Project #2: Process Management & LWP

0. Design for Pmanager & LWP

TODO LIST

1. exec2
2. setmemorylimit
3. pmanager
4. LWP

1. Process with various stack size

1-1. exec2

```
int exec(char *path, char **argv);
int exec2(char *path, char **argv, int stacksize);
```

- exec2는 기존 xv6의 exec와는 달리 원하는 size의 stack을 할당받고 path경로의 파일을 argv의 인자를 받아서 실행시키는 system call이다.
- 일부 프로그램은 더 많은 stack이 필요할 수 있으므로 이와 같은 system call을 정의하여 많은 stack 양이 필요한 프로그램도 잘 수행할 수 있도록 한다.

1-2. Implemetation

기존 exec코드에서 stack 및 guard 페이지를 할당 받는 부분은 다음과 같다.

```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

- exec는 stack 및 guard 페이지를 할당을 받고 `clearpteu` 함수를 통해 guard용 페이지를 접근이 불가능하게 pgdir를 지정한다.

exec2는 여러 갯수의 stack용 page를 할당받는 함수이므로 총 page 갯수는 `stacksize + 1` 이다.

따라서 exec를 exec2에 맞게 수정하면 다음과 같다.

```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
// stack size + 1 만큼의 PAGE를 할당 받는다. (가드용 페이지 1개 + stacksize 갯수의 PAGE)
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + (stacksize+1)*PGSIZE)) == 0)
  goto bad;
// user의 접근을 막는 gaurd용 page로 설정한다.
clearpteu(pgdir, (char*)(sz - (stacksize+1)*PGSIZE));
sp = sz;
```

- 추가적으로 thread에 맞게 수정한 부분은 LWP에서 더 자세히 다루겠다.

1-3. Exception Handling

```
if (stacksize <= 0 || stacksize > 100){
  cprintf("exec2: illegal stack size\n");
  return -1;
}
```

- stacksize가 범위에 맞지 않는 경우 예외처리를 위의 코드와 같이 구현하여 1이상 100이하의 stacksize를 올바르게 넣었을 경우에만 작동하도록 예외처리를 하였다.

2. Process memory limitation

2-1. setmemorylimit

```
int setmemorylimit(int pid, int limit);
```

- pid에 해당하는 process의 memory limit을 limit으로 설정하는 함수이다. 이 함수를 통해서 유저 프로세스는 특정 기준 이상의 공간을 더 할당 받지 못하도록 한다.
- 처음 생성 된 프로세스는 0으로 limit이 설정되어 있으며 생성 된 이후 이러한 system call을 이용해서 memory에 제한을 걸 수 있다.
- limit 0인경우 제한이 없다는 것을 의미한다. (초기값)

2-2. Implementation

- 구현을 위해서 `struct proc` 에 하나의 변수를 추가하였다.

```
// in proc.h
struct proc {
...
  int memlim;
...
};
```

- memlim 변수는 `setmemorylimit` system call에 의해서만 변경되는 값으로 초기에는 제한이 없다는 뜻인 0을 가지고 있다.

- 해당 프로세스가 가질 수 있는 memory size를 byte단위로 정보를 담고있다.

```
// memorylimit system call
int
setmemorylimit(int pid, int limit)
{
  // Set return value -1
  int ret = -1;
  struct proc *p;

  // If limit argument is less than 0
  // Return -1
  if (limit < 0)
  {
    return ret;
  }

  // Acquire ptable lock
  acquire(&ptable.lock);

  // Set limit of process corresponding to pid second argument named limit
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->pid == pid && p->sz <= limit)
    {
      // Set limit and ret
      p->memlim = limit;
      ret = 0;
      break;
    }

  // Release ptable lock
  release(&ptable.lock);

  // Return ret
  return ret;
}
```

- ptable을 순회하면서 pid에 해당하는 process의 memlim을 limit argument로 지정해준다.
- 성공하면 0을 아니면 -1을 리턴한다.

2-3. Exception Handling

1. 만약 호출시 limit argument가 음수면 바로 -1을 리턴한다.
2. ptable을 순회시 pid에 해당하는 프로세스가 존재하지 않을 경우에는 -1을 리턴한다.
3. pid에 해당하는 process가 이미 limit 초과의 공간을 사용중이라면 -1을 리턴한다.
    - process가 사용하고 있는 메모리는 sz라는 멤버 변수에서 정보를 관리하고 있기 때문에 limit과 비교하여 예외처리를 구현하였다.

2-4. thread_create, growproc

- exec를 제외하고 allocuvm을 호출하는 곳을 찾아보았더니 thread_create, growproc
- memory limit이 설정된 뒤에도 sz를 변경할 수 있기 때문에 sz를 늘리는 곳에서는 process의 memory limit을 체크해야한다.
- `growproc`

```
// in growproc
if(curproc->memlim != 0 && curproc->memlim < sz + n)
  return -1;
```

- growproc에서 현재 process의 memory limit이 0인지 체크하고 확장할 크기와 process의 memory limit을 비교하고 확장할 수 없으면 -1을 리턴한다.

- `thread_create`

```
// in thread_create
if (curproc->_ustack[t - curproc->ttable] == 0) {
  sz = PGROUNDUP(curproc->sz);

  if (curproc->memlim != 0 && curproc->memlim < sz + 2 * PGSIZE)
    goto bad;

  if ((sz = allocuvm(curproc->pgdir, sz, sz + 2 * PGSIZE)) == 0)
    goto bad;
  clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));
  curproc->_ustack[t - curproc->ttable] = sz;
  curproc->sz = sz;

  // stack용 페이지 수를 2개 늘린다.
  curproc->ssize = curproc->ssize + 2;
}
```

- allocuvm을 수행하기 전 growproc과 같이 memory limit을 체크한다.

3. Process manager

3-1. pmanager

- 현재 실행중인 프로세스들의 정보를 확인하고 관리할 수 있는 유저프로그램이며 종료 커맨드가 들어오기 전까지 한줄로 명령어를 받으며 해당 명령어에 대한 기능을 작동시킨다.
- 입력 받는 곳은 기존 sh와 차이를 주기 위해 `[pmanager] >>`로 커맨드 입력창이 나오도록 하였다.
- 다음과 같은 기능을 제공한다.
  1. list : 현재 실행중인 프로세스의 정보를 출력한다.
  2. kill <pid> : 특정 pid의 프로세스를 kill system call을 이용하여 처리한다.
  3. execute <path> <stacksize> : exec2 system call을 이용하여 path에 있는 명령어를 stacksize만큼의 userstack으로 만든다.
  4. memlim <pid> <limit> : pid에 해당하는 process의 memory limit을 memlim으로 설정한다.
  5. exit : pmanager를 종료한다.
  - (+) help : 따로 추가한 기능이다. 명령어의 목록을 확인 할 수 있다.

3-2. Implementation

- 기본적으로 `sh.c` 와 거의 유사하게 동작한다.

1. pmanager.c

- pmanager는 다음 코드로 구성되어있다.

```
printf(1, "[Process Manager]\n");
help();

// Read and run input commands.
while(getcmd(buf, sizeof(buf)) >= 0){
  // buf에서 cmd명을 분리해낸다.
  // 각 cmd명에 해당하는 함수를 출력한다.
  char *cmd = my_strtok(buf, " \n");
  // help
  if (!strcmp(cmd, "help")) {
    help();
  }
  // runnable or running 상태인 process들을 출력한다.
  else if(!strcmp(cmd, "list"))
  {
    list();
  }
  // pid에 해당하는 process를 제거한다.
  else if(!strcmp(cmd, "kill"))
  {
    int pid = atoi(my_strtok(0, " "));
    if (!kill(pid))
    {
      printf(1, "[kill] Success\n");
    }
    else
    {
      printf(1, "[kill] Fail\n");
    }
    wait();
  }
  // execute (exec2 system call 이용)
  else if(!strcmp(cmd, "execute"))
  {
    char *arg0 = my_strtok(0, " ");
    stacksize = 0;
    stacksize = atoi(my_strtok(0, " "));

    if(fork1() == 0) {
      runcmd(parsecmd(arg0));
      exit();
    }
  }
  // setmemorylimit system call
  else if(!strcmp(cmd, "memlim"))
  {
    int pid = atoi(my_strtok(0, " "));
    int limit = atoi(my_strtok(0, " "));

    if (setmemorylimit(pid, limit) != 0)
    {
      printf(1, "[memlim] Fail\n");
    }
    else
    {
      printf(1, "[memlim] Success\n");
    }
  }
  // exit인경우 loop를 탈출한다.
  else if(!strcmp(cmd, "exit"))
  {
    printf(1, "[exit] Exit\n");
    break;
  }
}

// exit
exit();
```

- 우선 getcmd를 통해서 입력을 받고 입력받은 것에 대해서 my_strtok를 이용하여 특정 delimiter를 기준으로 분리한다. (명세에 따르면 “ “)라고 한다. 그 후 각각에 명령에 맞게 type을 변경하여 system call이나 특정 기능을 수행시킨다. my_strtok는 기존 c에서 <string.h> library에 있는 `char* strtok(char* str, char* delimiters)` 를 참고하여 만들었다.

```
// in ulib.c
char*
my_strtok(char *str, const char* delimiters)
{
  static char* cur;
  char *delimiter;

  if (str != 0) cur = str;
  else str = cur;

  if (*cur == 0) return 0;

  while (*cur)
  {
    delimiter = (char *) delimiters;

    while (*delimiter)
    {
      if(*cur == *delimiter)
      {
        *cur = 0;
        ++cur;
        return str;
      }
      ++delimiter;
    }
    ++cur;
  }

  return str;
}
```

2. list

- 현재 작동중인 process의 정보를 출력하는 system call이다.
- 각 프로세스의 이름, pid, 스택용 페이지의 개수, 할당받은 메모리의 크기, 메모리의 최대 제한을 출력한다.

```
// list system call
void
list(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  cprintf("[Process Information]\n");

  // If Mainthread of process is Running or Runnable,
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (mainthread(p)->state != RUNNABLE && mainthread(p)->state != RUNNING)
      continue;
    cprintf("%s %d %d %d %d\n", p->name, p->pid, p->ssize, p->sz, p->memlim);
  }
  release(&ptable.lock);
}
```

- Process(main thread)의 상태가 running or runnable일 경우에 해당 정보를 출력한다.

- 구현 방식은 ptable을 순회하면서 main thread의 상태에 따라 출력하도록 구현했다.

3. kill

- pid에 해당하는 process를 kill system call을 이용하여 제거한다.
- 제거하고 wait를 통해서 정리해준다.
- 성공 여부를 출력한다.

```
// pid에 해당하는 process를 제거한다.
else if(!strcmp(cmd, "kill"))
{
  int pid = atoi(my_strtok(0, " "));
  if (!kill(pid))
  {
    printf(1, "[kill] Success\n");
  }
  else
  {
    printf(1, "[kill] Fail\n");
  }
  wait();
}
```

4. execute

- path 경로에 위치한 프로그램을 stacksize 개수만큼 스택용 페이지와 함께 실행하도록 한다.
- exec2를 인자와 함께 호출하도록 하였다.
- fork1을 통해서 pmanager를 복제하고 그 후 sh.c의 기존 코드를 통해서 arg0(path)의 실행 프로그램을 실행시킨다.
- stacksize는 전역 변수로 설정하여 exec2의 stacksize인자를 넣을 수 있도록 하였다.

```
else if(!strcmp(cmd, "execute"))
{
  char *arg0 = my_strtok(0, " ");
  stacksize = 0;
  stacksize = atoi(my_strtok(0, " "));

  if(fork1() == 0) {
    runcmd(parsecmd(arg0));
    exit();
  }
}
```

5. memlim

- setmemorylimit system call을 호출하여 pid를 가진 process의 메모리 제한을 limit으로 설정하도록 하였다.

```
else if(!strcmp(cmd, "memlim"))
{
  int pid = atoi(my_strtok(0, " "));
  int limit = atoi(my_strtok(0, " "));

  if (setmemorylimit(pid, limit) != 0)
  {
    printf(1, "[memlim] Fail\n");
  }
  else
  {
```

```
    printf(1, "[memlim] Success\n");
  }
}
```

6. exit

- 무한 loop를 탈출하도록 break문을 넣어서 종료하도록 하였다.

7. help

- 추가적으로 구현한 부분인데 pmanager에 어떠한 명령어를 실행할 수 있는지 출력하는 함수이다.

```
// help
// 도움말을 출력해주는 기능 (부가)
void help(void)
{
  printf(1, "****************************************** HELP ******************************************\n");
  printf(1, "* - list : print information of running & runnable process                                *\n");
  printf(1, "* - kill <pid> : kill the process corresponding to the pid                               *\n");
  printf(1, "* - execute <path> <stacksize> : execute the program located in the path with stacksize    *\n");
  printf(1, "* - memlim <pid> <limit> : Set the memlim of the process corresponding to the pid to limit *\n");
  printf(1, "* - exit : terminate pmanager program                                                    *\n");
  printf(1, "* - help : print all instruction of pmanager                                            *\n");
  printf(1, "******************************************************************************************\n");
}
```

4. LWP

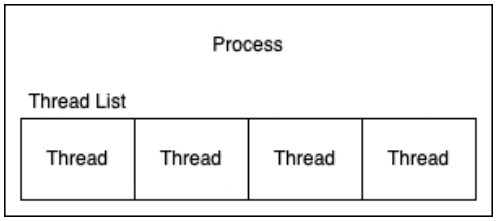
- Pthread in xv6

4-1. thread

- thread는 같은 프로세스 내에 있는 다른 thread와 자원, 주소 공간을 공유하고 multitasking을 가능하게 해주는 개념이다.
- 현재 xv6에서는 process만 존재하므로 thread를 만들어야한다
- 만들어야하는 system call은 다음과 같다.
- thread_create
- thread_exit
- thread_join

4-2. Design

- 내가 생각한 디자인은 다음과 같다.



- 하나의 process 내부에 thread들이 존재하는 방식으로 구현하였다.
- 각각의 thread는 user stack과 kernel stack이 존재하며, 기존 process또한 thread로 보았다.
- thread 내부에는 실행에 필요한 kstack, state, chan, trapframe, context, retval을 담았다.

1. thread_create
    a. 새 스레드를 생성하고 시작하는 api이다.
    b. allocproc처럼 thread list를 순회하면서 UNUSED인 곳을 찾고 거기에 필요한 정보들을 넣어서 할당한다.
2. thread_exit
    a. 스레드를 종료하고 값을 반환하는 api이다.
    b. 해당 state를 ZOMBIE로 만들고 해당 thread의 retval에 argument를 넣어서 마무리한다.
3. thread_join
    a. 스레드의 종료를 기다리고 thread_exit을 통해 반환한 값을 반환한다.
    b. wait와 유사하게 작성하였으며 해당 스레드가 종료될 때까지 loop를 돌린다. 해당 thread가 종료되었으면 kfree를 통해 반환한다.
    c. user stack은 나중에 공간 재사용을 위해서 반환하지 않는다.

4-3. Implementation

4-3-1. structure

- 우선 proc, thread 구조체를 다음과 같이 수정하였다.
- `struct thread`

```
// Thread
struct thread {
  char *kstack;                // Bottom of kernel stack for this thread
  enum procstate state;        // Process state
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan

  thread_t tid;                // Thread id
  void *retval;                // return value of thread
};
```

- 실행에 필요한 것들 그리고 thread에서 쓰이는 변수들을 담았다.
- `struct proc`

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process

  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
```

```
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  // Added
  int memlim;                  // If zero, memory size is unlimit.
  int ssize;                   // Stack용 페이지 갯수

  // Thread
  int rectidx;                // 가장 최근에 접근했던 thread의 index (for scheduler)
  int mainidx;                // main thread의 index
  int nextidx;                // 다음에 스케줄될 thread의 index
  struct thread ttable[NPROC];     // thread list
  uint _ustack[NPROC];             // user stack for thread
};
```

- 나의 디자인은 process도 main thread에 넣었으므로 실행에 필요한 기능을 제외하고 process에 나두었다.
- `ssize`는 스택용 페이지 수를 담는 변수이다.
- `rectidx` : 가장 최근에 접근했던 thread의 index를 담고 있다.
- `mainidx` : 메인 thread의 index번호이다.
- `nextidx` : 스케줄러에서 다음으로 탐색을 시작할 thread의 index이다.
- `_ustack`은 thread들의 sz들을 담고 있는 배열이다.
- `thread_t`: pid와 같이 int이며 `types.h`에 다음과 같이 선언하였다.

```
// in types.h
typedef int thread_t;
```

4-3-2. mainthread, mythread

```
// Return Main thread of p
// Return 0th thread of thread list of process
// process의 main thread의 주소를 return한다.
struct thread*
mainthread(struct proc* p) {
  return &p->ttable[p->mainidx];
}

// Return current thread of p like myproc function
// process의 작동중인 thread의 주소를 return한다.
struct thread*
mythread(struct proc* p) {
  return &p->ttable[p->rectidx];
}
```

- mainthread와 최근에 사용되었던 thread는 자주 접근하기 때문에 만들었다.
- mainthread는 thread list에서 main thread를 return 하는 함수이다.
- mythread는 thread list에서 가장 최근에 작동하였던 thread를 return하는 함수이다.

4-3-3. thread_create

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

```
found:
  t->tid = nexttid++;
  t->state = EMBRYO;

  // Allocate kernel stack.
  if((t->kstack = kalloc()) == 0)
    goto bad;

  // stack pointer를 우선 설정한다.
  sp = t->kstack + KSTACKSIZE;

  // stack pointer를 내리면서 작동에 필요한 구성요소들을 넣는다.
  // Leave room for trap frame.
  sp -= sizeof *curthread->tf;
  t->tf = (struct trapframe*)sp;
  *t->tf = *curthread->tf;

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

  // context 설정 부분
  sp -= sizeof *t->context;
  t->context = (struct context*)sp;
  memset(t->context, 0, sizeof *t->context);
  t->context->eip = (uint)forkret;

  // from exec
  // 기존에 할당 받은 user stack이 아닌경우 user stack을 할당받고 설정한다.
  if (curproc->_ustack[t - curproc->ttable] == 0) {
    sz = PGROUNDUP(curproc->sz);

    if (curproc->memlim != 0 && curproc->memlim < sz + 2 * PGSIZE)
      goto bad;

    if ((sz = allocuvm(curproc->pgdir, sz, sz + 2 * PGSIZE)) == 0)
      goto bad;
    clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));
    curproc->_ustack[t - curproc->ttable] = sz;
    curproc->sz = sz;

    // stack용 페이지 수를 2개 늘린다.
    curproc->ssize = curproc->ssize + 2;
  }

  // thread의 argument와 fake pc값을 넣어준다.
  sp = (char *)curproc->_ustack[t - curproc->ttable];
  sp -= 4;
  *(uint*)sp = (uint) arg;
  sp -= 4;
  *(uint*)sp = (uint) 0xffffffff;

  // Commit to the user image.
  // trap frame의 instruction pointer와 stack pointer를 설정해준다.
  t->tf->eip = (uint)start_routine;  // thread가 작동할 함수의 주소값
  t->tf->esp = (uint)sp;

  // 생성된 thread의 state를 RUNNABLE로 만들어준다.
  t->state = RUNNABLE;

  // thread 인자에 새로 생긴 thread의 tid(thread id)를 넣어준다.
  *thread = t->tid;
  release(&ptable.lock);
  return 0;
  ...
}
```

- 기본 적으로 allocproc, fork, exec의 구조를 보고 공부하여서 구성을 하였다. 절차는 다음과 같다.

1. 우선 thread list(ttable)을 순회하면서 UNUSED state를 탐색한다.
2. 그 후 kernel stack을 kalloc을 이용해서 할당 받는다.
3. stack pointer를 KSTACKSIZE만큼 늘리고 stack pointer에 필요한 정보를 넣어준다.
4. UNUSEDE된 thread의 userstack이 할당이 되어있지 않다면 늘리고 그렇지 않다면 기존 공간을 재활용한다.
5. 그 후 userstack에 필요한 정보인 argument와 fake pc값을 넣어준다.
6. 실제 thread가 시작될 주소와 stack pointer를 위에서 만든 stack pointer를 넣어주고 상태를 변화시킨다.
7. 새로 생성된 thread id를 인자에 전달해준다.

- 실행시 필요한 trapframe, stackpointer등을 미리 설정한다. 만약 user stack이 할당된 영역에 존재하면 즉 0이 아닐경우 그 부분을 재사용을 한다.

4-3-4. thread_exit

```
void
thread_exit(void *retval){
  struct proc *p = myproc();
  struct thread *t = mythread(p);

  acquire(&ptable.lock);

  // ZOMBIE와 retval을 설정해준다.

  // 현재 thread의 state를 ZOMBIE로 설정한다.
  t->state = ZOMBIE;
  // thread에 retval를 정보를 넣어준다.
  t->retval = retval;
  wakeup1((void *)t->tid);

  sched();
  panic("zombie exit");
}
```

- 현재 thread를 종료하는 system call로 현재 thread의 상태를 바꾸고 현재 thread를 join하며 기다리고 있는 thread를 깨워준다.
- join시에 tid를 이용해서 재우고 깨우는 방향으로 구현하였다.
- thread의 상태를 변경하고 thread 구조체 내부에 retval을 넣어주는 것으로 구현하였다.

4-3-5. thread_join

```
// thread join
int
thread_join(thread_t thread, void **retval) {
  struct thread *t;
  int havekids;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(t = curproc->ttable; t < &curproc->ttable[NPROC]; t++){
      if(t->tid != thread)
        continue;
      havekids = 1;
```

```
      if(t->state == ZOMBIE){
        if (retval != 0)
          *retval = t->retval;
        kfree(t->kstack);
        t->kstack = 0;
        t->state = UNUSED;
        t->tid = 0;
        release(&ptable.lock);
        return 0;
      }
    }

    // No point waiting if we don't have any children.
    // 자식이 없거나 현재 process가 kill count가 올라가 있으면 종료한다.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

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

- loop를 돌면서 해당 thread가 존재할 때까지 기다린다. sleep을 통해서 기다려야할 thread의 thread id의 channel가지고 sleep 하면서 기다린다.
- 만약 해당 thread가 없거나 현재 프로세스의 kill이 올라가있으면 -1을 리턴한다.
- 만약 해당 thread가 살아있으면 그 thread id를 channel로 설정하여 thread_join을 호출시킨 thread를 재운다.

4-4. Update the existing xv6 code

- proc 구조체에는 이제 실행관련한 멤버 변수들이 없고 thread array가 존재한다. 그리고 thread array 내부에 실행 관련한 정보들이 들어있으므로 기존 xv6에서의 process기반이였던 함수들을 thread를 기준으로 수정해야한다.

4-4-1. allocproc

- process의 구조체가 바뀌었으니 수정해야한다.

```
// in allocproc
t = mainthread(p);

p->state = EMBRYO;
p->pid = nextpid++;
p->memlim = 0;
p->ssize = 0;
p->rectidx = 0;
p->nextidx = 0;
p->mainidx = 0;
memset(p->_ustack, 0, sizeof(uint) * NPROC);

t->tid = nexttid++;
t->state = EMBRYO;

// Allocate kernel stack.
if((t->kstack = kalloc()) == 0){
  t->state = UNUSED;
  p->state = UNUSED;
  release(&ptable.lock);
  return 0;
}
```

- UNUSED 상태인 process를 찾았을 때 메인 스레드에 접근하여 할당하는 함수이다.
- rectidx, nextidx, mainidx를 0으로 초기화 하고 user stack 또한 초기화 해준다.

4-4-2. userinit

```
// Set up first user process.
void
userinit(void)
{
  struct proc *p;
  struct thread *t;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  t = mainthread(p);
  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;
  p->_ustack[t-p->ttable] = PGSIZE;
  memset(t->tf, 0, sizeof(*t->tf));
  t->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  t->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  t->tf->es = t->tf->ds;
  t->tf->ss = t->tf->ds;
  t->tf->eflags = FL_IF;
  t->tf->esp = PGSIZE;
  t->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;
  t->state = RUNNABLE;

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

- thread의 trapframe으로 thread의 state도 설정하도록 변경하였다.

4-4-3. fork

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();
  struct thread *nt;
  struct thread *curthread = mythread(curproc);

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }
```

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

  // copy memory limit variable
  np->memlim = curproc->memlim;

  // Clear %eax so that fork returns 0 in the child.
  nt->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

  safestrcpy(np->name, curproc->name, sizeof(curproc->name));

  pid = np->pid;

  acquire(&ptable.lock);

  // copy ustack
  for(int i = 1, j = 0; i < NPROC && j < NPROC;) {
    if (j == curproc->rectidx) {
      j++;
      continue;
    }
    np->_ustack[i] = curproc->_ustack[j];
    i++;
    j++;
  }
  np->_ustack[0] = curproc->_ustack[curproc->rectidx];

  np->state = RUNNABLE;
  nt->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

- thread와 관련된 코드로 바꾸었고 호출 시킨 현재 thread가 새로 만든 process의 mainthread가 되도록 변경한다.
- memory limit도 복제한다.

4-4-4. exit

```
for(t = curproc->ttable; t < &curproc->ttable[NPROC]; t++){
  if (t->state != UNUSED) {
    t->state = ZOMBIE;
  }
}
```

- exit을 호출시 thread들도 ZOMBIE state로 변경한다.

4-4-5. wait

```
if(p->state == ZOMBIE){
  // wait하면서 thread를 정리해준다.
  for(t = p->ttable; t < &p->ttable[NPROC]; t++){
    if (t->state == UNUSED) continue;
    p->_ustack[t-p->ttable] = 0;
    kfree(t->kstack);
    t->kstack = 0;
    t->state = UNUSED;
    t->tid = 0;
  }
  // Found one.
  pid = p->pid;
  freevm(p->pgdir);
  p->pid = 0;
  p->parent = 0;
  p->name[0] = 0;
  p->killed = 0;
  p->state = UNUSED;
  p->rectidx = 0;
  p->mainidx = 0;
  p->nextidx = 0;
  p->ssize = 0;
  p->memlim = 0;
  release(&ptable.lock);
  return pid;
}
```

- wait시 자식 프로세스의 thread들과 자식 프로세스를 정리한다.

4-4-6. scheduler

```
void
scheduler(void)
{
  struct proc *p;
  struct thread *t = 0;
  struct cpu *c = mycpu();
  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(p->state != RUNNABLE)
        continue;
      t = &p->ttable[p->nextidx];
      for(int i = 0; i < NPROC; i++, t++){
        if (t == &p->ttable[NPROC])
          t = p->ttable;
        if (t->state == RUNNABLE)
          break;
      }
      if (t->state != RUNNABLE)
        continue;
      // 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;
      p->rectidx = t - p->ttable;
      p->nextidx = p->rectidx + 1;
      if (p->nextidx == NPROC) p->nextidx = 0;
      switchuvm(p);
```

```
      t->state = RUNNING;

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

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
      t = 0;
    }
    release(&ptable.lock);

  }
}
```

- 2중 loop문을 이용하여 runnable state인 thread를 찾아서 수행하는 방향으로 구현하였다.
- runnable한 process를 찾고 그 안에서 nextidx의 thread부터 runnable한 thread를 찾아서 실행시킨다.
- rectidx를 runnable한 thread로 설정하고 nextidx를 rectidx + 1로 설정한다.

4-4-7. sched

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  ...
  if(mythread(p)->state == RUNNING)
    panic("sched running");
  ...
  swtch(&mythread(p)->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

- 가장 최근에 수행된 thread를 찾는 mythread를 이용한 코드로 수정하였다.
- 실행단위가 이제 thread단위 이므로 다음과 같이 수정하였다.

4-4-8. yield

```
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  mythread(myproc())->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

- 현재 thread 및 process의 상태를 runnable로 state를 바꾼다.

4-4-9. sleep

```
void
sleep(void *chan, struct spinlock *lk)
{
```

```
  struct proc *p = myproc();
  struct thread *t;
  ...
  // Go to sleep.
  t = mythread(p);
  t->chan = chan;
  t->state = SLEEPING;
  sched();

  // Tidy up.
  t->chan = 0;

  ...
}
```

4-4-10. wakeup1

```
static void
wakeup1(void *chan)
{
  struct proc *p;
  struct thread *t;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == RUNNABLE || p->state == SLEEPING)
      for(t = p->ttable; t < &p->ttable[NPROC]; t++)
        if(t->state == SLEEPING && t->chan == chan)
          t->state = RUNNABLE;
}
```

- 2중 loop문으로 thread의 chan을 검사하도록 변경하였다.

4-4-11. kill

```
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      for(struct thread *t = p->ttable; t < &p->ttable[NPROC]; t++) {
        if (t->state == SLEEPING)
          t->state = RUNNABLE;
      }
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

- 기존 코드에서 thread table을 순회하면서 자고 있는 thread들을 깨우는 코드로 변경하였다.

4-4-12. switchuvm

```
// in vm.c
void switchuvm(struct proc *p)
{
  ...
  if(mythread(p)->kstack == 0)
    panic("switchuvm: no kstack");
  ...
}
```

- process에 kstack이 없으므로 현재 thread의 kstack을 검사하도록 수정하였다.

4-4-13. exec / exec2

```
...
curproc->mainidx = curproc->rectidx;
curproc->_ustack[curproc->rectidx] = sz;

for(i = 0; i < NPROC; i++){
  if (i == curproc->mainidx) continue;
  t = &curproc->ttable[i];
  if (t->state != UNUSED)
    kfree(t->kstack);
  t->kstack = 0;
  t->tid = 0;
  t->retval = 0;
  t->state = UNUSED;
  curproc->_ustack[i] = 0;
}


curproc->memlim = 0;
curproc->ssize = 2; // curproc->ssize = stacksize + 1 in exec2
curproc->nextidx = curproc->rectidx;
mainthread(curproc)->tf->eip = elf.entry;  // main
mainthread(curproc)->tf->esp = sp;
...
```

- memlim은 0으로 초기화 해주고 rectidx에 해당하는 thread만 실행시킬 program으로 수정하고 나머지는 지워버린다.
- 그리고 mainidx, nextidx도 또한 현재 thread의 index로 변경 시켜준다.
- 그 후 trapframe의 eip, esp를 새로운 sp와 새로 실행될 program의 것으로 교체를 해준다.

4-4-14. trap

```
if(myproc() && mythread(myproc())->state == RUNNING &&
     tf->trapno == T_IRQ0+IRQ_TIMER)
  yield();
```

- 역시 thread가 실행단위이므로 현재 프로세스의 현재 스레드를 기준으로 코드를 수정하였다.

4-4. Exception Handling

- thread_join에서 retval이 0을 넣을 수도 있겠다는 생각을 해서 retval이 0이 아닐 때만 값을 넣도록 하였다.

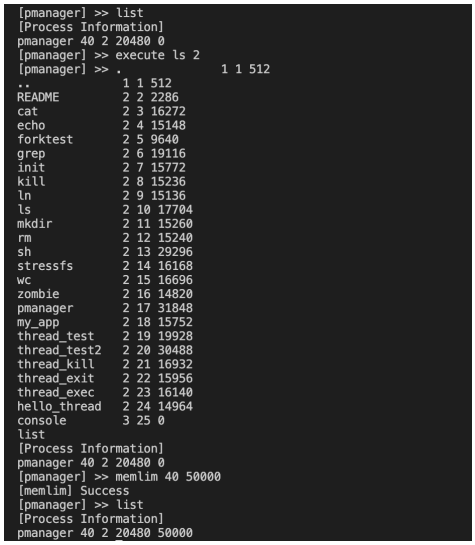
```
// thread_join in proc.c
if (retval != 0)
```

```
    *retval = t->retval;
```

5. Result

5-1. pmanager test

- pmanager에서 간단하게 테스트를 돌려보았다.
- list, memlim, execute, exit을 돌려보았으며 예상 결과대로 나오는 것을 확인하였다.
- ls도 또한 정상적으로 작동하였음을 확인하였다.



5-2. thread test

- thread (LWP)가 잘 작동하는 지 확인하는 test코드를 돌려보았다.

5-2-1. thread_test

- fork, sbrk, sleep등이 잘 작동하는지 확인하는 테스트 코드이다.
- malloc을 통해 sbrk가 잘 작동하였음을 fork를 통해서 Child가 잘 생성 되었음을 확인하였다.

```
$ thread_test
Test 1: Basic test
ThrThread 1 start
ead 0 start
Thread 0 end
Parent waiting for children...
Thread 1 end
Test 1 passed

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

Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 sThread 3 start
Thread 4 start
tart
Test 3 passed

All tests passed!
```

5-2-2. thread_exit

- printf(1, "This code shouldn't be executed!!\n"); 와 같은 출력이 나오지 않고 exit을 하였을 때 process가 종료되는 것을 확인하였고 올바르게 작동하였다는 것을 확인 할 수 있었다.

```
$ thread_exit
Thread exit test start
ThreaThread 1 start
Thread 2 Thread 3 start
Thread 4 start
d 0 start
start
Exiting...
```

5-2-3. thread_exec

- exec의 경우 나머지 스레드들을 잘 처리했는지 확인하는 테스트 코드이다.
- Hello, thread가 하나만 나옴으로써 exec가 정상적으로 작동하였음을 확인 하였다.

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Hello, thread!
```

5-2-4. thread_kill

- This code should be executed 5 times.가 5번 나오고
- This code shouldn't be executed!!가 출력되지 않아서 정상적으로 작동함을 확인하였다.

```
$ thread_kill
Thread kill test start
Killing process 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.
38
This code should be executed 5 times.
Kill test finished
```

6. Trouble shooting

6-1. thread_t tid

- 처음 디자인의 process는 `thread_t nexttid` 가 존재하였다. 다른 프로세스에는 같은 thread id가 들어갈 수 있기 때문에 그렇게 설정하였는데, `thread_join`에서 서로 다른 process에 같은 tid가 존재하면 `sleep` 및 `wakeup1` 시에 문제가 생겼다. 그래서 `nextpid` 와 같이 proc.c에서 전역 변수로 nexttid를 만들게 되었다.

```
thread_t nexttid = 1; // nexttid like nextpid in proc.c
```

6-2. exec

- 처음에는 thread list의 0번만 mainthread로 사용하려고 하였으나 kstack과 같은 문제가 발생하였다. 그래서 main thread의 index를 옮길 수 있도록 설정하였고 mainidx라는 변수를 process 내부에 집어넣게 되었다.

6-3. kstack

- 이번 프로젝트를 수행하면서 가장 많이 보았던 trap이다. kstack을 잘못지우거나 이미 존재하지 않은 kstack을 지우려고 하면 이러한 문제들이 생기는데 우선 필요한지 필요 없는 kstack인지 구분을 하고 만약 필요없는 kstack이라면 다음 코드를 이용해서 지웠다.

```
if (t->kstack != 0)
{
  kfree(t->kstack);
}
```

- 이렇게 되면 kstack의 값이 존재하는 경우에만 지울 수가 있었다.

6-4. proc

- 우선 새로운 프로세스를 만들기 위해서는 ptable에서 UNUSED한 공간을 찾고 그 후에 ttable을 이용하여 새로운 process를 만든다. 이를 위해서는 process에도 또한 state가 구현적으로나 논리적(process의 상태를 확인하기 위해서)으로도 필요하게 되어서 process의 state는 나두게 되었다. 즉, proc 및 thread 모두 state를 가지게 되었다.