Лекция 4: Процессы

Алексей Линёв
Александр Мощук
Кирилл Погорельский

somes slides are adapted from the OS course at the University of Washington

Управление процессами

- Мы переходим к изучению процессов, потоков и синхронизации выполнения
  - это важнейшая часть курса
  - по данной теме обязательно будут вопросы на промежуточном тесте
- Данная лекция посвящена процессам и управлению ими
  - что является единицей исполнения в ОС?
  - как процессы представлены в ядре ОС?
  - как ЦП распределяется между процессами?
  - каково множество состояний процесса?
    - и как реализуются переходы между ними?

Процесс

- *Процесс* – абстракция ОС, представляющая исполнительную активность
  - это объект исполнения
  - это объект планирования
  - это динамический контекст выполнения
    - напротив, программа – это статический набор инструкций
- Процессы часто называют заданиями, задачами или последовательными процессами
  - процесс – это программа в процессе выполнения
    - представляет последовательное выполнение инструкций программы

Из чего состоит процесс?

- Процесс включает
  - адресное пространство
  - код выполняющейся программы
  - данные выполняющейся программы
  - стек и указатель на его вершину (SP)
    - позволяет отследить историю вызовов процедур
  - программный счетчик, указывающий на следующую инструкцию (PC)
  - регистры общего назначения и их значения
  - множество ресурсов ОС
    - открытые файлы, сетевые соединения, звуковые каналы,…
- Другими словами – все, что необходимо для выполнения программы
  - или для перезапуска программы, исполнение которой было в определенный момент прервано

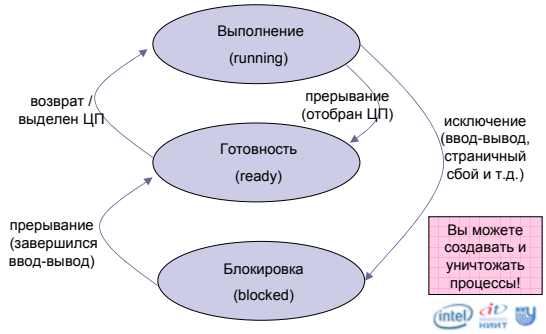
Управляющий блок процесса (УБП) (Process control block, PCB)

- Существует структура данных – управляющий блок процесса (УБП) – в которой хранятся все эти данные
  - каждый процесс имеет уникальный целочисленный идентификатор – process ID (PID)
- Когда процесс не выполняется, ОС хранит его контекст выполнения в УБП
  - программный счетчик (PC), указатель вершины стека (SP), значения регистров и т.д.
  - при вытеснении процесса с ЦП, его состояние сохраняется в УБП
- Замечание. Многие считают, что для этого используется методика, доступная лишь "посвященным"
  - и невообразимые структуры данных, которые вы никогда бы не смогли придумать сами
- *Неправда! Все гораздо проще, чем вам кажется!*

Состояния процесса

- Каждый процесс имеет *состояние*, указывающее на то, что он делает в настоящий момент
  - готов к выполнению – ожидает выделения ЦП
    - мог бы выполняться, но ЦП выделен другому процессу
  - выполнение – исполняется на ЦП
    - процесс, которому предоставлен ЦП
    - вопрос: сколько процессов могут выполняться одновременно?
  - ожидание – ожидает наступления события (например, ввода-вывода)
    - не может продолжать выполнение до наступления события
- Исполняющийся процесс переходит из состояния в состояние
  - UNIX: команда *ps* показывает состояние процесса в столбце STAT
  - в каком состоянии процесс находится большую часть времени?

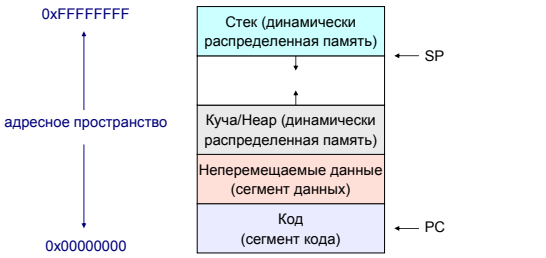
Состояния пользовательских процессов



Вернемся к рассмотрению УБП (PCB)

- УБП – структура данных с большим количеством полей
  - идентификатор процесса (PID)
  - идентификатор родительского процесса
  - состояние
  - программный счетчик (PC), указатель на вершину стека (SP), значения регистров
  - информация об адресном пространстве процесса
  - идентификатор пользователя-владельца процесса (UID)
  - приоритет планирования
  - статистическая информация
  - указатели для использования в очередях состояний
- В Linux
  - определены в `task_struct` (*`include/linux/sched.h`*)
    - **Обратите внимание на это для проекта №1!**
  - более 95 полей!

Адресное пространство процесса



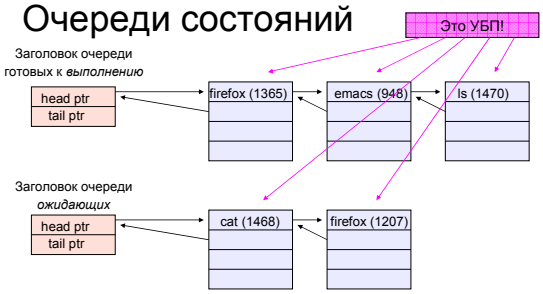
УБП и состояние аппаратного обеспечения

- Когда процесс в состоянии выполнения, его контекст выполнения (аппаратный контекст) загружен в ЦП
  - программный счетчик (PC), указатель на вершину стека (SP), значения регистров
  - ЦП содержит актуальные значения
- При остановке процесса (переходе в состояние ожидания) ОС сохраняет значения регистров в УБП
  - когда ОС переводит процесс назад в состояние выполнения, она загружает в регистры значения из УБП
- Процедура передачи ЦП от одного процесса другому называется *переключением контекста (context switch)*
  - системы разделения времени могут выполнять сотни или тысячи переключений в секунду
  - занимает порядка 5 микросекунд на современных системах

Очереди состояний

- ОС поддерживает набор очередей, в которых представлены состояния всех процессов в системе
  - обычно, по одной очереди для каждого состояния
    - выполнение, готов к выполнению,…
  - каждый УБП включен в очередь, соответствующую его текущему состоянию
  - при изменении состояния процесса, он извлекается из своей текущей очереди и включается в другую
- Повторюсь – *ничего особенного*! УБП перемещаются между очередями, реализованными в виде списков. *There is no magic!*

Очереди состояний



- Также может существовать множество очередей ожидания, каждая для своего типа ожидаемого события (от конкретного устройства, от таймера, ожидание сообщения,…)

Идентификатор процесса (Process ID)
Указатель на процесс-родитель (Pointer to parent)
Список процессов-потомков (List of children)
Состояние (Process state)
Указатель на описание адресного пространства (Pointer to address space descriptor)
Программный счетчик (Program count) **Указатель на вершину стека (stack pointer)** **Значения всех регистров (all register values)**
Идентификатор пользователя (uid, user id) Идентификатор группы (gid, group id) Эффективный идентификатор пользователя (euid, effective user id)
Список открытых файлов (Open file list)
Приоритет планирования (Scheduling priority)
Статистическая информация (Accounting info)
Указатели очереди состояния (Pointers for state queues)
Код завершения (возврата) (Exit ("return") code value)

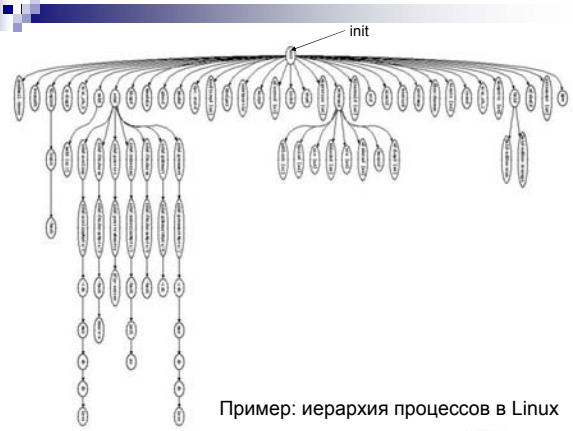
внутреннее представление УБП (упрощенное!)

УБП и очереди состояний

- УБП – структуры данных
  - динамически создаваемые в адресном пространстве ОС
- При создании процесса
  - ОС выделяет для него УБП
  - ОС инициализирует УБП
  - ОС помещает УБП в нужную очередь состояния
- В ходе выполнения процесса
  - ОС перемещает его УБП из очереди в очередь
- При завершении процесса
  - УБП может сохраняться в течении некоторого времени (код завершения,…)
  - в конце концов, ОС освободит его УБП

Создание процесса

- Новые процессы создаются уже существующими процессами
  - создатель называется *родителем* (parent)
  - созданный процесс называется *потомком* (child)
  - UNIX: команда ps показывает родителя процесса в столбце PPID
  - кто и когда создает первый процесс?
- В некоторых системах потомки наследуют ресурсы и привилегии своего родителя
  - UNIX: потомок наследует от родителя идентификатор пользователя, окружение (environment), список открытых файлов и т.д.
- После создания потомка родитель может дождаться завершения его выполнения или продолжить свое выполнение параллельно с потомком!



Пример: иерархия процессов в Linux

Создание процесса в UNIX

- В UNIX процессы создаются посредством выполнения системного вызова **`fork()`**
  - создает и инициализирует новый УБП
  - создает новое адресное пространство процесса
  - инициализирует новое адресное пространство, *полностью копируя* адресное пространство родителя
  - выделяет новому процессу те же ресурсы, которые выделены родителю (например, открытые к моменту создания файлы)
  - помещает новый УБП в очередь готовых к исполнению
- системный вызов **`fork()`** "возвращает управление дважды"
  - один раз – родителю, и один раз – потомку
  - родителю возвращается PID потомка
  - потомку возвращается 0
- **`fork()`** = "создай мою копию"

exec() vs. fork()

- Как же запускаются новые программы, ведь мы всего лишь запустили копию уже исполняющейся?
  - существует системный вызов **`exec()`**!
  - **`int exec(char * prog, char * argv[])`**
- **`exec()`**
  - останавливает выполнение текущего процесса
  - загружает программу из файла "prog" в адресное пространство
  - инициализирует аппаратный контекст, устанавливает аргументы вызова для новой программы
  - помещает УБП в очередь готовых к выполнению
  - Замечание: при этом новый процесс не создается!
- с точки зрения текущего процесса, системный вызов **`exec()`** "не возвращается"
  - что произошло, если он вдруг вернулся?

Командные интерпретаторы UNIX (повторение)

```
// псевдокод
int main(int argc, char **argv)
{
  while (1) {
    char *cmd = get_next_command();
    int pid = fork();
    if (pid == 0) {
        манипулирование файловыми дескрипторами stdin/stdout/stderr
        exec(cmd);
        panic("exec failed!");
    } else {
        waitpid(pid,&status,0);
    }
  }
}
```

для первого проекта это необязательно

Перенаправление ввода-вывода…

- $ ./myprog <input.txt >output.txt # UNIX
  - каждый процесс имеет таблицу открытых файлов
  - согласно общепринятому соглашению
    - 0: stdin
    - 1: stdout
    - 2: stderr
  - процесс-потомок наследует таблицу открытых файлов процесса-родителя
- Таким образом, шелл:
  - сохраняет текущие записи, соответствующие stdin/stdout
  - открывает input.txt в качестве stdin и output.txt в качестве stdout
  - fork()…
  - восстанавливает начальные значения stdin/stdout

Перенаправление ввода-вывода

- Обратите внимание – перенаправление абсолютно прозрачно для процесса-потомка
- Другие варианты перенаправления
  - $ ./myprog >>output.txt
  - $ ./myprog >output.txt 2>&1
  - $ ./myprog | less
  - $ ./myprog &

Сигналы…

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process
SIGTTOU	22,22,27	Stop	tty output for background process

Сигналы

```
int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);


struct sigaction {
     void (*sa_handler)(int);
     sigset_t sa_mask;  // mask of blocked signals
     int sa_flags;      // signal handling options
}
```