTRIES`, `NPTENTRIES`, and `PGSIZE` specify the size of page directories, page tables, and pages respectively.

Task 1:

```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 initia
  //Also, since this doesn't need to have a userspace, we don't need to assign a size to this

  //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);

}
```

We have created the function create_kernal_process in proc.c to create a kernel process with the given name and entry point.The function starts by calling `allocproc()`, which allocates and initializes a process structure. Next, it sets up the kernel's page table by calling `setupkvm().`

The `eip` (Extended Instruction Pointer) register stores the address of the next instruction to be executed. This is set to the `entrypoint` passed to the function, which is the function pointer that the kernel process will begin executing.

The process name is copied to the `p->name` field using `safestrcpy().`

It sets the state of the process to `RUNNABLE`, which means the process is ready to run.

This function will be used to create 2 kernel processes for swapping_in from memory and swapping_out from memory.

Preparations for Tasks 2 and 3:

1) File_Management:

We will need to create, open, read, write etc files containing data stored in pages of the process. For that we created functions in proc.c

```c
int
proc_close_file(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;
}
```

```c
int
proc_write_file(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
static struct inode*
proc_create_file(char *path, short type, short major, short minor)
{
  struct inode *ip, *dp;
  char name[DIRSIZ];

  if((dp = nameiparent(path, name)) == 0)
    return 0;
  ilock(dp);

  if((ip = dirlookup(dp, name, 0)) != 0){
    iunlockput(dp);
    ilock(ip);
    if(type == T_FILE && ip->type == T_FILE)
      return ip;
    iunlockput(ip);
    return 0;
  }

  if((ip = ialloc(dp->dev, type)) == 0)
    panic("create: ialloc");

  ilock(ip);
  ip->major = major;
  ip->minor = minor;
  ip->nlink = 1;
  iupdate(ip);

  if(type == T_DIR){  // Create . and .. entries.
    dp->nlink++;  // for ".."
    iupdate(dp);
    // No ip->nlink++ for ".": avoid cyclic ref count.
    if(dirlink(ip, ".", ip->inum) < 0 || dirlink(ip, "..", dp->inum) < 0)
      panic("create dots");
  }

  if(dirlink(dp, name, ip->inum) < 0)
    panic("create: dirlink");

  iunlockput(dp);

  return ip;
}
```

```c
int proc_open_file(char *path, int omode){

  int fd;
  struct file *f;
  struct inode *ip;

  begin_op();

  if(omode & O_CREATE){
    ip = proc_create_file(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  if((f = filealloc()) == 0 || (fd = proc_fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
  return fd;

}
```

```c
static int
proc_fdalloc(struct file *f)
{
  int fd;
  struct proc *curproc = myproc();

  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd] == 0){
      curproc->ofile[fd] = f;
      return fd;
    }
  }
  return -1;
}
```

These functions are similar to functions in sysfile.c

2) **string_from_int** : function to facilitate conversion of int to string

```c
void string_from_int(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;
  }

}
```

3) **Created 2 Circular Queue Structure**: Having lock, a queue of process (head and tail pointer to represent head and tail of circular queue) and Push and Pop functionality

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

```c
struct cir_q swap_out_queue;
```

```c
struct cir_q swap_in_queue;
```

```c
struct proc* cqpop(){

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

  return p;
}
```

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

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

  return 1;
}
```

Similar push and pop functions for swap_in_queue

Whenever the user process is initialized (inside the userinit function), both the queues are initialized

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

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

```

And their locks are initialized in pinot function:

```
pinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&swap_out_queue.lock, "swap_out_queue");
  initlock(&sleeping_channel_lock, "sleeping_channel");
  initlock(&swap_in_queue.lock, "swap_in_queue");
}
```

4) **Process Suspending Until Free Memory is Present:**

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

```

We create a global lock on variable sleeping_channel_count which stores the count of number of process sleeping while waiting for free memory. Any process waiting for free memory would be sleeping on channel sleeping_channel

Inside the allocuvm function of vm.c when a process tries to acquire new memory, kalloc function returns 0 if memory cannot be allocated. In that case, the process will go to sleep(state is SLEEPING). Also swap_out_kernel process is started if not running (more on this later) and process is added to swap_out queue to be swapped out.

```c
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);

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

      return 0;
    }
```

Whenever a page is freed, kfree is called. So all the processes waiting for free page are woken up (wakeup called on sleeping_channel)

```
  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);

}
```

Task 2:

Whenever a process is unable to allocate memory, it is suspended, and added to swap_out queue to be swapped out. A global variable is declared to check if the kernal process swap_out_process is running. If not, it is runned with the function created in Task 1

```
int swap_out_process_exists=0;
```

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

The function acquires lock on swap_out queue. While the queue is not empty, it iterates to all the process to be swapped out (in the queue) and pop them.

```
void swap_out_process_function(){

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

    pde_t* pd = p->pgdir;
```

For each process p, the function iterates over its page directory entries (PDEs), each of which points to a page table. If the page directory entry pd[i] has been accessed (PTE_A), it is skipped. (This is checked by checking the 6th (accessed bit) of the page dir entry) This means that the page table has recently been accessed, and the system assumes that its pages are still in use.

```
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;
```

If page_dir is not accessed, then we iterate over the entries of the corresponding page table. If the entry is access (still in used) or invalid, it is not stored.

```
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;
```

The page corresponding to physical address stored in page table has to be stored in a file. For this, pid and virtual address is extracted and file name is created accordingly. File with the given name is opened, content of physical address is stored there, and the file is closed. The physical address page is freed by calling kfree. The page_table entry is nullified and marked as swapped out.

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

        // file management
        int fd=proc_open_file(c, O_CREATE | O_RDWR);
        if(fd<0){
          cprintf("error creating or opening file: %s\n", c);
          panic("swap_out_process");
        }
          int proc_write_file(int fd, char *p, int n)
        if(proc_write_file(fd,(char *)pte, PGSIZE) != PGSIZE){
          cprintf("error writing to file: %s\n", c);
          panic("swap_out_process");
        }
        proc_close_file(fd);

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

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

        break;
      }
    }
  }
```

When the swap_out queue is empty, the swap_out_process exits, releasing the lock. It's state is set to unused, and name is marked with * for scheduler to terminate the process and clear it.

```c
    }

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

```c
scheduler(void)
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

      //If the swap out process has stopped running, free its stack and name.
      if(p->state==UNUSED && p->name[0]=='*'){

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

Task 3:

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. When a

Page_Fault occur (in trap.c when trapno = T_PGFLT) handlePageFault function is called.

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

```
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();
  }
}
```

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). 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 bit of 7th order (2^7). Thus, in order to check whether the page was swapped out or not, we check its 7th order bit using bitwise & with 0x0080. If it is set, we add a process to swap in the queue. A global variable is declared to check if the kernel process swap_in_process is running. If not, it is runned with the function created in Task 1.

Otherwise, we safely suspend the process using exit() as the assignment asked us to do.

```c
void swap_in_process_function(){

  acquire(&swap_in_queue.lock);
  while(swap_in_queue.s!=swap_in_queue.e){
    struct proc *p=rcqop2();

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

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

      int fd=proc_open_file(c,O_RDONLY);
      if(fd<0){
        release(&swap_in_queue.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(&swap_in_queue.lock);
        panic("mappages");
      }
      wakeup(p);
  }
```

Similar to swap_out_process_function, our swap_in_process_function iterates to all the process in the queue. It gets it's pid and virt address from p->addr. It opens the corresponding file, where the page info would be stored. Then a physical memory is allocated where the content of the file are copied. The physical address is mapped to the virt address of the process.

```c
        release(&swap_in_queue.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();

}
```

If the swap_in queue is empty, the process is marked as unused and named with *. This allows scheduler to identify and terminate the process and free it's memory

Task 4: Created a file test.c

Here 20 children are forked, and 4KB memory is allocated in each child. The memory is populated with random values from our function (depending on child number and mem num). Then we check if the data is stored correctly

```c
int math_func(int num, int j){
    return (num*num*num - 5*num + 3)*(j+1);
}

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

    for(int i=0;i<20;i++){
        if(!fork()){
            printf(1, "Child %d\n", i+1);
            printf(1, "Iteration Matched Different\n");
            printf(1, "--------- ------- ---------\n\n");

            for(int j=0;j<10;j++){
                int *arr = malloc(4096);
                for(int k=0;k<1024;k++){
                    arr[k] = math_func(k,i);
                }
                int matched=0;
                for(int k=0;k<1024;k++){
                    if(arr[k] == math_func(k,i))
                        matched+=4;
                }

                if(j<9)
                    printf(1, "    %d       %dB       %dB\n", j+1, matched, 4096-matched);
                else
                    printf(1, "   %d       %dB       %dB\n", j+1, matched, 4096-matched);

            }
            printf(1, "\n");

            exit();
        }
    }

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

Child 1

Count	Match-Found	Difference
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 2

Count	Match-Found	Difference
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 3

Count	Match-Found	Difference
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 4

Count	Match-Found	Difference
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 5

Count	Match-Found	Difference
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 6

Count	Match-Found	Difference
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 7

Count	Match-Found	Difference
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 8

Count	Match-Found	Difference
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 18
Count Match-Found Difference
    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 19
Count Match-Found Difference
    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 20
Count Match-Found Difference
    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
```