Операционные системы

Виктор Ашик для «Курса информационных технологий»

Яндекс

4bsd acorn aix aos bos bsd chorus dynix esa gnu hp-ux ibm interactive irix ix linux lite lsx mac microport mk mp mvs net open openedition os osf osx pwb qnx release reliantunix risc rt rtos sco sinix solaris svr4 system trusted ucla ultrix unicos unix unixware ux xenix xinu

Как выполняли программы до появления ОС?

Назовите ЭВМ без ОС?

Назовите ЭВМ без ОС?

2001 год выпуска

ЭЛЕКТРОНИКА МК-152

Учебная ОС: xv6

Современная компактная реализация Unix V6 для x86

Основана на Lions' Commentary on UNIX 6th Edition, with Source Code.

— не требует знания архитектуры PDP-11
— написана на современном C
— применяется в курсе MIT Operating Systems Engineering (6.828)

Назначение ОС

- совместная работа программ
- обслуживание программ
  - абстракция оборудования и упрощение работы с ним
  - (псевдо)параллельное исполнение нескольких программ
  - предоставление услуг

Какие Unix вы знаете?

UNIX®

1969 (AT&T/Bell Labs: Томпсон, Керниган, Ритчи)

Распространялась свободно

Стандарт де-факто и де-юро (POSIX)

AIX, Solaris, Mac OS X/FreeBSD/NetBSD/OpenBSD — генетически UNIX®

Linux, Minix — реализации

Есть реализации POSIX для Windows

Далее — *nix

*nix = переносимость

Стандарты

— **P**ortable **O**perating **S**ystem **I**nterface [for Uni**x**]

— ISO C/C++ (STL)

Runtime

— Shell

— Java

— Perl/Python/PHP/...

Операционная система должна

Загружаться (bootstrapping)

Обслуживать приложения

- Запускать процессы
- Отвечать на системные вызовы

Что приложения хотят от ОС?

Абстракцию оборудования

Координацию совместной работы с устройствами

Изоляцию сбойных приложений

Обмен между приложениями

Какие сервисы предлагает ОС?

Процессы

Память

Содержимое файлов

Каталоги и имена файлов

Безопасность

... (пользователи, IPC, сеть, терминалы)

Как приложения обращаются к системе?
ABI - syscalls

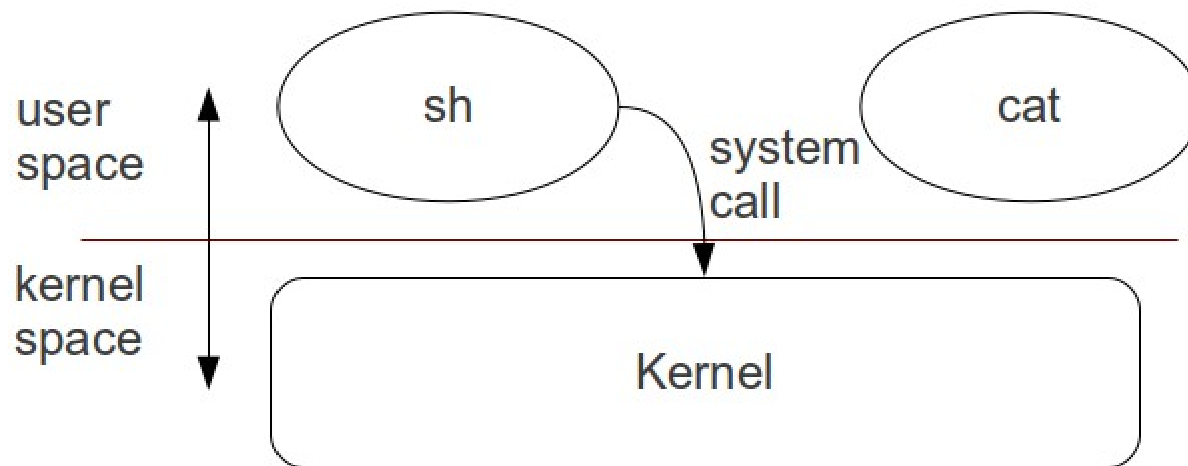
```
mov edx,len
mov ecx,msg
mov ebx,1      ;file descriptor (stdout)
mov eax,4      ;system call number (sys_write)
int     0x80   ;call kernel

mov eax,1      ;system call number (sys_exit)
int     0x80   ;call kernel

section .data

msg db 'Hello, world!',0xa
len equ $ - msg
```

Системные вызовы



Как приложения обращаются к системе?
ABI - syscalls

```
int main() {
	__asm__(
		"movl $20, %eax    \n"
		"call *%gs:0x10    \n"
		"movl %eax, pid    \n"
	);
	printf("pid is %d\n", pid);
	return 0;

}
```

Как это записать на С?
ABI - syscalls

```
#include <unistd.h>
/* ... */
const char msg[] = "Hello world";
write( STDOUT_FILENO, msg, sizeof( msg ) - 1 );

/* Где вызов функции write превращается в syscall? */
```

Демо

`ltrace`

`strace`

Shell: /bin/*sh

Интерпретатор команд *nix

```
While (1) {
    write (1, "$ ", 2);
    readcmd (cmd, args);   // parse user input
    if ((pid = fork ()) == 0) {  // child?
       exec (cmd, args, 0);
    } else if (pid > 0) {   // parent?
       wait (0);   // wait for child to terminate
    } else {
       perror ("fork");
    }
}
```

Что делает код:

Системные вызовы: read, write, fork, exec, wait

Соглашения:

— при ошибке код возврата -1

— Код ошибки заносится в errno

— Функция perror выводит сообщение об ошибке errno

Вызов: fork

Создает потомка, точную копию вызвавшего процесса и возвращает управление родителю и потомку

Совпадает:

— память процесса (код, данные, стек)

— Атрибуты процесса: владелец, права, контекст

Отличаются:

— PID

— Код возврата при успешном fork

 —У потомка 0

 —У родителя pid потомка

Вызов: exec

Замещает содержимое памяти вызвавшего процесса инструкциями и данными из файла на диске

То есть исполняет файл

Процесс остается тем же (сохраняет pid, uid, …)

Вызов: wait

Ожидает завершения одного из потомков

Зачем?

Что будет если потомок завершится раньше wait?

Почему fork/exec разделены?

Как ls узнает текущий каталог, stdin, stdout?

— Наследует cwd и открытые файлы от родителя

Copy-on-write делает fork очень быстрым

Вызовы: read/write

Аргументы:

— Номер файла

— Указатель на буфер

— Число байтов

Возвращает число байтов

Другие системные вызовы

fork()

exit()

wait()

kill(pid)

getpid()

sleep(n)

exec(filename, *argv)

sbrk(n)

open(filename, flags)

read(fd, buf, n))

write(fd, buf, n)

close(fd)

dup(fd)

pipe(p)

chdir(dirname)

mkdir(dirname)

mknod(name, major, minor)

fstat(fd)

link(f1, f2)

unlink(filename)

```
int pid;
pid = fork();
if(pid > 0){
  printf("parent: child=%d\n", pid);
  pid = wait();
  printf("child %d is done\n", pid);
} else if(pid == 0){
  printf("child: exiting\n");
  exit();
} else {
  printf("fork error\n");
}
```

```
char *argv[3];
argv[0] = "echo";
argv[1] = "hello";
argv[2] = 0;
exec("/bin/echo", argv);
printf("exec error\n");
```

```
char buf[512];
int n;
for(;;){
  n = read(0, buf, sizeof buf);
  if(n == 0)
    break;
  if(n < 0){
    fprintf(2, "read error\n");
    exit();
  }
  if(write(1, buf, n) != n){
    fprintf(2, "write error\n");
    exit();
  }
}
```

Организация ОС: Традиционный подход

Виртуализуется часть ресурсов

- ЦПУ
- память

Каждое приложение «монопольно» ими распоряжается

Зачем? Так проще писать приложения

*nix, Windows NT

Пример: виртуализация ЦПУ

Цель: эмулировать отдельный ЦПУ для каждого процесса

- Переключение ЦПУ прозрачно
- Процессу не нужно беспокоиться о других процессах

ОС выполняет процессы по-очереди до прерывания по таймеру

Таймер позволяет процессу не беспокоиться о переключениях

Пример: виртуализация ЦПУ

Как это достигается?

— ОС сохраняет и восстанавливает состояние (контекст) ЦПУ при каждом переключении

Что сохраняется?

— Регистры, флаги, указатели на таблицы виртуальной памяти

Где сохраняется?

— В таблице процессов

Прерывание таймера приводит к переключению на другой процесс

Демо

vmstat

cat /proc/1/status

ps

Сколько у процесса может быть состояний?

man ps
/STATE

Пример: виртуализация памяти

Идея: для процесса доступно:

— все адресное пространство 2^32

— оно ссылается на его «частную память»

Удобно, безопасно

Как достигается? Физическая память общая.

Варианты виртуализации памяти

Вытеснение процесса на диск при переключении

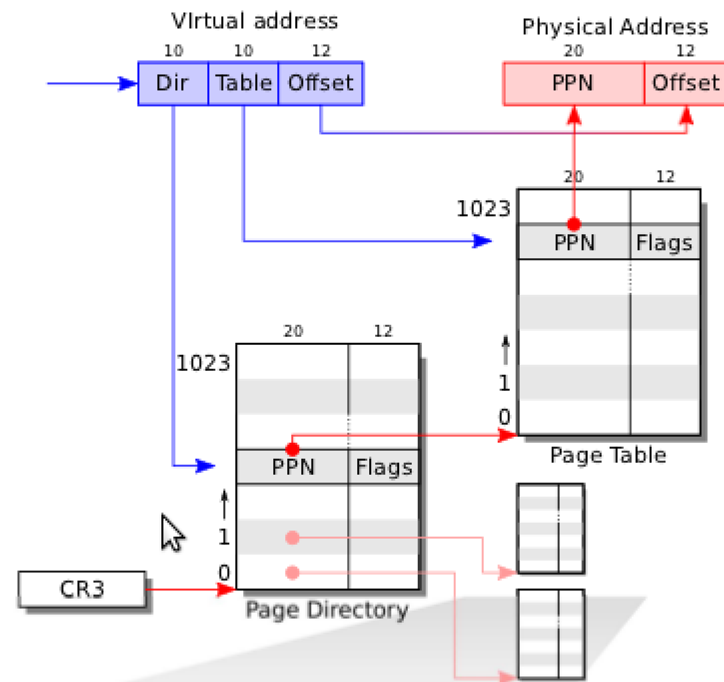
- медленно
- имело смысл в 90-х

Использование сегментов x86

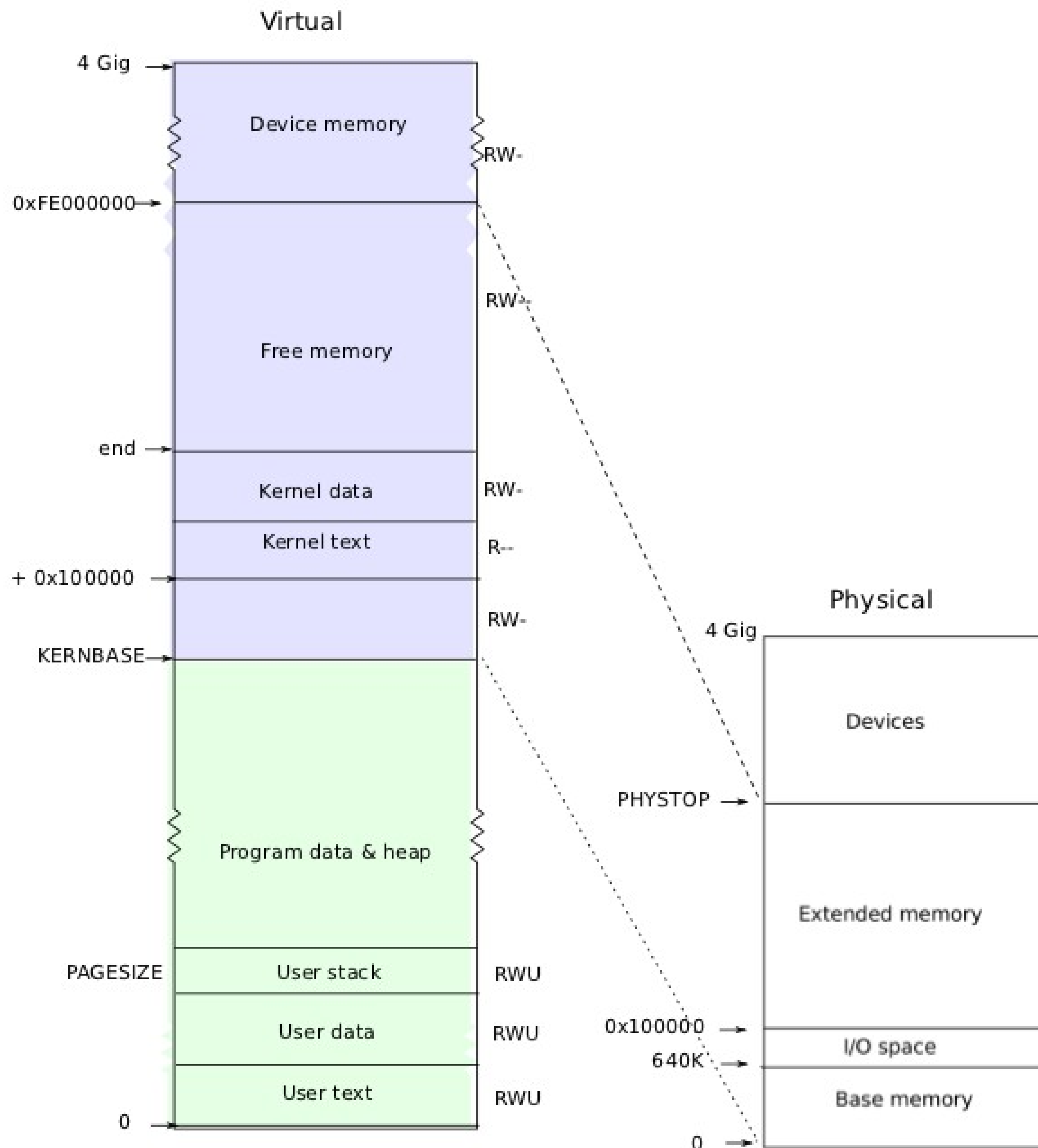
- переключать сегменты CS, DS на разные адреса
- фрагментация памяти

Применить страничный механизм ЦПУ

- Таблица физических адресов каждого 4К блока — страницы памяти
- Это таблица страниц
- Страницы можно помечать недействительными и переносить в подкачку — на диск
- При обращении к такой странице возникает прерывание и страницу можно загрузить обратно
- copy-on-write — избавить fork от копирования памяти



Page table and page directory entries are identical except for the D bit.



Virtual
4 Gig
Device memory
RW-
0xFE000000
RW-
Free memory
end
Kernel data
RW-
Kernel text
R--
+ 0x100000
RW-
KERNBASE
Program data & heap
PAGESIZE
User stack
RWU
User data
RWU
User text
RWU
0
Physical
4 Gig
Devices
PHYSTOP
Extended memory
0x100000
I/O space
640K
Base memory
0

Демо

vmstat

cat /proc/swaps

swapon -s

Подходы к построению систем

Монолитное ядро

Гибридное (модульное ядро)

Микроядро

Экзоядро

Планирование процессорного времени

Многозадачность:

— Кооперативная (Windows 3.1, Mac OS 9, WOW16)

— Вытесняющая (preemptive)

Планирование процессорного времени

Долгосрочное

— batch

Среднесрочное

— BSD: выгрузка в подкачку неактивных процессов

— Android: завершение задач при нехватки памяти

Краткосрочное

— Планировщик ЦПУ в ядре

Алгоритмы планировщика ЦПУ

FIFO / FCFS

SJF (Shortest Job First)

Приоритетное планирование (OCPB)

Round-Robin

Многоуровневые очереди с обратной связью

- CFQ
-

Управление памятью

Распределение

Защита

Разделение (sharing)

Логическая организация (сегменты)

Физическая организация (подкачка)

Планирование ввода-вывода

Цели:

— Сократить время поиска диском

— Приоритезировать ввод-вывод

— Разделить полосу пропускания устройства между процессами

— Гарантировать исполнение запросов не позднее крайнего срока

Планирование ввода-вывода

Реализации:

- Случайное планирование (RSS)
- FIFO/FCFS
- LIFO
- CFQ
- SCAN (лифт)
- Noop (FIFO с объединением)
- Anticipatory (упреждающий)
- Deadline (
- ...

Управление памятью

Кеш-буфер

Промахи страниц

- Жесткие
- Мягкие

Алгоритмы виртуальной памяти

- LRU
- Опережающая подгрузка
- ...

#yakit2

03.10 Что такое быть системным администратором.
05.10 Операционные системы, их история, устройство и функционирование.
10.10. Архитектура ЭВМ и интерфейсы периферийного оборудования.
12.10 Устройство GNU/Linux на примере Debian/Ubuntu и Fedora/RHEL (Экстрополис).
17.10 Управление хранением данных: RAID, LVM, резервное копирование и восстановление.
19.10 Системы управления базами данных на примере MySQL.
24.10 Экскурсия. Сети, протоколы и сетевое оборудование.
26.10 Управление хранением данных: файловые системы (Экстрополис).
31.10 Информационная безопасность и средства её обеспечения.
02.11 Виртуализация вычислительных ресурсов.
07.11 Экзамен

Список литературы

1. Э. Реймонд, «Искусство программирования для Unix», Вильямс, 2005
2. Б. Керниган, Р. Пайк, Unix. Программное окружение, Символ-Плюс, 2003
3. Про PDP-11 (сайт Сергея Вакуленко, октябрь 2011)
   http://vak.ru/doku.php/proj/pdp11
4. Lions commentary on the Sixth Edition UNIX Operating System:
   http://www.lemis.com/grog/Documentation/Lions/
5. Source code listing for the Lions' Commentary in PDF and PostScript
   http://v6.cuzuco.com/
6. Сайт курса MIT 6.828 2011:
   http://pdos.csail.mit.edu/6.828/2011/

Виктор Ашик
куратор практик

vashik@yandex-team.ru
+7 495 739-70-00

ул. Льва Толстого, 16
Москва, Россия, 119021
Яндекс Москва