List 7

Patch

Do każdego z zadań załączam odpowiednie pliki patch:

- 7.1.patch
- 7.2.patch
- 7.3.patch

Oraz całościowy patch do upstream xv6:

- all.patch

Exercises

Exercise 1

(10p) Napisać `ps` w xv6.

Dodałem syscalle `getnumproc()`, `getmaxpid()` oraz `getprocinfo()`, licznik `sw` (context switches) do struktury proc.

Znając strukturę `procstate`:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Listowane są jedynie aktywne procesy (nie posiadające statusu *UNUSED*).

Wystawiłem prostą strukturę danych `uproc` (user process) do przekazania informacji o procesie do użytkownika:

- Nadrzędny PID `ppid`
- Liczba przełączeń kontekstu przez schedulera `sw`
- Rozmiar w bajtach `sz`

Oraz kilka dodatkowych parametrów niewymienionych w zadaniu:

```
struct uproc {
    int pid; // process id
    int ppid; // parend process id
    int sz; // size
    int sw; // number of context switches
    int state; // state (enum procstate, e.g. 4 == RUNNING, 2 == SLEEPING)
    char name[16]; // debug name, passed because why not
};
```

Dodatkowo w celu zachowania spójności `ptable` po stronie jądra syscall `getprocinfo` tymczasowo przetrzymuje lock `ptable.lock`.

```
if (!holding(&ptable.lock)) {
  acquire(&ptable.lock);
}

// some cool getprocinfo code available in 7.1.patch

release(&ptable.lock);
```

Efekt końcowy to

```
$ ps
Process Status
NumProc 3 MaxPID 3
PID: 1, PPID: 0, Size: 12288, State: 2, Name: init, Sw: 27
PID: 2, PPID: 1, Size: 16384, State: 2, Name: sh, Sw: 19
PID: 3, PPID: 2, Size: 12288, State: 4, Name: ps, Sw: 4
```

Wraz z wykonywaniem kolejnych procesów widzimy, że sw w `sh` rośnie, co oznacza, że po zakończeniu zadanego procesu kontekst przełącza się z powrotem na `sh`.

Exercise 2

(15p) Dopisać priority-scheduler do xv6.

Wykorzystałem najprostszą postać algorytmu Fixed-Priority Pre-Emptive scheduling

Wyjaśnienie jego logiki jest następujące: -> znajdź najważniejszy aktywny proces -> wykonuj póki się nie skończy, lub do czasu yield -> znajdź kolejny najważniejszy aktywny proces

Jako że xv6 ma już system interrupt na timer do `yield()` wystarczyło jedynie zmodyfikować scheduler w `proc.c` (oraz oczywiście dopisać dwa syscalle):

- dodałem priority (int) do struktury proc oraz dwa syscalle `setprio`, `getprio` które pozwalają kontrolować te zmienną:

```
// 7.2 add sys_setprio
int sys_setprio(void) {
  struct proc *curproc = myproc();

  int prio;

  if (argint(0, &prio) < 0)
  {
    return -1;
  }

  if (prio < 0 || prio > 1000)
  {
    return -1;
  }

  curproc->priority = prio;

  return 0;
}

// 7.2 add sys_getprio
int sys_getprio(void) {
  struct proc *curproc = myproc();

  return curproc->priority;
}
```

Przy wyborze nowego procesu zamiast wybierać pierwszy w pętli sprawdzam, który z bieżących ma najwyższy priorytet:

```
struct proc *p, *p1;
struct proc *highestPriorityProc;

highestPriorityProc = p;

for (p1=ptable.proc; p1 < &ptable.proc[NPROC]; p1++) {
  if (p1->state != RUNNABLE) {
    continue;
  }
  if (p1->priority > highestPriorityProc->priority) {
    highestPriorityProc = p1;
  }
}

p = highestPriorityProc;
```

Test widoczny w `testsched.c` napisałem aby umożliwiał sprawdzenie następujących warunków:

- długie taski wykonują się zgodnie z priorytetem (im większy priorytet tym szybciej się kończą -> spełnia (a))
- ostatnie wystartowane taski były krótkie więc wykonały się najpierw (nie ma drainage -> spełnia (b))
- między taskami o tym samym priorytecie leci round robin, w tym przypadku jako że są mniejsze niż jedna ramka czasowa to kończą się w kolejności zakolejkowania

```
$ testsched
testsched starting
priority in [0,1000], higher value ~ more cpu time
Child (id=6, prio=99, task='short') process has finished
Child (id=7, prio=99, task='short') process has finished
Child (id=8, prio=99, task='short') process has finished
Child (id=3, prio=500, task='long') process has finished
Child (id=2, prio=300, task='long') process has finished
Child (id=5, prio=200, task='long') process has finished
Child (id=4, prio=150, task='long') process has finished
Child (id=1, prio=100, task='long') process has finished
testsched done.
```

Exercise 3

(15p) Dopisać forkcb - fork z callbackiem, który najpierw wykonuje wskazaną funkcję.

Dodałem syscalle `forkcb` oraz `exitcb`, oraz dodałem dwie zmienne do struktury `proc`.

```
// struct proc in proc.h
uint fork_callback;          // Address of the function to be called after
fork
uint original_eip;           // Original EIP
```

W momencie gdy wywołany zostaje `forkcb` ustawiam callback oraz oryginalny EIP, a w `exitcb` wracam do oryginalnego EIP (całość odbywa się poprzez modyfikację trapframe programu).

```
// 7.3 add forkcb, exitcb
int sys_forkcb(void (*callback)())
{
  struct proc *curproc = myproc();

  if (argptr(0, (void*)&callback, sizeof(void*)) < 0)
    return -1;

  curproc->fork_callback = (uint)callback;
  return 0;
}

int sys_exitcb(void)
{
```

```
  struct proc *curproc = myproc();
  curproc->tf->eip = curproc->original_eip;
  return 0;
}
```

Wystarczy następnie że w funkcji `fork` sprawdzę czy obecny jest adres `fork_callback` i jeśli tak to go wykonam.