201920871 윤상훈

proc.c

clone

```
int                                          //clone, join 구현부
clone(void *fcn, void *arg1, void *arg2, void *stack){

  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();
  void *nfake, *narg1, *narg2;

  if((np = allocproc()) == 0){  //현재 프로세스를 할당하고
    return -1;                  //비어있으면 -1 반환
  }

  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;
  np->pgdir = curproc->pgdir; //현재 프로세스 정보 복사해주고

  narg2 = stack - PGSIZE;
  narg1 = narg2 + sizeof(void *);
  nfake = narg1 + sizeof(void *);   //스택 포인터 위치를 지정해주고

  *(uint*)narg2 = (uint)arg2;       //스택에 저장
  *(uint*)narg1 = (uint)arg1;
  *(uint*)nfake = 0xffffffff;

  np->tf->esp = (uint)stack - PGSIZE; //레지스터 위치 설정
  np->tf->ebp = (uint)stack - PGSIZE + 3 * sizeof(void *);
  np->stack = stack;  //스택의 시작 위치를 넣어줌
  np->tf->eax = 0;
  np->tf->eip = (uint) fcn;

  for(i = 0; i < NOFILE; i++) //자식 프로세스에 현재 프로세스 복사
    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;    //새로운 프로세스 상태 runnable로 갱신
  acquire(&ptable.lock);
  np->state = RUNNABLE;
  release(&ptable.lock);

  return pid;
}
```

join

```
int
join(void **stack){
    struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    havekids = 0;

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {  //페이지 테이블을 순회하며
      if(p->parent != curproc || p->pgdir != p->parent->pgdir)  //자식 프로세스 확인
        continue;

      havekids = 1;   //자식 프로세스이고 zombie 상태이면
      if(p->state == ZOMBIE){

        pid = p->pid;
        kfree(p->kstack); //자식 프로세스 초기화 과정
        p->kstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        *stack = p->stack;
        p->stack = 0;

        release(&ptable.lock);
        return pid;
      }
    }

    if(!havekids || curproc->killed){ //자식이 없거나 현재 프로세스가 종료되었다면
      release(&ptable.lock);
      return -1;  //lock 해제하고 -1반환
    }

    sleep(curproc, &ptable.lock); //아니라면 자식이 종료까지 기다림
  }
}
```

clone은 fork 함수 join은 wait 함수와 동작이 비슷하므로 두 함수를 기반으로 수정하였다.
clone의 경우 fork에서 단순히 copyuvm을 사용해 새로운 프로세스를 복사하는 것이 아닌 같은 프로세스 안에 새로운 스택을 만드는 방식으로 수정하였다. 주어진 과제 자료에서 clone 함수의 스택 구조에 따라 스택의 초기값들의 위치를 지정해주고 대입한다. 이후 스택 레지스터의 위치 또한 적절하게 수정해주고 복사한 후 프로세스를 runnable 상태로 갱신하며 pid를 반환하고 종료한다.

join의 경우 wait 함수와 매우 유사하지만 stack의 초기화 과정을 추가하였다. 페이지 테이블을 순회하며 자식 프로세스를 확인하고, 자식프로세스이며 zombie 상태라면 자식 프로세스를 초기화 하는 과정을 거친다. 이때, *stack = p->stack을 통해 종료된 자식 프로세스의 스택 포인터를 전달하여 join을 하고있는 프로세스가 종료 상태를 알 수 있도록 한다. 이후 자식이 없거나 현재 프로세스가 종료되었다면 lock을 해제하고 -1을 반환하며 아니라면 자식이 종료될 때까지 기다린다. 위 과정을 위해 proc.h 파일에 proc 구조체에서 void *stack을 추가하였다.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  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)
  void *stack;
};
```

위 두 함수는 systemcall이다. 따라서 systemcall에 추가하기 위한 과정을 진행하였다.

sysproc.c

```
10  int
11  sys_clone(void)
12  {
13    void *fcn, *arg1, *arg2, *stack;
14    if(argptr(0, (void*)&fcn, sizeof(void*)) < 0 ||
15       argptr(1, (void*)&arg1, sizeof(void*)) < 0 ||
16       argptr(2, (void*)&arg2, sizeof(void*)) < 0 ||
17       argptr(3, (void*)&stack, sizeof(void*)) < 0)
18      return -1;
19
20    return clone((void *)fcn, (void *)arg1, (void *)arg2, (void *)stack);
21  }
22
23  int
24  sys_join(void)
25  {
26    void *stack;
27    if (argptr(0, (void*)&stack, sizeof(void *)) < 0)
28      return -1;
29
30    return join(stack);
31  }
```

사용자 프로그램에서 호출될 때, 인자들을 추출하고 유효성을 확인한다. 잘못되었다면 -1을 반환하고 그렇지 않다면 각 함수를 호출한다.

defs.h

```
105  // proc.c
106  int             cpuid(void);
107  void            exit(void);
108  int             fork(void);
109  int             growproc(int);
110  int             kill(int);
111  struct cpu*     mycpu(void);
112  struct proc*    myproc();
113  void            pinit(void);
114  void            procdump(void);
115  void            scheduler(void) __attribute__((noreturn));
116  void            sched(void);
117  void            setproc(struct proc*);
118  void            sleep(void*, struct spinlock*);
119  void            userinit(void);
120  int             wait(void);
121  void            wakeup(void*);
122  void            yield(void);
123  int             clone(void* , void*, void*, void*);   //syscall 함수에 대한 구현부
124  int             join(void**);
```

proc.c 위치에 구현되어있음으로 proc.c의 함수를 모아둔 위치에 추가해준다.

syscall.c

```
105  extern int sys_uptime(void);
106  extern int sys_clone(void);
107  extern int sys_join(void);
```

```
130  [SYS_close]   sys_close,
131  [SYS_clone]   sys_clone,
132  [SYS_join]    sys_join,
```

그리고 운영체제에서 동작하는 systemcall 함수의 동작을 위해 syscall.c 파일에 두 함수를 추가하였다.

syscall.h / user.h / usys.S

syscall.h

```
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_clone  22
#define SYS_join   23
```

user.h / usys.S

```
int clone(void *fcn, void *arg1, void *arg2, void *stack);
int join(void **stack);
```

```
SYSCALL(uptime)
SYSCALL(clone)
SYSCALL(join)
```

마지막으로 syscall.h에 매크로를 지정해주고 user.h와 usys.S 파일에 추가하여 사용자 레벨에서의 함수를 선언해주며 커널 동작을 할 수 있도록 한다.

다음은 thread_create, thread_join 외 3가지 함수에 대한 구현이다.

ulib.c

```
int
thread_create(void *fcn, void* arg1, void* arg2)
{
  void* stack = malloc(PGSIZE);
  int pid;
  //printf(1, "arg1 value in thread_create: %d\n", *(int *)arg1);
  if((pid = clone(fcn, arg1, arg2, stack)) < 0){
    return -1;
  }
  return pid;
}

int
thread_join()
{
  void * stack;
  int pid;
  if((pid = join(&stack)) < 0)
    return -1;
  free(stack);
  return pid;
}

void
lock_init(lock_t *lock)
{
  lock->flag = 0;
}

void
lock_acquire(lock_t *lock){
  //printf(1, "lock 요구");
  while(xchg(&lock->flag, 1) != 0);
}

void
lock_release(lock_t *lock){
  xchg(&lock->flag, 0);
}
```

사용자 수준 함수에 대해서 ulib.c에 구현하였다. thread_creat에서는 PGSIZE 만큼의 stack을 만들고 clone을 통해 복제한다. 이 과정을 실패했다면 -1을 반환하고 성공했다면 clone을 통해 얻은 새로운 스레드의 pid를 반환한다.

thread_join은 join을 호출하고 이를 실패시 -1을 반환하며 성공했다면 자식 스레드의 stack을 free하고 pid를 반환한다.

그 외에 lock_init는 lock flag를 초기화하고 lock_acquire는 spin을 통해 lock flag를 1로 바꿔주며 lock_release는 flag를 0으로 바꿔준다. 이때, acquire와 release는 x86.h의 xchg를 사용하여 원자성을 보장한다.

user.h

```
// ulib.c
int stat(const char*, struct stat*);
char* strcpy(char*, const char*);
void *memmove(void*, const void*, int);
char* strchr(const char*, char c);
int strcmp(const char*, const char*);
void printf(int, const char*, ...);
char* gets(char*, int max);
uint strlen(const char*);
void* memset(void*, int, uint);
void* malloc(uint);
void free(void*);
int atoi(const char*);
int thread_create(void *fcn, void * arg1, void * arg2);
int thread_join();
void lock_init(lock_t *lock);
void lock_acquire(lock_t *lock);
void lock_release(lock_t *lock);

```

마지막으로 user.h에 위 5가지 함수에 대해 ulib.c 위치에 추가해준다.

구현결과

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
2. sleep for 100 ticks
3. sleep for 100 ticks
below should be a jarbled mess:
1. sleep for 100 ticks
2. sleep for 100 ticks
3. sleep for 100 ticks
```

우선 구현 결과를 보면 lock을 사용하지 않았음에도 printf가 순차적으로 진행된 것을 확인할 수 있었다. 이는 예시로 보여준 구현 결과와 달랐으나 분명 lock을 사용하였을 때는 1, 2, 3번 과정이 순차적으로 출력되고 사용하지 않았을 때는 동시에 나오는 등 lock 동작이 수행되고 있다는 것을 알 수 있었다. 이에 따라 다음처럼 printf 문을 늘려 다시 테스트해 보았다.

test_thread

```
void f1(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  //printf(1, "f1: arg1 = %d, num = %d\n", *(int*)arg1, num);
  if (num) lock_acquire(lk);
  printf(1, "1. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "1. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "1. sleep for %d ticks\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}
```

```
void f2(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  //printf(1, "f2: arg1 = %d, num = %d\n", *(int*)arg1, num);
  if (num) lock_acquire(lk);
  printf(1, "2. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "2. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "2. sleep for %d ticks\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}
```

```
void f3(void* arg1, void* arg2) {
  int num = *(int*)arg1;
  //printf(1, "f3: arg1 = %d, num = %d\n", *(int*)arg1, num);
  if (num) lock_acquire(lk);
  printf(1, "3. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "3. sleep for %d ticks\n", SLEEP_TIME);
  printf(1, "3. sleep for %d ticks\n", SLEEP_TIME);
  sleep(SLEEP_TIME);
  if (num) lock_release(lk);
  exit();
}
```

위와 같이 수정하여 테스트한 결과는 다음과 같다.

첫 번째	두 번째	세 번째
$ test_thread below should be sequential print statements: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks below should be a jarbled mess: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks	$ test_thread below should be sequential print statements: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks below should be a jarbled mess: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks	$ test_thread below should be sequential print statements: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep for 100 ticks below should be a jarbled mess: 1. sleep for 100 ticks 1. sleep for 100 ticks 1. sleep for 12. sleep for 100 ticks 2. sleep for 100 ticks 2. sleep for 100 ticks 3. sleep for 100 ticks 3. sleep f00 ticks or 100 ticks 3. sleep for 100 ticks

위 결과를 보면 처음 두 번의 경우는 여전히 순차적으로 나왔지만 세 번째 경우 일부 printf문이 섞인 것을 확인할 수 있었다. 물론 예시 구현 결과처럼 완전히 섞인 모습은 아니지만, lock을 잡지 않았을 순서가 보장되지 않는 모습을 확인할 수 있었다.