UNIVERSIDADE DE SÃO PAULO

ESCOLA DE ARTES CIÊNCIAS E HUMANIDADES

ACH2044 - Sistemas Operacionais

Exercício programa 1
2024

Objetivo

O objetivo deste exercício programa é criar uma nova chamada de sistema (*syscall*) com a seguinte assinatura: `int getreadcount(void)`.

Esta nova *syscall* deve informar a quantidade de vezes que a chamada de sistema `read()` foi chamada (por qualquer processo) desde que o sistema operacional iniciou (valor que pode ser armazenado em um contador maroto).

É só isso 😇

Ambiente de desenvolvimento

Vamos utilizar um sistema operacional didático chamado xv6, desenvolvido no MIT para apoiar o curso de sistemas operacionais[1]. É um sistema operacional bastante simples, o que nos permitirá ver na prática alguns conceitos que estudamos em sala de aula, sem a carga cognitiva de um sistema operacional maduro. Para comparação, o xv6 tem pouco mais de 11 mil linhas de código, enquanto que o kernel Linux tem quase 30 milhões[2].

Antes de colocar as mãos na massa é importante preparar o ambiente de desenvolvimento. Vamos precisar de:

- Uma máquina com um Linux (alternativamente, você pode usar uma máquina virtual, porém provavelmente ficará um pouco lento). Vou assumir que é alguma versão do Ubuntu (se você usa outra distribuição, provavelmente vai conseguir adaptar os passos de preparação),
- Compilador gcc (foi testado que funciona com a versão 11. Também foi testado que não funciona com a versão 13. Outras versões não foram testadas, e podem ou não funcionar)
- Emulador QEMU
- Código fonte do xv6

Comece instalando o gcc na versão correta (se já não estiver instalado). Em um terminal, digite:

```
$ sudo apt install gcc-11
```

[1] https://pdos.csail.mit.edu/6.828/2017/xv6.html
[2] https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/

Se já houver uma outra versão do gcc instalada (as versões mais recente do Ubuntu utilizam o gcc-13 por padrão), será preciso configurar para usar o gcc-11:

```
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-11 1
```

Se você precisar usar voltar a usar uma outra versão do gcc, você pode selecionar com o comando:

```
$ sudo update-alternatives --config gcc
```

Em seguida, instale o QEMU:

```
$ sudo apt install qemu-system
```

Agora, vamos instalar, compilar, e rodar o xv6. Primeiro, pegue o código fonte neste link: https://github.com/mit-pdos/xv6-public/archive/refs/tags/xv6-rev11.zip, salvando o arquivo em algum diretório escolhido.

Descompacte o arquivo zip, entre no diretório criado e compile o xv6:

```
$ unzip xv6-rev11.zip
$ cd xv6-public-xv6-rev11/
$ make
```

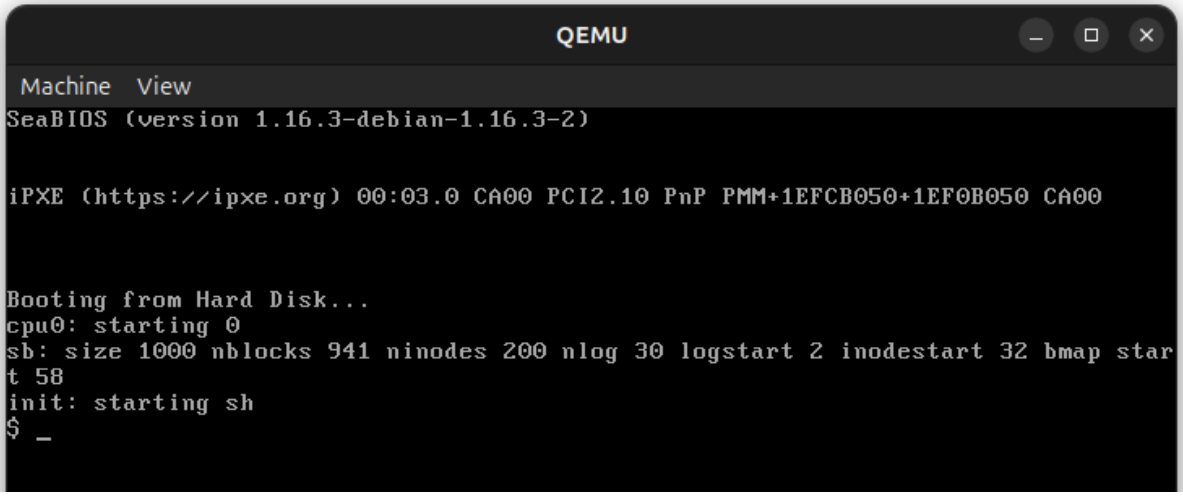
A compilação deve terminar sem erros, exibindo alguma mensagem parecida com essa:

```
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 4.0057e-05 s, 12.8 MB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
394+1 records in
394+1 records out
201780 bytes (202 kB, 197 KiB) copied, 0.000899014 s, 224 MB/s
```

Vamos agora executar o xv6 no QEMU. Você pode escolher a quantidade de CPUs que a máquina virtual possui. Por exemplo, para uma máquina com uma CPU basta digitar:

```
$ CPUS=1 make qemu
```

Deverá aparecer uma janelinha com o xv6 rodando, algo como a Figura abaixo:



Muito provavelmente será útil utilizar o gdb[3] para ajudar a depurar problemas durante a implementação do exercício. Para executar o QEMU no modo depuração, basta usar o comando (se não for informada a quantidade de CPUs, o valor padrão são duas CPUs):

```
$ make qemu-gdb
```

Em um outro terminal (uma nova aba ou janela), invoque o gdb a partir no diretório de compilação do xv6 e conecte ao servidor gdb iniciado pelo QEMU:

```
$ gdb kernel
(gdb) target remote :26000
(gdb) <insira breakpoints, etc…>
(gdb) continue
(gdb) <faça a depuração>
```

Pronto! Já temos o xv6 rodando e podemos começar a trabalhar.

Para não deixá-los à deriva com 11 mil linhas de código, vamos estudar brevemente como o xv6 lida as chamadas de sistema. Preste atenção onde as definições importantes relacionadas às chamadas de sistema são feitas, pensando que você deverá fazer algo parecido implementar a `getreadcount()`.

[3] Esta é uma ótima oportunidade de aprender a usar o gdb, se ainda não souber. Algumas referências:
https://www.ime.usp.br/~coelho/desafios/gdb.pdf
https://www.ime.usp.br/~pf/algoritmos/apend/GDB_Reference_Card.pdf
https://cseweb.ucsd.edu/classes/fa09/cse141/tutorial_gcc_gdb.html

Introdução às entranhas do xv6

Como você deve se lembrar, uma chamada de sistema é o procedimento feito por um processo (rodando em modo de usuário) para transferir o controle para o sistema operacional (modo kernel) no momento em que deseja realizar uma operação privilegiada (tal como ler dados do disco). Este procedimento faz parte do mecanismo de *Limited Direct Execution*, que permite que os processos executem diretamente no hardware (eficiente), porém controlando o que o processo pode fazer (controle).

Vamos seguir o caminho do que acontece no código do xv6[4] para entender melhor o funcionamento das chamadas de sistema. Ao iniciar uma chamada de sistema, o processo pede para que o sistema operacional execute alguma operação para ele (o SO, por sua vez, verifica se a operação é permitida). Para isso é preciso passar o controle da CPU para o sistema operacional, ao passo que o modo de execução deve ser alterado para o modo kernel.

Passando o controle para o S.O: instrução trap

Em primeiro lugar, para fazer uma chamada de sistema, precisamos estar em modo usuário executando um processo (aplicação) qualquer. A aplicação então deseja realizar uma operação privilegiada, por exemplo, usar a função `read()` para ler alguns bytes de um arquivo. Contudo, tudo o que a chamada de sistema faz é colocar alguns valores em certos registradores e então executar algum tipo de instrução de `trap`, como pode ser visto no arquivo **usys.S**[5]. Abaixo vemos a definição da função `read` (na verdade, ela é definida através de uma macro, mas estamos mostrando a macro já expandida, para facilitar).

```
.globl read;
read:
  movl $5, %eax;
  int $64;
  ret
```

Vamos decifrar este código assembly! Há apenas três instruções:

1. `movl $5, %eax;` move o valor constante 5 no registrador `%eax`
2. `int $64;` instrução de trap (que na arquitetura x86 é chamada de int, abreviação de interrupt, enquanto que 64 é o valor convencionado para *system calls* no xv6)
3. `ret` que indica o retorno da função

O valor 6 que foi colocado no registrador `%eax` será utilizado pelo sistema operacional para determinar qual é a função de tratamento adequada para a chamada de sistema (através de uma tabela). Outros argumentos da função são colocados na pilha de execução do processo (*stack*).

[4] Veremos algumas convenções que são específicas do xv6 e outras especificidades que são relacionadas à arquitetura x86

[5] Arquivos com extensão S, em geral, denotam arquivos fontes em assembly

Passando para o kernel

Ao executar a instrução `int`, a CPU entra em ação e faz algumas coisas importantes. Uma delas é mudar o modo de execução para o modo kernel, aumentando o privilégio de execução. Na arquitetura x86, o que geralmente acontece é alterar o CPL (Current Privilege Level) de 3 (modo em que as aplicações de usuário usam) para 0 (modo usado pelo kernel). Existe a possibilidade de usar níveis de privilégio intermediários, mas em geral não são muito usados.

Outra coisa importante é transferir o controle para o **vetor de interrupção** (trap vector) do sistema. O sistema operacional informa à CPU, durante a inicialização, qual é a localização do código a ser executado quando ocorre uma interrupção. Isto é feito na função `mainc` do arquivo `main.c`:

```
int
mainc(void)
{
    ...
    tvinit(); // inicialização dos vetores de interrupção
    ...
}
```

A função `tvinit()`, definida em trap.c, é a que faz a maior parte do trabalho pesado:

```
void tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

  // esta é a linha que nos interessa...
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

A macro `SETGATE()` é o ponto principal da função. Esta macro é usada para modificar o array **`idt`**, de forma que ele aponte para as funções adequadas que serão executadas após acontecer uma interrupção. Para chamadas de sistema, o que nos interessa é o `SETGATE` após o laço `for`. Veja abaixo o código desta macro, bem como a estrutura em cima da qual ela trabalha (definidos no arquivo `mmu.h`) :

```
// Gate descriptors for interrupts and traps
struct gatedesc {
  uint off_15_0 : 16;   // low 16 bits of offset in segment
  uint cs : 16;         // code segment selector
  uint args : 5;        // # args, 0 for interrupt/trap gates
  uint rsv1 : 3;        // reserved(should be zero I guess)
  uint type : 4;        // type(STS_{TG,IG32,TG32})
  uint s : 1;           // must be 0 (system)
  uint dpl : 2;         // descriptor(meaning new) privilege level
  uint p : 1;           // Present
  uint off_31_16 : 16;  // high bits of offset in segment
};

// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint) (off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint) (off) >> 16;                  \
}
```

Essencialmente, tudo o que a macro `SETGATE()` faz é preencher os valores em uma estrutura de dados. O parâmetro mais importante é o parâmetro `off`, que indica diretamente à CPU onde está o código de tratamento da *trap*. No código de inicialização (dentro da função `tvinit`) o valor de `vectors[T_SYSCALL]` é passado como o parâmetro `off` (o valor de `T_SYSCALL` é constante e igual a 64). Portanto, a função que estiver indicada no array `vectors` será chamada quando ocorrer uma *syscall*. Há outros detalhes, mas não vamos nos ater a eles (você pode consultar documentação sobre a arquitetura x86 se quiser saber mais).

Até aqui, apenas preenchemos a estrutura de dados, mas ainda não informamos nada ao hardware. O hardware só é informado mais adiante na sequência de inicialização do SO. No caso específico do xv6, isso acontece na função `mpmain()` no arquivo main.c, que por sua

vez chama a função `idtinit` que fica em `trap.c`, que, por fim, chama a função `lidt()` definida em x86.h.

```
static void
mpmain(void)
{
  idtinit();
  ...

void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}

static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

Na função `lidt()` podemos ver que uma única instrução de assembly é usada para informar a CPU a localização na memória do vetor de interrupções (*interrupt descriptor table* – IDT). Note que esta função é executada em cada um dos processadores, no caso de um sistema multi-CPU. Após esta instrução ser executada (o que, obviamente, só pode ser feito no modo kernel), as chamadas de sistema feitas por usuários podem ser tratadas.

Acionando a função de tratamento da trap

Agora que a função de tratamento de *trap* foi indicada à CPU pelo SO, podemos ver o que acontece após uma aplicação rodando em modo usuário executar uma instrução `int`. Antes de mais nada, o hardware deve realizar algumas tarefas. Primeiramente, a CPU faz algumas tarefas que são difíceis ou até mesmo impossíveis do SO fazer via software: salvar os valores de registradores na pilha, como o IP (*instruction pointer*, equivalente ao PC – *program counter*), eflags (que determina certos estados de execução da CPU), ponteiro de pilha, entre outros. Podemos ver na estrutura `trapframe` (arquivo x86.h) o que é esperado que seja armazenado pelo hardware:

```
struct trapframe {
  // registers as pushed by pusha
  uint edi;
  uint esi;
  uint ebp;
  uint oesp;      // useless & ignored
  uint ebx;
  uint edx;
  uint ecx;
  uint eax;

  // rest of trap frame
  ushort es;
  ushort padding1;
  ushort ds;
  ushort padding2;
  uint trapno;

  // below here defined by x86 hardware
  uint err;
  uint eip;
  ushort cs;
  ushort padding3;
  uint eflags;

  // below here only when crossing rings, such as from user to kernel
  uint esp;
  ushort ss;
  ushort padding4;
};
```

Como pode ser visto na parte debaixo da estrutura, certos trechos do *trap frame* são preenchidos pelo hardware (até o campo *err*), enquanto que o restante deve ser preenchido pelo SO. O primeiro código executado pelo SO é a função `vector64()` encontrada no arquivo `vectors.S`.

```
.globl vector64
vector64:
  pushl $64
  jmp alltraps
```

Este código coloca o número da *trap* na pilha (preenchendo o valor do campo `trapno` do *trap frame*) e, em seguida, chama a função `alltraps()` que termina de salvar os restante do contexto no *trap frame* (definida em `trapasm.S`).

```
# vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushal

  # Set up data segments.
  movl $SEG_KDATA_SEL, %eax
  movw %ax,%ds
  movw %ax,%es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

Esta função coloca o valor de mais alguns registradores na pilha e depois usa a instrução `pushal` para colocar o restante dos registradores de propósito geral no *trap frame*. Em seguida, o SO altera o valor de alguns registradores relacionados ao gerenciamento de memória, para que possa acessar seus próprios endereços de memória. Por fim, a função de tratamento da trap (que se chama, criativamente, `trap`) é chamada.

Tratamento da trap

Agora que os detalhes de baixo nível relacionados ao preenchimento do *trap frame* foram concluídos, o código em assembly chama a função `trap()` definida do arquivo `trap.c`, que é uma função genérica em C para o tratamento de *traps*, recebendo como parâmetro um ponteiro para o *trap frame*. Como esta função é genérica e executada para todos os tipos de interrupção, é necessário determinar o que de fato deve ser feito, através do número identificador da interrupção, encontrado no campo `trapno` do *trap frame*. Primeiramente, é verificado se é uma *syscall* (comparando com a constante `T_SYSCALL`, definida no arquivo `traps.h` como 64), caso que é tratado conforme mostrado abaixo:

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(cp->killed)
      exit();
    cp->tf = tf;
    syscall();
    if(cp->killed)
      exit();
    return;
  }
  ... // continues
}
```

Este trecho de código verifica se o processo atual (ou seja o processo que fez a chamada de sistema, representado pelo ponteiro `cp`) já morreu, caso em que a função `exit` é chamada para limpar tudo relacionado ao processo, e nenhuma chamada de sistema chegará a ser executada. Em seguida a função `syscall()` é chamada para realizar de fato a chamada de sistema (mais detalhes abaixo). Por fim, é verificado se o processo morreu durante a *syscall* antes de retornar. Veremos o que acontece após o `return` mais a frente.

Encontrando a chamada de sistema no vetor

```
static int (*syscalls[])(void) = {
[SYS_chdir]   sys_chdir,
[SYS_close]   sys_close,
[SYS_dup]     sys_dup,
[SYS_exec]    sys_exec,
[SYS_exit]    sys_exit,
[SYS_fork]    sys_fork,
[SYS_fstat]   sys_fstat,
[SYS_getpid]  sys_getpid,
[SYS_kill]    sys_kill,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_mknod]   sys_mknod,
[SYS_open]    sys_open,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_unlink]  sys_unlink,
[SYS_wait]    sys_wait,
[SYS_write]   sys_write,
```

```
};

void
syscall(void)
{
  int num;

  num = cp->tf->eax;
  if(num >= 0 && num < NELEM(syscalls) && syscalls[num])
    cp->tf->eax = syscalls[num]();
  else {
    cprintf("%d %s: unknown sys call %d\n",
        cp->pid, cp->name, num);
    cp->tf->eax = -1;
  }
}
```

Uma vez que a função `syscall()` (definida no arquivo `syscall.c`) finalmente é chamada, não resta muito mais o que fazer. O número da chamada de sistema está no registrador `%eax` (lembra?), que podemos acessar através do *trap frame* e usar para determinar qual é a rotina de tratamento adequada, de acordo com o array de ponteiros de função `syscalls[]`[6]. Basicamente, todo sistema operacional usa uma tabela como esta para definir suas chamadas de sistema. Após verificar se a posição escolhida do array é válida e aponta para uma função não nula, a rotina na posição escolhida é chamada. Por exemplo, se a chamada de sistema `read()` foi chamada pelo usuário, a função `sys_read()` seria chamada neste ponto. O valor de retorno, como você pode observar, é armazenado no valor `eax` do *trap frame* (que eventualmente será copiado para o registrador `%eax`) para ser repassado ao usuário.

Retornando para o usuário

O caminho de volta é relativamente simples. Primeiro, a chamada de sistema retorna um valor inteiro que vai para o registrador `eax` (conforme vimos acima). O código retorna para a função `trap()`, que simplesmente retorna para onde foi chamada no código assembly (a partir da função `alltraps`, no arquivo `trapasm.S`).

```
# Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
```

[6] Esta inicialização usa a sintaxe de “inicializares designados”, veja mais no link abaixo https://acervolima.com/inicializadores-designados-em-c/

```
iret
```

Este código apenas remove alguns valores da pilha para permitir que o contexto do processo seja restaurado corretamente. Por fim, a instrução especial é chamada: `iret` (instrução de retorno de *trap*). Esta instrução é semelhante ao retorno de uma função qualquer, porém ela reduz o nível de privilégio para modo usuário e salta para instrução seguinte à instrução `int`, executada pelo processo ao chamar *syscall*, de forma que todo o estado anteriormente salvo no *trap frame* é restaurado.

Considerando o exemplo da função `read()`, neste ponto, o código retorna para o código da biblioteca de sistema desta função. A próxima instrução executada é uma instrução de retorno de função simples (`ret`, em vez de `iret`), que retorna para o chamador da função `read()`. Ufa, terminou!

Resumo

Nós vimos o caminho completo para execução de uma chamada de sistema, incluindo os pontos de entrada e saída do kernel. Como pudemos observar, é bem mais complicado do que uma chamada de função comum, requerendo que uma série de passos sejam seguidos cuidadosamente pelo sistema operacional (com auxílio do hardware), para garantir que o estado da aplicação de usuário seja salvo e restaurado corretamente. O conceito é simples, os detalhes, no entanto, são um pouco complicados.

Dicas

Uma boa tática para modificar algo em uma base de código grande é encontrar algo parecido com o que você quer fazer e a partir disso fazer um *copy & paste & modify*. Por exemplo, pegue uma outra chamada de sistema (como a `getpid`) e veja o que é possível aproveitar dela e modifique conforme necessário.

Você provavelmente vai gastar mais tempo entendendo o código do xv6 do que implementando a chamada de sistema nova.

Você pode (e deve) postar suas dúvidas no eDisciplinas. Porém, lembre-se de **não** publicar trechos de código modificados/criados por você diretamente.

Você provavelmente desejará testar a sua implementação da chamada de sistema. Para isso, você pode criar um programa de espaço de usuário que a utiliza. Para isso, basta adicionar o arquivo no mesmo diretório do xv6 e alterar o `Makefile` para compilar o seu novo programa. Basta adicioná-lo à lista `UPROGS` (com um underline antes e sem o .c). Por exemplo, suponha que você deseja adicionar o programa hello.c, abaixo:

```
#include "types.h"
#include "user.h"

int main(int argc, char *argv[])
{
  printf(1, "Hello World!\n");
  exit();
}
```

O `Makefile` deve ser alterado da seguinte forma:

`Makefile` antes	`Makefile` depois

(… outras coisas)
 	
`UPROGS=\`
`_cat\`
`_echo\`
`_forktest\`
`_grep\`
`_init\`
`_kill\`
`_ln\`
`_ls\`
`_mkdir\`
`_rm\`
`_sh\`
`_stressfs\`
`_usertests\`
`_wc\`
`_zombie\`
 	
(outras coisas …)
	(… outras coisas)
	
	`UPROGS=\`
	`_cat\`
	`_echo\`
	`_forktest\`
	`_grep\`
	`_init\`
	`_kill\`
	`_ln\`
	`_ls\`
	`_mkdir\`
	`_rm\`
	`_sh\`
	`_stressfs\`
	`_usertests\`
	`_wc\`
	`_zombie\`
	`_hello\`
	
	(outras coisas…)

Em seguida, basta compilar o xv6 novamente com `make`. Ao rodar o xv6 no qemu, o novo programa estará disponível para ser executado.

Note que, embora escrever programas em C para o xv6 seja bastante semelhante a escrever programas em C para o Linux, há algumas diferenças em relação à assinatura de funções e aos headers que devem ser incluídos. Você pode olhar os outros programas da lista `UPROGS` para ver (e copiar) como as coisas são feitas.

Entrega e critérios de correção

A entrega deve ser feita exclusivamente no eDisciplinas, até a data indicada. Deverá ser feito upload de dois arquivos:

- Um arquivo zip com o seu xv6 modificado (execute um `make clean` para se livrar de arquivos desnecessários, e compacte o diretório todo)
- Um arquivo pdf contendo um diário de bordo da implementação do exercício
- **OS ARQUIVOS DEVEM SER NOMEADOS COM O SEU NÚMERO USP.** Por exemplo: 555666777.zip 55566677.pdf

O diário de bordo tem como objetivo ajudá-lo a entender o que está acontecendo (e me ajudar a avaliar se você está realmente entendendo o que deveria ser entendido), porém sem o desgaste de escrever um relatório formal.

O diário não tem um formato predefinido, porém sugere-se que ele seja dividido em "entradas" referente à uma sessão de programação. Uma entrada pode conter o seu objetivo antes de iniciar a mexer no código, pensamentos que você teve ao longo da sessão de programação, e finalizar com o registro dos sucessos obtidos e reveses encontrados. Você pode escolher um formato diferente que faça mais sentido para você, porém, o diário deve registrar claramente o seu processo de entendimento e resolução do problema, ou seja, deve conter os "porquês" e não apenas os "o quês". As entradas do diário não precisam necessariamente ser muito longas ou elaboradas.

Veja um possível exemplo de uma entrada do diário (de um EP hipotético deu uma outra disciplina hipotética de um universo paralelo hipotético):

```
15/09/1999

Tarefa: preciso modificar os dados transmitidos para incluir a
possibilidade de serem criptografados.

=> os dados são armazenados num struct iov
==> onde que a estrutura é alocada?
==> aparentemente é alocada no momento em que o usuário transmite dados,
então posso aumentar o tamanho da alocação neste momento (aparentemente
deu certo!)

=> preciso pensar numa forma de reduzir o tamanho dos dados repassados
após remover a tag de encriptação…
==> tentei apenas repassar o valor de tamanho reduzir para a função de
processamento da struct, porém deu um problema…
```

A nota do EP terá três componentes, todas com mesmo peso:

- Diário de bordo
- Teste 1: teste simples da chamada e sistema (sistema com 1 CPU)
- Teste 2: teste da chamada de sistema em um programa que usa `fork()` (sistema com 2 CPU)

No eDisciplinas serão fornecidos dois programas que são semelhantes aos testes 1 e 2.

Boa sorte e divirta-se!