3️⃣ Project03

2018008068 김재형

목차

1️⃣ 과제 명세

기존의 xv6는 프로세스의 스케줄링을 지원하지만, 스레드의 스케줄링을 지원하지 않는다는 점에서부터 과제는 시작합니다. 추가적으로 POSIX Thread 의 동기화 라이브러리를 사용하지 않고, Lock 과 Unlock 을 구현해보기로 합니다.

1. Thread Scheduling 구현

👉 POSIX THREAD 구현

- thread_create, thread_exit, thread_join

👉 THREAD 스케줄링으로 인한 시스템 콜 수정

- Fork, Exec, Sbrk, Kill, Sleep, Pipe

2. Lock & UnLock 구현

👉 C라이브러리에서 제공하는 동기화 API (pthread_mutex) 사용하지 않고 구현

🔎 기존 Xv6 코드 분석 (Process의 작동과정 위주로)

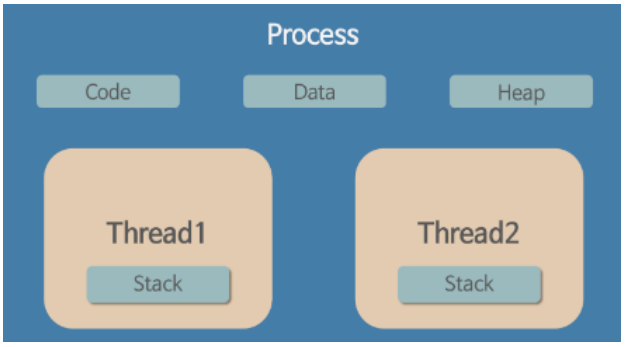
⚖️ 목표

기존 xv6 의 프로세스의 fork, exec의 과정과

기존 xv6의 페이징 단위 메모리 관리기법(Hierarchial memory paging) 이 어떠한 코드를 통해서 이루어지는지 알아보는 과정을 통하여, 프로세스와 일부다른 thread의 경우 어떠한 방식으로 스케줄링을 구현하면 될지, 대략적인 design 을 생각해볼수있습니다.

📖 배경지식

1. 스레드(Thread) 란 프로세스 내에서 실행되는 여러흐름의 단위로, Light Weight Process(LWP) 라고 이해하면됩니다.



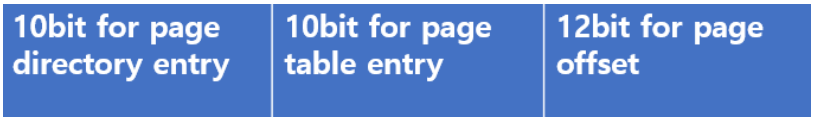
이때 스레드는 프로세스내에서 **Stack 만 따로 할당**받고, Code Data Heap 영역은 공유합니다.

각각의 스레드는 **별도의 레지스터와 스택**을 가지고 있지만, **힙메모리는 서로** 읽고 쓸수 있습니다.

2. **32bit CPU 의 Xv6**

Xv6 는 총 32비트를 다루며 , 4KB의 페이지 크기 할당방식을 사용합니다. 물리메모리 주소는 1Byte 단위이며, **32bit의 가상주소는 총 2^32 x 1Byte = 4GB** 의 주소를 표현할수있습니다.

이때, 물리메모리 와 가상메모리는 모두 동일한 4KB 의 페이지 단위를 사용하며, 4KB 페이지 내에서 내가 원하는 Offset으로 접근하기 위해서는 12bit 가 필요합니다. 이를 통해 page table entry , 즉, page 의 개수는 2^20 (32-10 =22) 가 필요하다는것을 알 수있습니다. 하지만 해당 page table entry에는 물리메모리의 주소값이 들어가있지만, 해당 주소값들 역시 페이징 기법으로 인하여 4KB 단위로 나누어 떨어져있어야 합니다. 이렇게 될시 2^20의 offset으로는 찾기에 한계가 있습니다. 이를 해결하기 위해서 10bit 는 Page directory Entry 를 표현하여야 하며, 나머지 10bit로 page table entry 값을 표현할수 있어야합니다.




```
Click to add a breakpoint
// +--------10------+-------10-------+---------12----------+
// | Page Directory |   Page Table   | Offset within Page  |
// |      Index     |      Index     |                     |
// +----------------+----------------+---------------------+
//  \--- PDX(va) --/ \--- PTX(va) --/

// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o))
```

(memory managing unit)mmu.h 파일을 참조할시, 다음과같이 구현되어있는것을 확인할수있습니다.

즉, 가상주소 32bit 하나만으로 physical address → page directory entry→ page table entry → PYHSICAL MEMORY 로 접근이 가능한것으로 확인할수있습니다.

❓ Fork 시스템콜별 실행순서 분석

기존 프로세스의 생성시 가상메모리 및 물리메모리 할당이 어떠한 함수를 사용하고 해당 함수에서 어떠한 일들이 일어나는 분석을 한 후에, 프로세스 생성시 일어나는 일을 알아보았다면, thread 생성시에는 과연 어떠한 메모리 할당과정이 필요한지 추측해봅니다.

1. np = allocproc() 으로 프로세스를 할당합니다.

allocproc에서 일어나는 일을 분석해보면 다음과같습니다.

2. allocproc()에서는 kalloc() 함수가 실행됩니다. (fork 된 프로세스의 kernel을 allocation 하는 과정이라고 생각하면됩니다.)
3. kalloc 함수에서 일어나는 일은 다음과같습니다.
    a. kmem의 freelist에서 하나의 주소값을 가져옵니다.(물리메모리 주소값을 가져옵니다.)

a → step 을 통하여 해당 프로세스의 kernel 영역의 bottom address를 저장합니다.

```
95
96    // Allocate kernel stack.
97    if((p->kstack = kalloc()) == 0){
98      p->state = UNUSED;
99      return 0;
100   }
101   sp = p->kstack + KSTACKSIZE;
102
```

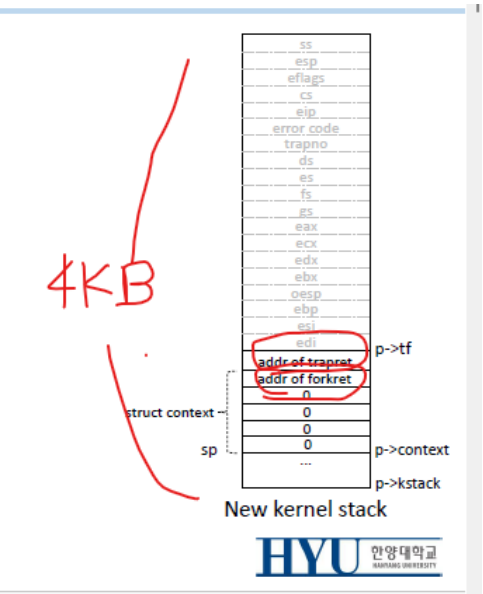
위 코드는 allocproc 함수의 일부입니다. sp 는 커널의 스택 포인터를 정의합니다.

sp = p→kstack(커널스택의 가장 하단주소) + KSTACKSIZE(4096) 인것으로 보아 kernel 영역의 stack 사이즈는 4KB 인것을 유추할수있습니다.

allocproc이 정상적으로 이루어진이후(kernel 영역할당 및 kernel 영역의 stack pointer 세팅)

이때 kernel 영역의 stackpointer 세팅에는 아래 그림과같이 trapret, forkret 값세팅이 포함됩니다.

```
// Leave room for trap frame.
sp -= sizeof *p->tf;
p->tf = (struct trapframe*)sp;

// Set up new context to start executing at forkret,
// which returns to trapret.
sp -= 4;
*(uint*)sp = (uint)trapret;

sp -= sizeof *p->context;
p->context = (struct context*)sp;
memset(p->context, 0, sizeof *p->context);
p->context->eip = (uint)forkret;

return p;
```



forkret는 새로 생성된 프로세스가 처음 실행할때 호출되는 함수로 이해하면됩니다.

allocproc함수에서 p→context 를 초기화할시, eip = forkret로 설정하는데 이것은 새 프로세스가 처음 스케줄링 되어 실행될시 forkret 함수에서 시작하도록 합니다.

또한 trapret는 커널모드에서 사용자 모드로 전환할시, 프로세스가 올바르게 사용자 모드로 돌아갈수 있도록 합니다. 즉, forkret 함수 실행이후, trapret 를 실행하도록 합니다.

다시 본론으로 돌아와서 allocproc을 통하여 프로세스 할당이후 kernel stack 영역을 세팅한 이후의 과정인, user virtual memory 영역을 할당할 예정입니다.

```
// Copy process state from proc.
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
  kfree(np->kstack);
  np->kstack = 0;
  np->state = UNUSED;
  return -1;
}
np->sz = curproc->sz;
```

이는 copyuvm으로 이루어집니다. copyuvm 함수는 다음과같은 프로세스로 구성되어집니다.

결론적으로 copyuvm(*pde_t *pgdir*, *uint sz*) 은 부모프로세스의 user virtual memory 영역의 sz (size) 만큼 자식프로세스의 page table 을 복사합니다.

1. setupkvm 함수 실행
    a. 새로운 프로세스에 대한 kernel 영역의 페이지 디렉토리를 할당합니다.
    b. mapapges 함수가 실행됩니다. 이때 mappage에 대한 설명은 아래에 적어놓았습니다.

```
220
221   for(i = 0; i < NOFILE; i++)
222     if(curproc->ofile[i])
223       np->ofile[i] = filedup(curproc->ofile[i]);
224   np->cwd = idup(curproc->cwd);
225
226   safestrcpy(np->name, curproc->name, sizeof(curproc->name));
227
228   pid = np->pid;
229
230   acquire(&ptable.lock);
231
232   np->state = RUNNABLE;
233
234   release(&ptable.lock);
```

`np->ofile[i]` 값은 새로운 프로세스의 파일 디스크립터 테이블이라고 이해하면됩니다. 파일 디스크립터란 운영체제에서 파일또는 기타 입출력 소스를 관리하는 소스라고 이해하면됩니다.

Mappages 함수 설명

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

해당 함수는 va ~ va+size 까지의 범위를 pagerounddown 을 이용하여 page단위로 세팅한후 ,

해당 가상주소에 해당하는 page table entry 값들을 모두 돌면서 해당 page table entry 의 비트 값들을 수정하면서 올바른 physical memory 주소에 매핑될수있도록 합니다.

WalkPgdir 함수 설명

walkpgdir 의 return 값은 pte_t* 입니다. 이는 page table entry 값을 의미합니다.

input 값은 (pgdir, va, alloc)이며, pagedirectory 시작주소와, virtual address 값으로부터 page table entry 값을 반환합니다. 이때 page table entry 값은 virtual address에 해당하는 물리주소를 표현할수있는 가상메모리 주소값이라고 이해하면 됩니다.

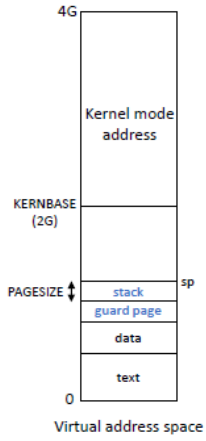
반환값은 va 에 해당하는 page table entry의 주소값으로 이해하면됩니다.

❓ Exec 시스템콜별 실행순서 분석

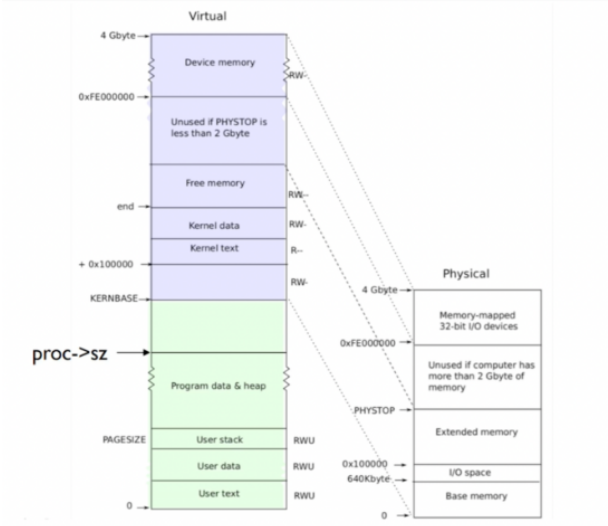
● Exec 시스템 콜은 **덮어씌운다는 의미**로 이해하면 됩니다. Exec 시스템 콜을 호출한 현재 프로세스의 공간은 TEXT, DATA, Block Symbol 영역을 새로운 프로세스의 이미지로 덮어씌웁니다. 이때 별도의 프로세스 공간을 만들지 않습니다. 즉, exec 시스템 콜은 새로운 프로세스를 만드는 것이 아니라, 현재 만들어진 exec 인자에 있는 프로그램 실행 파일을 읽은 후에 부모 프로세스 공간의 exec 인자에 있는 실행파일에 대한 TEXT, DATA, SYMBOL 영역을 덮어씌웁니다. 즉, **별도의 메모리 공간에 복사하지 않는 다는것이 핵심**입니다.

1. setupkvm 함수를 통하여 kernel 영역의 page directory 를 할당한후에 이에 맞는 가상메모리주소와 물리메모리 주소를 올바르게 매핑합니다.
2. allocuvm 을 통하여 기존에 sz 의 값에서 user virtual memory 의 값이 세팅되어있었지만, ph.vaddr + ph.memsz 만큼 sz 값을 새롭게 세팅합니다.

3. exec 을 통하여 새롭게 페이지 할당시 두 페이지를 할당한후에 위에 page 하나를 stack 영역으로 지정합니다. 이때 아래 page는 빈페이지로 guard page 로 세팅합니다.



❓ Physical Memory 할당과정 분석



Xv6는 위의 그림과같이 가상메모리주소 4GB 중에서 상위 2GB는 kernel 영역으로, 하위 2GB는 User 영역으로 할당하며 이에 해당하는 Physical memory 역시 매핑하고있습니다.

그렇다면 xv6어느 코드 부분에서 physical memory 를 할당하는지 궁금할수있습니다.

```
1  // Memory layout
2
3  #define EXTMEM  0x100000            // Start of extended memory
4  #define PHYSTOP 0xE000000           // Top physical memory (224MB)
5  #define DEVSPACE 0xFE000000         // Other devices are at high addresses
6
```

memory layout.h에서 PHYSTOP 은 224MB 로 쓰여져 있는것을 확인할수 있습니다.

그리고 해당 코드는

```
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // detect other processors
  lapicinit();     // interrupt controller
  seginit();       // segment descriptors
  picinit();       // disable pic
  ioapicinit();    // another interrupt controller
  consoleinit();   // console hardware
  uartinit();      // serial port
  pinit();         // process table
  tvinit();        // trap vectors
  binit();         // buffer cache
  fileinit();      // file table
  ideinit();       // disk
  startothers();   // start other processors
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
  userinit();      // first user process
  mpmain();        // finish this processor's setup
}
```

```
void
kinit1(void *vstart, void *vend)
{
  initlock(&kmem.lock, "kmem");
  kmem.use_lock = 0;
  freerange(vstart, vend);
}

void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
  kmem.use_lock = 1;
}
```

kalloc.c에서 kinit1, kinit2 에서 할당하는것을 확인할수있습니다.

이는 모든 physical memory 가 boot time에 kmem 구조체의 free list에 포함되는것으로 알수있습니다. kinit1() 함수는 초기 4MB 할당 , kinit은 이후 4MB~224MB 까지 할당하는것으로 알수있습니다.

kmem struct는 kernel memory 의 약자이며, freelist는 물리페이지의 비어있는 공간을 링크드 리스트 형식으로 가지고 있는것으로 알수있습니다.

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

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

kinit1, kinit2 에있는 freerange 함수는 vstart~ vend (가상메모리 주소의 시작~ 끝) 까지 돌면서

페이지단위로 재조정한후에 (PGROUNDUP) kfree 함수로 해당 메모리를 free 시킵니다.

kree 함수는 page를 1로 가득채운후에 freelist (page pool) 에 연결시킵니다.

```
void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

```
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);
}
void
```

2️⃣ Design 선택 및 Architecture 설계

🧐Xv6에서의 Thread 설계

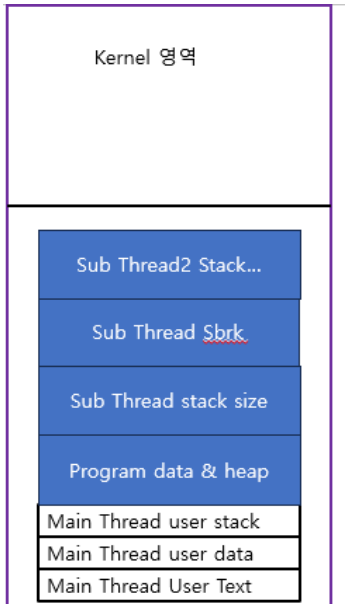
스레드 역시 Light Weight Process (LWP) 즉, 프로세스처럼 작동할수 있도록 설계하였습니다.

위 말은 , 스레드와 프로세스를 동일한 스케줄러 상에서 차별받지 않고 구현될수 있도록 하였습니다.

개념상으로는 process 구조체 아래에 thread 구조체를 두어서 하나의 프로세스가 여러개의 스레드를 소유하는 형식이 맞으나, 프로세스 기반으로 이루어진 기존 xv6를 스레드 기반으로 바꾸기에는 상당한 overhead 가 있을것으로 예상되어서 process 구조체를 그대로 두고 각 구조체마다 thread_id 값을 새롭게 추가하여 thread를 구현하였습니다. main thread일 경우 tid값은 0이지만, 생성된 thread일경우 tid 값은 0 초과로 설정하였습니다. 자세한 내용은 Implementation 에서 서술할계획입니다.

🏭 XV6 메모리 아키텍처 설계

아래 그림과 같이 xv6 메모리 아키텍처를 설계하였습니다.



원래는 Stack 영역은 Kernel Base, 즉 위에서부터 내려오는 것으로 설계하는것이 이론상으로 맞으나 ,

xv6 에서는 stack size 와 기존의 malloc 을 통한 sbrk는 size , 즉 stack pointer를 증가시키는 방향으로 , 즉 서로 같게 증가를 시키면서 메모리를 차곡차곡 쌓아가기에 다음과 같이 메모리를 할당하였습니다. 하지만, 이렇게 된다면 free, 즉 할당을 해제시킨 메모리에 대한 이슈가 생깁니다. 이부분은 future task 로 남겨두었습니다.

🔒 Lock & UnLock 설계

Lock 및 UnLock 을 구현하기 위해서 총가지 방식을 생각하였습니다.

1. **Bakery 알고리즘 :** 기존의 Peterson 알고리즘을 사용할려고 했으나 기존 Peterson 알고리즘은 2개의 프로세스에대해서 동시성을 보장해주는 것으로 확인하였습니다. 따라서 Peterson 알고리즘을 n개의 프로세스에서도 보장해줄수있도록 하는 flag, turn 개념이 아닌 ticket 번호 개념으로 알고리즘을 구현하였습니다.
2. Semapore를 이용하여 구현하였습니다. lock 시 P 함수를 호출하여 감소시키며, unlock시, V를 호출하여 증가시키는 원리를 이용하여 구현할수 있습니다. 이때 P,V를 이용하여 더하기 및 마이너스를 할시 test_and_set, swap 과같은 atomic 한 함수를 사용하여야 했으므로, 이때는 test_and_set 하드웨어 라이브러리를 사용하였습니다.

3️⃣ Implementation

0. Process 구조체 수정

```
52
53    thread_t tid; // Thread ID (0 if main thread)
54    struct proc* main;  // Main thread of current processs (0 if main thread)
55    void* retval;  // Temporary save return value
56    uint sbase; // Base address for the stack of a new thread
57  };
58
```

- Thread 스케줄링을 프로세스와 동일한 level에서 작동시키기 위해서는 pid 이외에 thread_id 값을 추가합니다.
- main thread의 주소값을 알기위해 main 값을 새로이 선언합니다.
- 새로운 trhead의 user stack base address 를 알기위해 sbase 값을 선언합니다.
- thread_join 의 return value 를 저장하기 위한 retval 값을 선언합니다.

1. Thread_create 구현

Process 의 생성부분과 무엇이 다른지 알아야한다. Process의 모든부분을 동일한 메모리 영역을 가리키도록 복사하되, stack 영역과, 레지스터 값은 독립적인 값을 가지도록 세팅하여야합니다.

1. allocproc 으로 동일 프로세스를 생성합니다. (증가된 pid 값감소)
2. pgdir 값이 main thread 와 동일한 값을 참조하도록 설정합니다.
3. main thread 값의 stack 영역을 증가시킵니다.
4. 생성된 스레드의 user stack 영역 2페이지를 확보합니다. 이때 첫페이지는 가드 페이지로 설정합니다.
5. Trap frame 값역시 분리되어야 하므로 trap frame 값역시 분리합니다.
6. PC(esp regitser)값 역시 분리되어야하므로 따로 설정합니다.

- thread_create main Logic

```
int
thread_create(thread_t* thread, void *(*start_routine)(void *), void *arg)
{
 ........

  if(!(curproc->main)) main = curproc;
  else main = curproc->main;

  // 프로세스 할당
  if((np=allocproc()) ==0) return -1;
  nextpid--;

  // thread를 의 초기값 할당 ㅇㅇ
  np->tid = nexttid++;
  *thread = np->tid;

  np->main = main;
  np->parent = main;
  np->pid = main->pid;

  pgdir = main->pgdir;
  sbase = main->sz;
```

```
  sz = main->sz;
  main->sz += 2 * PGSIZE;

  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0){
    np->state = UNUSED;
    return -1;
  }

// User stack 영역 확보 첫번째 페이지는 접근 불가 설정
  clearpteu(pgdir,  (char*)(sz - 2*PGSIZE));
  content[0] = 0xffffffff; // Fake return address. Never return.
  content[1] = (uint)arg;

  sp = sz - 8;
  if(copyout(pgdir, sp, content, 8) != 0){
    panic("create_tread");
  }

  // thread 이므로 동일한 page dir 가지도록 설정 기존 process와
  // stack 영역은 대신 분리되어야한다.
  np->pgdir = pgdir;

  np->sz = main->sz;
  np->sbase = sbase; // sbase 분리

  *np->tf = *main->tf;

  np->tf->eip = (uint)start_routine;
  np->tf->esp = sp;

 ....
}
```

2. Thread_exit 구현

1. 현재 thread가 종료될시, 기존 main thread 의 channel에서 자고있는 sub thread를 모두 wakeup시킵니다. 이때 기존 프로세스의 exit 함수와의 차이점을 살펴봐야합니다.
2. 이후sched함수를 호출하여 scheduler에게 실행상태를 넘깁니다.

```
void
thread_exit(void *retval)
{
  struct proc *curproc = myproc();
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Save return value.
  curproc->retval = retval;
```

```
  // Using fileclose, decrement file ref count by 1
  // Files won't be closed because main thread has ref count.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Main thread might be sleeping in thread_join().
  wakeup1(curproc->main);

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

3. Thread_join 구현

1. ptable을 돌면서 thread_id 값을 찾아야하므로, 만일 찾았으며 해당 thread가 좀비상태일시 (완료가 되었다는 뜻이므로)0을 반환하도록 합니다.
2. 만일 해당 thread가 아직 sleep 상태가 아닐시 현재 curproc 를 sleep 상태로 전환합니다. wait sleep이 되는 형태로 만일 thread_exit 이 일어날시 main thread의 channel에서 wakeup이 일어납니다.

thread_join 은 wait 함수와 작동방식이 유사하며 wait 함수의 일부를 수정하여 만들었습니다 기존 wait 함수는 child process 가 종료되가까지를 기다리는 것이라면 thread 의 join은 thread을 정리하는 것이기에 pgdir 전체를 free 해서는 안됩니다. 즉 기존의 free와는 다른방식으로 작동시키는것을 지원하기 위해서 thread_clear를 통해서 구현하였습니다.

- Thread join main logic

```
  for(;;){
    // Scan through table looking for exited thread with given thread ID.
    found = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->tid != thread)
        continue;
      found = 1;
      if(p->state == ZOMBIE){
        // Found the one.
        // Put return value that saved in thread_exit to retval
        *retval = p->retval;
        thread_clear(p);

        release(&ptable.lock);
        return 0;
      }
    }
```

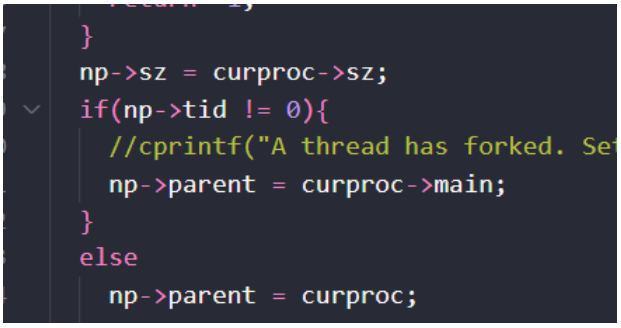
```
    // No point waiting if we cannot find the thread with given ID.
    if(!found || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for sub thread to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
```

- Thread Clear코드

```
void thread_clear(struct proc* p){
  // Do not free pgdir because it is shared with other threads.
  // freevm(p->pgdir);
  kfree(p->kstack);
  p->kstack = 0;
  p->pid = 0;
  p->tid = 0;
  p->main = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->state = UNUSED;
}
```

4. fork system call 수정

1. 만일 thread에서 fork를 수행하였을시, 해당 thread의 parent 는 현재 thread의 main thread로 설정합니다. 즉 메인 스레드가 아닌 스레드가 다른 프로세스의 부모가 되는 것을 막위한 코드를 추가했습니다.



5. exec system call 수정

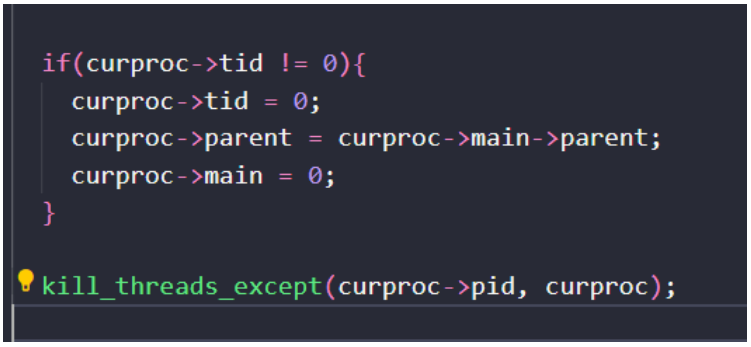
만일 exec 시스템 콜을 실행할시, 해당 시스템 콜을 실행한, 스레드를 제외한 모든 스레드가 종료된 이후에, 해당 스레드의 정보가 새로운 프로세스로 덮어씌워질수있도록 구현하였습니다.

이를 위해서 kill_threads_except라는 함수를 새롭게 구현하였습니다.

exec 시스템 콜을 호출하는 스레드는 메인 스레드일수도 있으며, 메인 스레드가 아닐수도 있습니다.

어떠한 스레드에서 호출되었던지 간에 해당 스레드는 남기고 모두 종료해야 했기에 exit를 참조해가며 작성하였습니다.

- 만일 메인 스레드가 아닌 스레드에서 exec 시스템 콜이 호출될시 , 메인스레드의 여러 attribute 등을 참조하여 main thread로써 작동할수 있도록 하며, 기존 메인 스레드를 포함한 나머지 모든 스레드를 정리하고 자원 반납을 하도록 하였습니다.



- kill thread except main logic

```
acquire(&ptable.lock);

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
  if(p->pid != pid || p == cp){
    continue;
  }

  for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
    if(q->parent == p){
      q->parent = initproc;
      if(q->state == ZOMBIE)
      wakeup1(initproc);
    }
  }
   // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(p->ofile[fd]){
      fileclose(p->ofile[fd]);
      p->ofile[fd] = 0;
    }
  }
   ....
  acquire(&ptable.lock);

  thread_clear(p);
```

6. sbrk systemcall 수정

- sbrk 함수는 내부적으로 growproc 함수가 호출되어서 heap영역을 n만큼 할당하는 함수입니다.

이때 thread에서 sbrk 가 호출될시 , main 스레드의 size 값이 증가되어서 할당되어야 하므로 해당 부분을 수정하였습니다.

```
sz = main->sz;
if(n > 0){
  if((sz = allocuvm(main->pgdir, sz, sz + n)) == 0)
    return -1;
} else if(n < 0){
  if((sz = deallocuvm(main->pgdir, sz, sz + n)) == 0)
    return -1;
}
main->sz = sz;
```

7. kill system call 수정

기존 시스템 콜의 경우 pid 에 해당하는 process를 kill 호출할시 해당 프로세스의 killed =1 로 설정한이후에 , trap.c에서 killed 되었을시 ,exit을 호출하여 프로세스를 종료합니다. 스레드 역시 이와 유사하지만,

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++
  if(p->pid == pid && p->tid ==0){
    p->killed = 1;
    // Wake process from sleep if necessary.
    if(p->state == SLEEPING)
      p->state = RUNNABLE;
```

다음과같이 main thread 의 kill 에대해서만 적용될수있도록 수정하였습니다.

8. sleep system call 수정

디자인 철학자체가 스레드와 프로세스 모두 프로세스단위로 sleep이 호출되므로 크게 변하는 것은 없습니다.

9. pipe system call 수정

Pipe 시스템 콜역시 크게 수정한 부분은 없습니다.

10. Exit 시스템콜 수정

- 만일 해당 프로세스가 종료될시 해당 프로세스의 subthread 역시 종료되어야하므로 subthread 종료 로직을 추가합니다.
- 현재 종료하는 프로세스가 main thread 일시부모 프로세스를 깨움으로써, 부모프로세스의 chan 에 있는 sleeping 프로세스를 깨우도록 합니다.
- 현재 종료하는 프로세스가 메인thread 가 아닐시, main thread 역시 죽여야하므로 main thread 의 killed 값을 1로 설정한후에 main thread wakeup을 실행합니다.

11. Locking & UnLocking 구현

0. Atomic 함수 사용

제가 설정한 mutex라는 변수가 atomic하게 동작할수 있도록 구현하였습니다.

동시에 여러개의 thread에서 shared_resource에서 접근할려고 하여도

atomic_compare_exchange_weak(&mutex, &expected, 0) 함수는 제가 설정한 mutex(atomic)한 변수값과 expected 값을 비교하여 값이 같을 시 expected을 atomic 에 저장된 값으로 바꾼후에 지정된 메모리 순서제약 조건을 적용하여 value로 설정합니다.

해당 함수를 사용하여 P,V 함수를 실행시 ,atomic을 보장하여 lock 과 Unlock 을 구현하였습니다.

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdatomic.h>

int shared_resource = 0;

#define NUM_ITERS 100000
#define NUM_THREADS 10000

atomic_int mutex = 1;

void lock() {
    int expected = 1;
    while (!atomic_compare_exchange_weak(&mutex, &expected, 0)) {
        expected = 1;
        usleep(100);  // CPU 사용을 줄이기 위해 작은 sleep을 사용할 수 있습니다.
    }
}

void unlock() {
    atomic_store(&mutex, 1);
}
```

1. Bakery알고리즘 으로 구현 (n개의 프로세스일시)

기존 Peterson 알고리즘은

flag 배열과 turn 변수를 사용하여 두개의 프로세스가 병렬적으로 실행할시 동기화를 구현하였습니다.

하지만 요번 과제에서는 n개의 프로세스가 병렬적으로 실행될때 두개 이상의 프로세스가 실행될수 있으므로 Peterson 알고리즘을 확장한 버전인 베이커리 알고리즘으로 구현하였습니다.

결론적으로 Atomic 변수 를 사용이 더 좋았으나 베이커리 알고리즘 역시 의미있었기에 적어놓았습니다.

베이커리 알고리즘은 번호표라는 개념을 부여하여서 자기순서가 올때까지 대기하며, 번호표 값이 낮을수록 우전으로 쓰레드가 Critical section 에 접근할수 있도록 합니다.

lock 함수 호출시 해당 스레드는 자신이 번호표를 받기 시작했다는 것을 알리기위해서 turn 배열의 값을 true 로 호출하고 현재 존재하는 라벨의 최대값에서 1을 더해주어 자신의 번호표가 나중에 뽑혔다는것을 나타냅니다 .

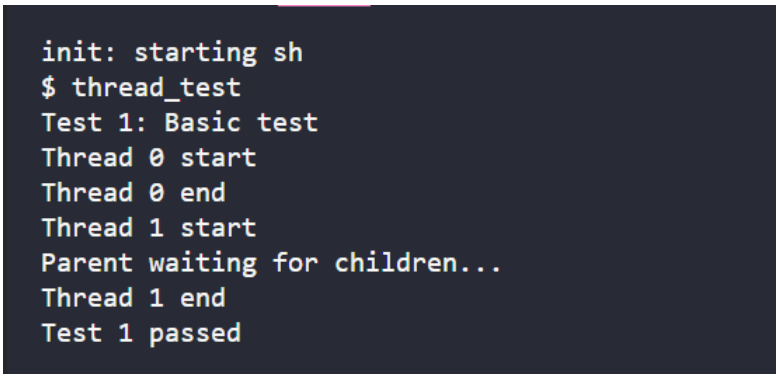
이후 다른 클라이언트들이 번호표를 뽑고있는지 체크한후, 다른 클라이언트들과의 번호표를 비교하여 대기후 내 차례가 되었을시 critical section 에 접근하게 됩니다.

```
int max_number() {
    int max = 0;
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (number[i] > max) {
            max = number[i];
        }
    }
```

```
    return max;
}


void lock(int thread_id)
{
    turn[thread_id] = true;
    number[thread_id] = 1 + max_number();
    turn[thread_id] = false;

    for (int i = 0; i < NUM_THREADS; ++i) {
        if (i == thread_id) continue;

        while (turn[i]);  // 다른 프로세스가 번호를 선택하는 동안 대기

        while (number[i] != 0 &&
              (number[i] < number[thread_id] ||
              (number[i] == number[thread_id] && i < thread_id))) usleep(100);
    }
}

void unlock(int thread_num)
{
    number[thread_num] = 0;
}
```

4️⃣ Result

Thread test ：Test 1 (정상작동 확인)



Threadt test : Test 2 (정상작동 확인)

```
Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
1 start
Child of thread 0 end
Thread 0 end
Child of thread 2 end
Thread 2 end
Child of thread 1 end
Child of thread 3 end
Child of thread 4 end
Thread 1 end
Thread 3 end
Thread 4 end
Test 2 passed
```

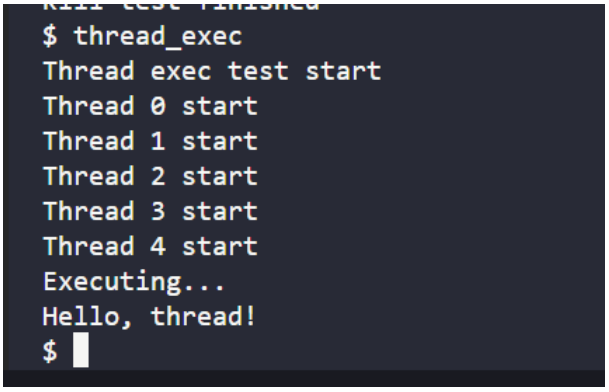
Thread Test: Test3:SBrk 정상작동 확인

```
Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
TThread 4 start
hread 3 start
Test 3 passed

All tests passed!
$
```

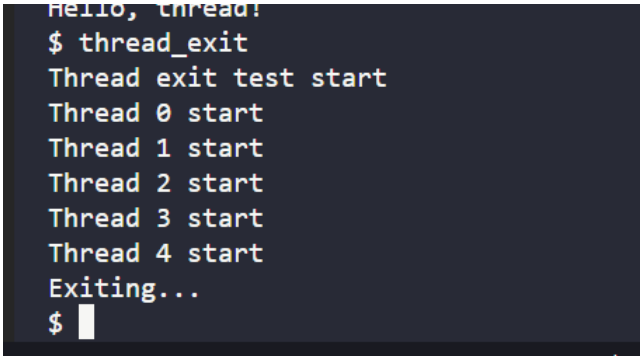
Thread kill test : 정상작동 확인

```
$ thread_kill
Thread kill test start
Killing process 10
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
Kill test finished
$
```

Thread exec : 정상작동 확인



Thread Exit : 정상작동 확인

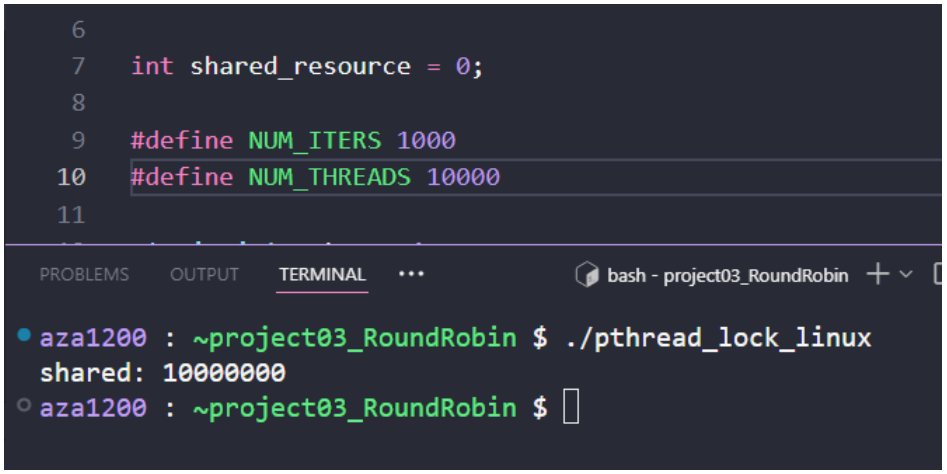


Lock & UnLock Test

Num iters: 10000

Num *threads* 10000

일시 test 결과 (정상작동확인)



Num iters: 100000 (십만)

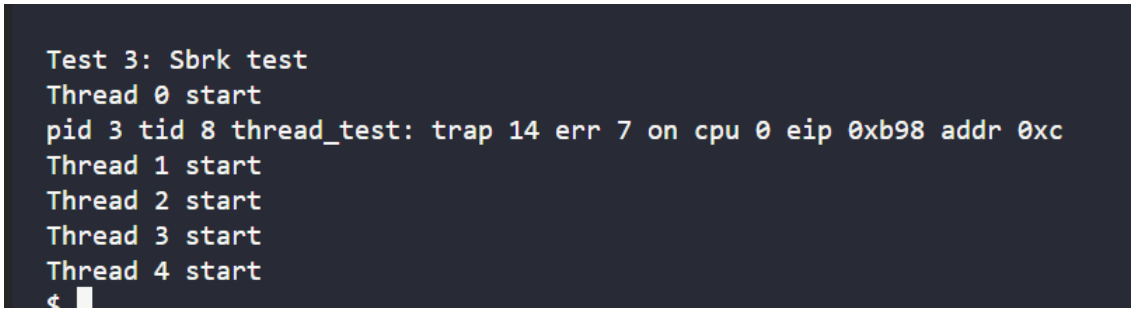
Num threads : 10000(만)

예측결과값 : 10억

test 사진 첨부 (정상작동 확인)

5️⃣ Trouble Shooting

🥲 SBRK Test Fail (최종적으로 해결함!)



- SBRK 테스트는 process의 thread 별로 64KB의 메모리를 할당하여 heap 영역이 다른 thread의 메모리 영역을 침범하였는지 침범하지 않았는지 확인하는 task 입니다. 해당 task를 해결하기 위해서 Lazy Page Allocation 기법을 적용해 보았으나 저의 코드 과정에서 오류인지는 모르겠으나 왜 틀렸는지 몰랐습니다.

다양한 thread들이 생성되면서 user heap 영역을 생성하였으나 이를 정상적으로 free 하지 못해서 생성된 에러라고 생각합니다. exit 시, 메모리 해제를 정상적으로 구현하지 못했다고 생각합니다.

대안으로 생각했던 방법 참고링크: https://pdos.csail.mit.edu/6.S081/2020/labs/lazy.html

모든 페이지를 할당하지않고, 필요하면 할당하는방식입니다.

만일, trap 함수로 14번 에러가 나왔을시 ,그때가 되어서야 mappage 함수로 할당하는 방식입니다.

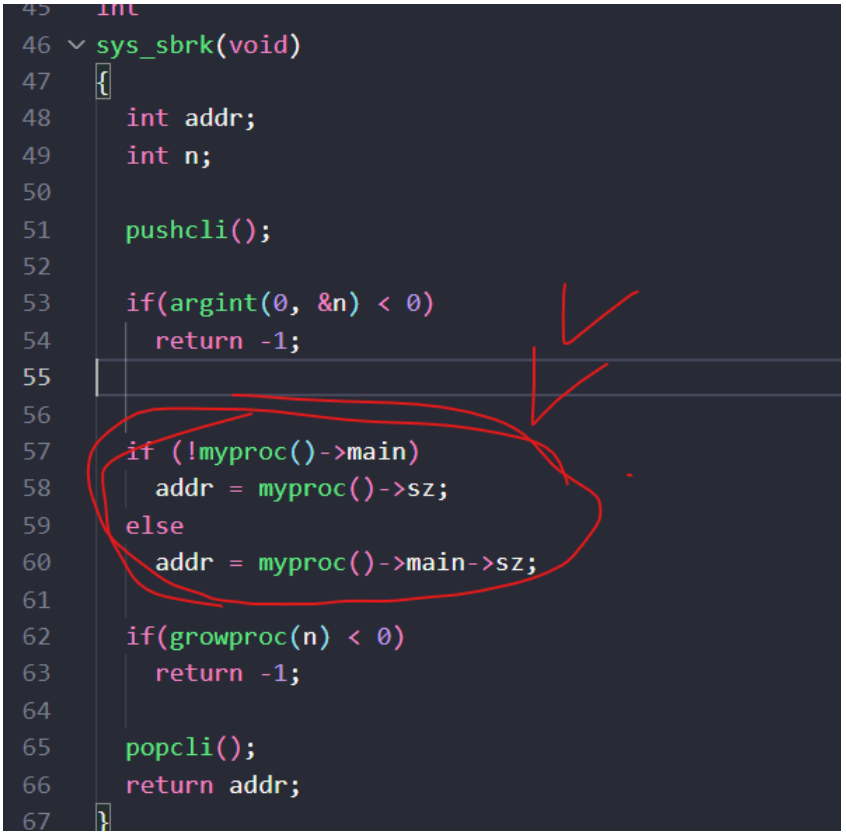
하지만, 해당 함수를 적용할시, fork, exit thread_create 등 많은 부분을 수정해야 했기에 sbrk 에러를 가지고 갔습니다..

Lazy Page Allocation 구현 일부코드

```
case T_PGFLT:
  a = rcr2();
  a = PGROUNDDOWN(a);
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(myproc()->pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U);
  break;
```

해결방안

- 결론적으로 말하자면 코드의 자그마한 부분의 실수였다.



빨간색 동그라미로 표시했던 부분은 원래 **addr = myproc()→sz** 로 표기했었다.

만일 shell 에서 어떠한 프로세스를 실행할시 , fork 를 실행한후 exec 을 호출한다. 위 과정에서

sbrk 를 호출하는데 , 이때

만일 myproc()→ main 일시, 즉 메인 스레드일시 addr = main→ sz로 설정해야 하며,

만일 main thread 가 아닐시 addr = myproc()→sz로 설정해야한다.

이러한 이유는 만일 main thread가 아닐시 이때에는 myproc()→ main = NULL 값으로 설정되어있기에 ,addr = myproc()→sz 로 설정했어야 했던것이다 .

🆚 Bakery vs atomic Function who is right?

1. **Bakery Algorithms DeadLock?**

결론적으로 말하면 Bakery 알고리즘을 사용하였지만 deadlock 이슈가 발생하였습니다.

thread, iter 중 10000개가 넘어갈지 프로세스가 종료가 되지않는 이슈가 발생했습니다.

deadlock 인지, 그냥 프로그램실행시간이 길어진건지 감이 잘 안잡혔습니다.

Num Iters = 1000, Num Threads =10000 예측값: 1000만 일시에는 잘되었으나

NUm Iters = 10000, Num Threads = 10000일시에는 종료가 되지않는 deadlock 이 걸린 것으로 확인되었습니다.

```
void lock(int thread_id)
{
    turn[thread_id] = true;
    number[thread_id] = 1 + max_number();
```

```
    turn[thread_id] = false;

    for (int i = 0; i < NUM_THREADS; ++i) {
        if (i == thread_id) continue;

        while (turn[i]); ;  // 다른 프로세스가 번호를 선택하는 동안 대기

        while (number[i] != 0 &&
              (number[i] < number[thread_id] ||
              (number[i] == number[thread_id] && i < thread_id))) usleep(100);
    }
}
```

이는 만일 turn[thread_id] = true로 동시에 설정할시...

어떠한 프로세스가 turn[thread_id] = true로 동시에 설정할시

while (turn[i]); ; 가 서로가 서로를 기다리는 알고리즘으로 작동하여 deadlock 이 걸렸을거라 추측합니다.

```
void unlock(int thread_num) { number[thread_num] = 0; }
```

이러한 문제를 해결하기 위해 결국에는 atomic 한 변수를 지원하는 C 라이브러리를 사용해야 했습니다.

- Part of pthread_lock_linux.c

```
void lock() {
    int expected = 1;
    while (!atomic_compare_exchange_weak(&mutex, &expected, 0)) {
        expected = 1;
        usleep(100);  // CPU 사용을 줄이기 위해 작은 sleep을 사용할 수 있습니다.
    }
}

void unlock() {
    atomic_store(&mutex, 1);
}
```

결론적으로 atomic 라이브러리를 지원하는 C 코드를 사용하여 간단하게 구현했습니다.