操作系统辅导 I

李嘉睿

2020.11.18

Contents

1 操作系统基本概念

1. 从应用角度看，操作系统提供了哪些**层次**的接口？

2. 操作系统内核架构有哪些？分别有什么特点/优劣势？

3. 什么是 *dual-mode*、特权级（*privilege level*）？在 x86 架构中这是如何表示的？导致用户态和内核态发生切换的三种情况分别是什么？

2 进程 & 线程

进程是运行中程序的一个示例，是资源分配的基本单位。线程是 CPU 调度的基本单位，分为用户态线程和内核态线程，由 POSIX 的 pthreads 线程库提供支持。用户态线程又被称为纤程 (fiber routine)，由 POSIX 中的 ucontext 提供支持。一般将程序语言提供支持的纤程支持称为协程 (coroutine)，目前：Lua，go，C++，Ruby 等都提供了协程支持。当前程序语言发展趋势是从一对一模型向多对一模型进行转变。

2.1 比较进程和线程

1. 进程和线程有哪些区别？请分别画出单线程进程和多线程进程的虚拟地址空间布局。（注：布局中应包含 a. 程序代码 b. 程序数据 c. 堆 d. 用户栈 e. 代码库 (例如 libc)f. 内核栈 g. 内核代码及数据）

2. 目前主流操作系统使用的都是一对一线程模型（例如：linux、windows）。为什么内核态线程需要一个独立的内核栈？如果内核栈在用户地址空间会怎么样？

3. 一个用户态线程对应一个内核态线程，则 1）用户态和内核态相互切换的时候，必须知道对方的内核栈地址并配置寄存器 2）线程间切换的时候，一定发生在内核态。则也要知道下一个执行线程的内核栈地址。但通过 pintos 的 *thread* 结构体，可以发现只存储有一个 *stack* 指针。请问 1）存储的是内核栈地址还是用户栈地址？为什么？2）另一个栈地址存储在什么位置？3）其它保存的寄存器的值又存在什么位置？

```
1  struct thread
2  {
3    /* Owned by thread.c. */
4    tid_t tid;                          /* Thread identifier. */
5    enum thread_status status;          /* Thread state. */
6    char name[16];                      /* Name (for debugging purposes). */
7    uint8_t *stack;                     /* Saved stack pointer. */
8     ...
9  };
```

2.2 进程 & 线程接口

2.2.1 进程接口：fork、wait、exec

```c
/* q1: 若fork调用都成功执行，则该程序创建了几个新进程？ */
int main(void) {
    for (int i = 0; i < 3; i++)
    pid_t pid = fork();
}
/* q2: 下面程序的打印结果可能是什么？*/
int main(void) {
    int stuff = 5;
    pid_t pid = fork();
    printf("The last digit of pi is %d\n", stuff);
    if (pid == 0)
    stuff = 6;
}
/* q3: 假设PID=202011，下面程序的打印结果可能是什么？*/
int main(void) {
    pid_t pid = fork();
    int exit;
    if (pid != 0) {
        wait(&exit);
    }
    printf("Hello World\n: %d\n", pid);
}
/* q4: 下面程序的打印结果是？*/
int main(void) {
    char** argv = (char**) malloc(3*sizeof(char*));
    argv[0] = "/bin/ls";
    argv[1] = ".";
    argv[2] = NULL;
    for (int i = 0; i < 10; i++) {
        printf("%d\n", i);
        if (i == 3)
        execv("/bin/ls", argv);
    }
}
```

2.2.2 线程接口：pthreads

```c
/* q1: 下列程序的输出可能是什么？ */
/* q2: 如何修改能够在"MAIN"之后输出""HELPER" */
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}
int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

3 进程同步

3.1 共享对象的实现

共享对象 (shared objects) 指的是多线程环境下能保证数据访问安全性（也称为线程安全）的对象。例如：java 中的 *ConcurrentHashMap*。共享对象由同步变量（例如：锁、信号量）以及状态变量（被保护的变量）构成。共享变量是分层实现的，下表已经给出了共享变量的分层结构，请写出每一层的具体实现方法。

层	实现方法
并发应用程序	多线程服务器、多线程数据库引擎
共享对象	
同步变量	
原子指令	
硬件	

3.2 同步原语实现

假设你正在使用线程实现一个多处理器系统。信号量和管程的接口如下：

信号量：

```
1  struct sem_t {
2  // internal fields
3  };
4  void sem_init(sem_t *sem, unsigned int value) {
5  // Initialize semaphore with initial value
6  ...
7  }
8  void sem_P(sem_t *sem) {
9  // Perform P() operation on the semaphore
10 ...
11 }
12 void sem_V(sem_t *sem) {
13 // Perform V() operation on the semaphore
14 }
```

管程包含了一个锁和一个或多个条件变量 (condition variables)。它们分别的接口如下

```
struct lock_t {
   // Internal fields
};
void lock_init(lock_t *lock) {
   // Initialize new lock
   …
}

void acquire(lock_t *lock) {
   // acquire lock
   …
}
void release(lock_t *lock) {
   // release lock
   …
}
```

```
struct cond_t {
   // Internal fields
};
void cond_init(cond_t *cv, lock_t *lock) {
   // Initialize new condition variable
   // associated with lock.
   …
}
void cond_wait(cond_t *cv) {
   // block on condition variable
   …
}
void cond_signal(cond_t *cv) {
   // wake one sleeping thread (if any)
   …
}
void cond_broadcast(cond_t *cv) {
   // wake up all threads waiting on cv
   …
}
```

1. 假设 1：我们使用 *test-and-set* 实现了锁同时希望避免长时间的自旋 *spin-waiting*。假设 2：我们在用户态使用这些锁来同步多个处理器上的线程。请解释为什么仍需要内核为该实现提供支持。并给出相应的系统调用接口 (提示：可参考 *Linux Futex* 接口)。

2. 请使用信号量实现锁，下列需要实现的每个方法均不超过 5 行。

```
1  struct lock_t {
2
3
4
5  };
6  void lock_init(lock_t *lock) {
7
8
9
10 }
11 void acquire(lock_t *lock) {
12
13
14
15 }
16 void release(lock_t *lock) {
17
18
19
20 }
```

3. 使用信号量实现管程，下列需要实现的方法均不超过 5 行。

```
1  struct sem_t {
2
3
4
5  };
6  void sem_init(sem_t *sem, unsigned int value) {
7
8
9
10 }
11 void sem_P(sem_t *sem) {
12
13
14
15 }
16 void sem_V(sem_t *sem) {
17
18
19
20 }
```

4. 请使用信号量以及第 2 题中实现的锁实现条件变量。不应当使用列表或队列。每个方法均不超过 5 行。提示：信号量接口无法查询其等待队列的大小，因此需要自行记录该信息。

3.3 补充材料：条件变量

4 案例分析：pnitos 中断控制器

上下文切换中，中断控制器既处理硬件异步中断也处理软件同步中断（即异常）。根据下方 pintos 的中断控制器源码（*intr-stubs.S*）回答问题。

```
/**
* An example of an entry point that would reside in the interrupt
* vector. This entry point is for interrupt number 0x30.
*/
.func intr30_stub
intr30_stub:
pushl %ebp /* Frame pointer */
pushl $0 /* Error code */
pushl $0x30 /* Interrupt vector number */
jmp intr_entry
.endfunc
/* Main interrupt entry point.

An internal or external interrupt starts in one of the
intrNN_stub routines, which push the 'struct intr_frame'
frame_pointer, error_code, and vec_no members on the stack,
then jump here.

We save the rest of the 'struct intr_frame' members to the
stack, set up some registers as needed by the kernel, and then
call intr_handler(), which actually handles the interrupt.

We "fall through" to intr_exit to return from the interrupt.
*/
.func intr_entry
intr_entry:
/* Save caller's registers. */
pushl %ds
pushl %es
pushl %fs
pushl %gs
pushal

/* Set up kernel environment. */
cld /* String instructions go upward. */
mov $SEL_KDSEG, %eax /* Initialize segment registers. */
mov %eax, %ds
mov %eax, %es
leal 56(%esp), %ebp /* Set up frame pointer. */

/* Call interrupt handler. */
pushl %esp
.globl intr_handler
call intr_handler
addl $4, %esp
.endfunc

/* Interrupt exit.

Restores the caller's registers, discards extra data on the
```

```
51 stack, and returns to the caller.
52
53 This is a separate function because it is called directly when
54 we launch a new user process (see start_process() in
55 userprog/process.c). */
56 .globl intr_exit
57 .func intr_exit
58 intr_exit:
59 /* Restore caller's registers. */
60 popal
61 popl %gs
62 popl %fs
63 popl %es
64 popl %ds
65
66 /* Discard 'struct intr_frame' vec_no, error_code,
67 frame_pointer members. */
68 addl $12, %esp
69
70 /* Return to caller. */
71 iret
72 .endfunc
```

0. *kernel interrupt handler* 是一个线程吗?

1. *pushal* 和 *popal* 指令分别的作用是什么?

2. 中断处理程序（ISR, *interrupt service routine*）必须依靠内核栈运行。为什么必须要内核栈? 哪一条指令负责将栈指针切换到内核栈?

3. 在 *call intr_handler* 指令前的 *pushl %esp* 作用是什么?

4. 在 *intr_exit* 函数中，如果我们反转了这 5 条 *pop* 指令的顺序会怎么样?

5 遇到的一些操作系统面试题

5.1 排序

编程实现链表排序。要求：额外空间复杂度为 *O(1)*。提示：参考 pintos 源码中链表排序的实现。

5.2 信号 (signal) 的实现

进程间通信中，信号是如何实现的？是否一旦满足信号触发条件，对应进程就能立刻收到消息？提示：是否和一种调度有关系？

6 gdb 简明教程

6.1 常用命令

- **run, r**：从程序起始处开始执行。允许参数传递和基本的 io 重定向。
- **quit, q**：退出 gdb
- **kill**：停止程序执行
- **break, break x if condition**：在指定的位置处挂起程序，例如：指定函数/行号处挂起 *break strcpy*，*break file.c:80*
- **delete, d**：删除某个断点
- **info breakpoints, i b**：显示所有断点，打印出的序号可用来删除断点
- **step, s/si**：执行下一行代码/汇编代码，跳进函数
- **next, n/ni**：执行下一行代码/汇编代码，不跳进函数
- **continue**：继续执行，直到下一个断点
- **finish**：继续执行，直到当前函数结束
- **print, p**：打印变量中的值
- **call**：执行任意函数/代码并打印结果
- **watch，rwatch，awatch**：当指定条件触发时挂起程序，例如：*watch* $x > 5$
- **backtrace，bt，bt full**：打印当前程序的函数调用栈（stack trace）
- **disassemble**：打印当前函数的反汇编表示

print 和 *call* 命令能够执行被 debug 的程序中的任意函数。支持对变量赋值和函数调用，例如：*call close(0)* 或 *print* $i = 4$

6.2 其它常用的命令和示例

- **使用 tui（终端用户界面）模式**：*layout src/asm/split*（使用源码/汇编/源码 + 汇编窗口）
- 调试时如果窗口文本出现混乱，可以**刷新窗口**：*ctrl + l*
- 显示当前进程 stack 信息：*i f*(是 *info frame* 的缩写) 。打印当前进程寄存器信息：*i r eip ebp esp*
- 打印自某个**内存地址**开始的值。
  - *x/nxw addr*：打印自 *addr* 开始的 n 个 16 进制 word(4 字节)
  - *x &var*：默认打印 *var* 对应地址开始 4 字节的值，以 16 进制显示，注意：x86 是小端法，显示时末位字节存在低地址处
  - *x $ebp*：打印 *$ebp* 寄存器中的值。
  - *x/i*：以指令形式打印内存中的代码
- 运行时输入重定向：$r < foo.txt$

- 在 gdb 中执行 shell 命令：*shell ls*
- pintos 中 gdb 相关的扩展宏。
  - pintos 打印线程列表：*dumplist &all_list thread allelem*
  - pintos 装载用户符号表（之后可以 debug 用户进程）：*loadusersymbols (file path)*
  - pintos 在 page fault 后打印调用栈：*btpagefault*

6.3 其他资源

- GDB Cheat Sheet: `https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf`
- pintos 中 debug 技巧：`https://courses.cs.washington.edu/courses/cse451/12au/pintos/doc/WWW/pintos_9.html#SEC149`

7 参考

- UCB cs162 课件、讲义与往年试卷：`https://cs162.eecs.berkeley.edu/`
- 上海交通大学并行与分布式系统研究所 - 现代操作系统：原理与实现
- Operating Systems: Principles and Practice (2nd Edition)
- Operating System Concepts (9th Edition)
- 阿里云操作系统组 2020 暑期实习面试