



دانشگاه صنعتی اصفهان
دانشکده مهندسی برق و کامپیوتر

عنوان: تکلیف اول درس سیستم‌های عامل ۱

نام و نام خانوادگی: علیرضا ابره فروش

شماره دانشجویی: ۹۸۱۶۶۰۳

نیم سال تحصیلی: پاییز ۱۴۰۰

مدرس: دکتر محمدرضا حیدرپور

دستیاران آموزشی: مجید فرهادی - دانیال مهرآیین - محمد نعیمی

فهرست مطالب

۲	۱	عنوان سوال اول
۲	۱.۱	عنوان بخش اول سوال اول
۲	۲.۱	عنوان بخش دوم سوال اول
۲	۳.۱	عنوان بخش سوم سوال اول
۲	۴.۱	عنوان بخش چهارم سوال اول
۲	۵.۱	عنوان بخش پنجم سوال اول
۲	۶.۱	عنوان بخش ششم سوال اول
۲	۷.۱	عنوان بخش هفتم سوال اول
۲	۲	عنوان سوال دوم
۳	۳	عنوان سوال سوم
۳	۴	عنوان سوال چهارم
۴	۵	عنوان سوال پنجم
۴	۱.۵	
۵	۲.۵	
۵	۳.۵	
۵	۶	عنوان سوال ششم
۵	۱.۶	
۶	۲.۶	
۸	۷	سوال هفتم
۸	۱.۷	سوال ۶ فایل
۹	۲.۷	سوال ۷ فایل
۹	۸	عنوان سوال جدول
۹	۹	عنوان سوال ششم
۹	۱.۹	عنوان بخش اول سوال ششم
۱۰	۲.۹	عنوان بخش دوم سوال ششم
۱۰	۱۰	عنوان سوال هشتم
۱۰	۱۱	ضمیمه

۱ عنوان سوال اول

اگر سوال بخش بندی شده نباشد، پاسخ آن در این قسمت نوشته می شود.

۱.۱ عنوان بخش اول سوال اول

پاسخ بخش اول سوال در این قسمت نوشته می شود.

۲.۱ عنوان بخش دوم سوال اول

پاسخ بخش دوم سوال در این قسمت نوشته می شود.

۳.۱ عنوان بخش سوم سوال اول

پاسخ بخش دوم سوال در این قسمت نوشته می شود.

۴.۱ عنوان بخش چهارم سوال اول

محتوای برخی از رجیسترها مانند Program Counter یا Stack Pointer توسط kernel handler قابل ذخیره سازی نیستند. چون خود آن‌ها هم نرم افزار هستند و تا CPU بخواهد آن‌ها را وارد مرحله اجرا کند، محتوای Program Counter و Stack Pointer عوض می شود. پس سخت افزار قبل از فراخوانی kernel handler، به طور خودکار محتوای رجیسترهای برخی از پروسس‌های متوقف شد را با push کردن آن‌ها در interrupt stack حفظ می کند.

۵.۱ عنوان بخش پنجم سوال اول

پاسخ بخش دوم سوال در این قسمت نوشته می شود.

۶.۱ عنوان بخش ششم سوال اول

پاسخ بخش دوم سوال در این قسمت نوشته می شود.

۷.۱ عنوان بخش هفتم سوال اول

پاسخ بخش دوم سوال در این قسمت نوشته می شود.

۲ عنوان سوال دوم

سیستم کال fork() یک پروسس جدید به نام پروسس child می سازد. پس از ایجاد پروسس child، هر دو پروسس (child و parent) دستوراتی که پس از fork() متناظرشان آمده اند را یک به یک اجرا می کنند. به این ترتیب پس از اولین fork()، در مجموع ۲ پروسس، پس از دومین fork()، در مجموع ۴ پروسس، پس از سومین fork()، در مجموع هشت پروسس و ... خواهیم داشت. در نتیجه پس از اجرای حلقه، تعداد childهای ایجاد شده برابر است با:

$$2 + 4 + \dots + 2^{\log_2 n} = 2^{(\log_2 n) + 1} = 2n$$

در مجموع با احتساب پروسس parent اصلی، ۲n پروسس خواهیم داشت.

۳ عنوان سوال سوم

در این قسمت با نحوه درج روابط و فرمول‌ها آشنا می‌شوید:

$$E = mc^2$$

۴ عنوان سوال چهارم

یک راه این است که مقدار رجیستری که به جدول وقفه (Interrupt Vector Table) اشاره می‌کند را عوض کنیم. درواقع می‌توانیم مکان دلخواهی از حافظه را به عنوان مکان جدول وقفه مشخص کنیم. توسط دستور lidt در x86 می‌توان این رجیستر را مقداردهی کرد. بنابراین این دستور است که مشخص می‌کند که این جدول کجای حافظه قرار گرفته است. بدیهی است که تنها سیستم عامل باید دسترسی استفاده از این دستور را داشته باشد. در غیراینصورت، اگر پروسسی بتواند از این دستور استفاده کند، می‌تواند هر کاری را روی سیستم انجام دهد. چون می‌تواند جدول وقفه را طبق نظر خودش تنظیم کند و سپس آدرس حافظه‌ای که به ابتدای آن جدول اشاره می‌کند را در داخل آن رجیستر خاص توسط دستور lidt ثبت کند و در نتیجه بسیاری از مفاهیم و سرویس‌هایی که در مورد سیستم عامل مد نظر داشتیم نقض می‌شود.

```
#include <stdint.h>
```

```
struct IdtRecord
```

```
{
```

```
    uint16_t limit;
```

```
    uintptr_t base;
```

```
};
```

```
int main()
```

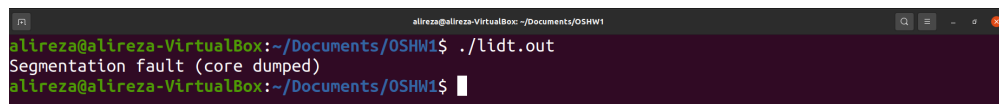
```
{
```

```
    struct IdtRecord* x;
```

```
    __asm__("lidt %0" : : "m" (*x));
```

```
    return 0;
```

```
}
```



شکل ۱: خطای رخ داده پس از اجرای برنامه

با اجرای برنامه بالا مشاهده می‌شود که Segmentation fault (core dumped) رخ می‌دهد. segmentation fault یک خطا است که توسط سخت‌افزار در راستای memory protection، به سیستم عامل اطلاع می‌دهد که یک برنامه تلاش کرده است که به یک بخش حفاظت شده از حافظه دسترسی پیدا کند (a memory access violation). در کامپیوترهای استاندارد x86، این یک فرم از general protection fault است.

۵ عنوان سوال پنجم

۱.۵

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    int n = 0;
    printf("Enter n: ");
    scanf("%d", &n);
    int stat;
    while(n != 1)
    {
        int rc = fork();
        if(rc < 0)
        {
            fprintf(stderr, "Fork Failed!");
        }
        else if(rc == 0)
        {
            int m = n;
            if(m % 2 == 0)
            {
                m /= 2;
            }
            else
```

```

    {
        m = 3 * m + 1;
    }
    return m;
}
else
{
    wait(&stat);
    printf("%d, ", n);
    fflush(stdout);
    n = WEXITSTATUS(stat);
    if(n == 1) printf("%d", n);
}
}
return 0;
}

```

۲.۵

زیرا در UNIX/POSIX، exit code یک برنامه از نوع unsigned 8-bit تعریف شده است. به طور دقیق‌تر system call‌های خانواده wait در UNIX نتیجه یک پروسس را به یک 32-bit integer کدگذاری می‌کنند. از این ۳۲ بیت برای اطلاعاتی همچون وقوع یا عدم وقوع dump core در پروسس، exit به دلیل یک سیگنال (و چه سیگنالی)، و ... تقسیم می‌شود. از این رو تنها ۸ بیت از ۳۲ بیت برای exit code باقی می‌ماند. پس در نتیجه مقادیر ۰ تا ۲۵۵ به این شکل قابل برگردانده شدن هستند.

۳.۵

۶ عنوان سوال ششم

۱.۶

orphan process: پروسسی که parentش وجود ندارد (یا پایان یافته یا بدون اینکه برای متوقف شدن childش صبر کرده باشد، متوقف شده باشد)، orphan process نامیده می‌شود.

zombie process: پروسسی که اجرای آن پایان یافته است اما هنوز در جدول پروسس‌ها مقداری دارد که به پروسس parent نسبت داده می‌شود، zombie process نامیده می‌شود. یک پروسس child همواره پیش از پاک شدن از جدول پروسس‌ها به یک zombie process تبدیل می‌شود. پروسس parent، exit status پروسس child را می‌خواند و پروسس child از جدول پروسس‌ها حذف می‌شود.

۲.۶

برنامه‌ی ۱ یک orphan process ایجاد می‌کند. چون پروسس child حدوداً ۱۰۱ ثانیه پس از پایان یافتن پروسس parent به پایان می‌رسد. همینطور که در تصویر اول مشهود است، مقدار ppid پروسس child پس از ثانیه اول برابر ۲۱۷۴ (pid پروسس parent) است اما پس از ثانیه سوم (پایان یافتن پروسس parent) این مقدار برابر ۱۲۸۶ (pid پروسس systemd که نقش subreaper را برای پروسس orphan برعهده دارد و در قسمت بالای جدول پروسس‌ها قرار دارد) می‌باشد.

```
alireza@alireza-VirtualBox: ~/Documents/OSHW1
alireza@alireza-VirtualBox: ~/Documents/OSHW1$ ./source_1.out
Parent: pid = 2174, ppid = 2158
Child: pid = 2175, ppid = 2174
alireza@alireza-VirtualBox: ~/Documents/OSHW1$ Child: pid = 2175, ppid = 1286
```

شکل ۲: اجرای برنامه‌ی ۱

alireza@alireza-VirtualBox: ~/Documents/OSHW1										alireza@alireza-VirtualBox: ~/Documents/OSHW1									
4	S	1000	1286	1	0	80	0	-	4771	ep_pol	?	00:00:00	systemd						
1	I	0	1287	2	0	80	0	-	0	-	?	00:00:00	kworker/0:4-events						
5	S	1000	1288	1286	0	80	0	-	25822	-	?	00:00:00	(sd-pam)						
0	S	1000	1293	1286	0	69	-11	-	419893	poll_s	?	00:00:00	pulseaudio						
0	S	1000	1296	1286	0	99	-	-	166904	poll_s	?	00:00:00	tracker-miner-f						
0	S	1000	1298	1286	0	80	0	-	2073	ep_pol	?	00:00:00	dbus-daemon						
1	S	1000	1300	1	0	80	0	-	62204	-	?	00:00:00	gnome-keyring-d						
0	S	1000	1306	1286	0	80	0	-	62082	poll_s	?	00:00:00	gvfsd						
0	S	1000	1311	1286	0	80	0	-	95516	futex_	?	00:00:00	gvfsd-fuse						
0	S	1000	1325	1286	0	80	0	-	81516	poll_s	?	00:00:00	gvfs-udisks2-vo						
0	S	1000	1336	1286	0	80	0	-	61652	poll_s	?	00:00:00	gvfs-gphoto2-vo						
0	S	1000	1340	1286	0	80	0	-	61127	poll_s	?	00:00:00	gvfs-goa-volume						
0	S	1000	1345	1286	0	80	0	-	138456	poll_s	?	00:00:00	goa-daemon						
4	S	1000	1347	1273	0	80	0	-	43163	poll_s	tty2	00:00:00	gdm-x-session						
4	S	1000	1352	1347	2	80	0	-	149141	ep_pol	tty2	00:00:24	Xorg						
0	S	1000	1356	1286	0	80	0	-	81810	poll_s	?	00:00:00	goa-identity-se						
0	S	1000	1359	1286	0	80	0	-	81339	poll_s	?	00:00:00	gvfs-afc-volume						
0	S	1000	1367	1286	0	80	0	-	61083	poll_s	?	00:00:00	gvfs-mtp-volume						
0	S	1000	1392	1347	0	80	0	-	49842	poll_s	tty2	00:00:00	gnome-session-b						
1	S	1000	1467	1	0	80	0	-	7811	do_wai	?	00:00:00	VBoxClient						
1	S	1000	1468	1467	0	80	0	-	40845	vbg_hg	?	00:00:00	VBoxClient						
1	S	1000	1479	1	0	80	0	-	7811	do_wai	?	00:00:00	VBoxClient						
1	S	1000	1480	1479	0	80	0	-	40870	vba_co	?	00:00:00	VBoxClient						

شکل ۳: جدول پروسس‌ها (قسمت بالا)

```

alireza@alireza-VirtualBox: ~/Documents/OSHW1
0 S 1000 1713 1286 0 80 0 - 119402 poll_s ? 00:00:00 gsd-sharing
0 S 1000 1714 1286 0 80 0 - 81677 poll_s ? 00:00:00 gsd-smartcard
0 S 1000 1715 1286 0 80 0 - 82604 poll_s ? 00:00:00 gsd-sound
0 S 1000 1724 1286 0 80 0 - 99128 poll_s ? 00:00:00 gsd-usb-protect
0 S 1000 1727 1286 0 80 0 - 89137 poll_s ? 00:00:00 gsd-wacom
0 S 1000 1730 1286 0 80 0 - 81726 poll_s ? 00:00:00 gsd-wwan
0 S 1000 1731 1551 0 80 0 - 198406 poll_s ? 00:00:00 evolution-alarm
0 S 1000 1733 1286 0 80 0 - 89402 poll_s ? 00:00:00 gsd-xsettings
0 S 1000 1737 1551 0 80 0 - 57950 poll_s ? 00:00:00 gsd-disk-utilit
0 S 1000 1795 1286 0 80 0 - 87705 poll_s ? 00:00:00 gsd-printer
0 S 1000 1854 1587 0 80 0 - 43795 poll_s ? 00:00:00 ibus-engine-sim
0 S 1000 1896 1286 0 80 0 - 231076 poll_s ? 00:00:01 nautilus
0 S 1000 1994 1286 0 80 0 - 42737 poll_s ? 00:00:00 gvfsd-metadata
0 S 1000 1999 1551 0 80 0 - 108082 poll_s ? 00:00:00 update-notifier
1 I 0 2098 2 0 80 0 - 0 - ? 00:00:00 kworker/u2:1-events_power_effic
1 I 0 2142 2 0 80 0 - 0 - ? 00:00:00 kworker/0:1-events
0 S 1000 2151 1286 1 80 0 - 206125 poll_s ? 00:00:01 gnome-terminal-
0 S 1000 2158 2151 0 80 0 - 4812 poll_s pts/1 00:00:00 bash
0 S 1000 2164 2151 0 80 0 - 4812 do_wai pts/2 00:00:00 bash
1 S 1000 2175 1286 0 80 0 - 624 hrtime pts/1 00:00:00 source_1.out
0 R 1000 2186 2164 0 80 0 - 5013 - pts/2 00:00:00 ps
alireza@alireza-VirtualBox:~/Documents/OSHW1$

```

شکل ۴: جدول پروسس‌ها (قسمت پایین)

برنامه‌ی ۲ یک zombie process ایجاد می‌کند. چون پروسس parent حدوداً ۹۹ ثانیه پس از پایان یافتن پروسس child به پایان می‌رسد. اگر در این بازه زمانی جدول پروسس‌ها را بررسی کنیم می‌بینیم که پروسسی با Z_STAT وجود دارد که pidش برابر pid پروسس child است و این نشان‌دهنده این است که برنامه‌ی ۲ یک zombie process تولید کرده است. به تصاویر زیر توجه کنید:

```

alireza@alireza-VirtualBox: ~/Documents/OSHW1
alireza@alireza-VirtualBox:~/Documents/OSHW1$ gcc source_2.c -o source_2.out
alireza@alireza-VirtualBox:~/Documents/OSHW1$ ./source_2.out
Parent: pid = 475714, ppid = 475589
Child: pid = 475715, ppid = 475714

```

شکل ۵: اجرای برنامه‌ی ۲


```
alireza@alireza-VirtualBox: ~/Documents/OSHW1$ ps aux | grep 'Z'
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alireza   475715  0.0  0.0      0      0 pts/0    Z+   12:24   0:00 [source_2.out] <defunct>
alireza   475717  0.0  0.0  17540  656 pts/1    S+   12:24   0:00 grep --color=auto Z
alireza@alireza-VirtualBox: ~/Documents/OSHW1$
```

شکل ۶: جدول پروسس‌ها

۷ سوال هفتم

۱.۷ سوال ۶ فایل

با اجرای دستور سوال، یک پروسه دارای ۳ دستور که از IOs و سه پروسه هر کدام دارای ۵ دستور که از CPU استفاده می‌کنند برای اجرا آماده می‌شوند.

خیر-منابع به طور موثر و بهینه به کار گرفته نشده‌اند. همانطور که در تصویر اول می‌بینیم IOs در بازه‌ی زمانی ۷ تا ۱۸ و همچنین CPU در بازه‌ی زمانی ۱۹ تا ۲۳ و ۲۶ تا ۳۰ بی‌کار هستند و درنهایت در ۳۱ واحد زمانی CPU تنها ۶۷/۷۴ درصد مواقع، و IOs تنها ۴۸/۳۹ درصد مواقع مشغول بوده‌اند. به عبارت دیگر در ۳۱ واحد زمانی، CPU، ۲۱ واحد زمانی و IOs، ۱۵ واحد زمانی در حال استفاده بوده‌اند.

```
alireza@alireza-VirtualBox: ~/Documents/OSHW1/HW-CPU-Intro$ python3 process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p
Time  PID: 0  PID: 1  PID: 2  PID: 3  CPU  IOs
1     RUN:io  READY  READY  READY  1    1
2     WAITING RUN:cpu  READY  READY  1    1
3     WAITING RUN:cpu  READY  READY  1    1
4     WAITING RUN:cpu  READY  READY  1    1
5     WAITING RUN:cpu  READY  READY  1    1
6     WAITING RUN:cpu  READY  READY  1    1
7*    READY  DONE    RUN:cpu  READY  1    1
8     READY  DONE    RUN:cpu  READY  1    1
9     READY  DONE    RUN:cpu  READY  1    1
10    READY  DONE    RUN:cpu  READY  1    1
11    READY  DONE    RUN:cpu  READY  1    1
12    READY  DONE    RUN:cpu  READY  1    1
13    READY  DONE    RUN:cpu  READY  1    1
14    READY  DONE    RUN:cpu  READY  1    1
15    READY  DONE    RUN:cpu  READY  1    1
16    READY  DONE    RUN:cpu  READY  1    1
17    RUN:io_done DONE    DONE    DONE    1    1
18    RUN:io  DONE    DONE    DONE    1    1
19    WAITING DONE    DONE    DONE    1    1
20    WAITING DONE    DONE    DONE    1    1
21    WAITING DONE    DONE    DONE    1    1
22    WAITING DONE    DONE    DONE    1    1
23    WAITING DONE    DONE    DONE    1    1
24*   RUN:io_done DONE    DONE    DONE    1    1
25    RUN:io  DONE    DONE    DONE    1    1
26    WAITING DONE    DONE    DONE    1    1
27    WAITING DONE    DONE    DONE    1    1
28    WAITING DONE    DONE    DONE    1    1
29    WAITING DONE    DONE    DONE    1    1
30    WAITING DONE    DONE    DONE    1    1
31*   RUN:io_done DONE    DONE    DONE    1    1

Stats: Total Time 31
Stats: CPU Busy 21 (67.74%)
Stats: IO Busy 15 (48.39%)
alireza@alireza-VirtualBox: ~/Documents/OSHW1/HW-CPU-Intro$
```

شکل ۷:

۲.۷ سوال ۷ فایل

تفاوتی که این حالت با حالت قبل دارد این است که عملکرد CPU و IOs در بازه‌های بیشتری همزمان به موازات یکدیگر رخ می‌دهند. درواقع در این حالت پس از پایان یافتن IO، بلافاصله به این پروسس سوئیچ می‌کند و در نتیجه ۵ دستور اجرای IO (waiting)، با ۵ دستور اجرای CPU موازی می‌شود. با دقت در تصویر دوم می‌بینیم که زمان کل به ۲۱ واحد کاهش یافته است، CPU در کل ۲۱ واحد مشغول بوده است، و IOs نیز در ۱۵ واحد در حال استفاده بوده است (درست است که در هر دو حالت، IOs در ۱۵ واحد زمانی استفاده شده است، اما در حالت قبل، IOs تنها در ۴۸/۳۹ درصد مواقع مشغول بوده است، درحالی که در این حالت در ۷۱/۴۳ درصد مواقع در حال استفاده بوده است. پس در این سوال با شرایط ذکر موجود، اجرای بلافاصله پروسسی که اخیراً I/O ای آن کامل شده است ایده خوبی می‌تواند باشد.

```

alireza@alireza-VirtualBox: ~/Documents/OSHW1/HW-CPU-Intro$ python3 process-run.py -l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_IMMEDIATE -c -p
Time  PID: 0      PID: 1      PID: 2      PID: 3      CPU      IOs
1      RUN:to      READY      READY      READY      1
2      WAITING    RUN:cpu     READY      READY      1
3      WAITING    RUN:cpu     READY      READY      1
4      WAITING    RUN:cpu     READY      READY      1
5      WAITING    RUN:cpu     READY      READY      1
6      WAITING    RUN:cpu     READY      READY      1
7*     RUN:to_done DONE        READY      READY      1
8      RUN:to     DONE        READY      READY      1
9      WAITING    DONE        RUN:cpu     READY      1
10     WAITING    DONE        RUN:cpu     READY      1
11     WAITING    DONE        RUN:cpu     READY      1
12     WAITING    DONE        RUN:cpu     READY      1
13     WAITING    DONE        RUN:cpu     READY      1
14*    RUN:to_done DONE        DONE        READY      1
15     RUN:to     DONE        DONE        READY      1
16     WAITING    DONE        DONE        RUN:cpu     1
17     WAITING    DONE        DONE        RUN:cpu     1
18     WAITING    DONE        DONE        RUN:cpu     1
19     WAITING    DONE        DONE        RUN:cpu     1
20     WAITING    DONE        DONE        RUN:cpu     1
21*    RUN:to_done DONE        DONE        DONE        1

Stats: Total Time 21
Stats: CPU Busy 21 (100.00%)
Stats: IO Busy 15 (71.43%)
alireza@alireza-VirtualBox: ~/Documents/OSHW1/HW-CPU-Intro$

```

شکل ۸:

۸ عنوان سوال جدول

در این قسمت با نحوه درج جداول آشنا می‌شوید:

خانه شماره ۳	خانه شماره ۲	خانه شماره ۱
خانه شماره ۶	خانه شماره ۵	خانه شماره ۴
خانه شماره ۹	خانه شماره ۸	خانه شماره ۷

جدول ۱: جدول شماره ۱

۹ عنوان سوال هشتم

در این قسمت با نحوه ارجاع به سایر منابع آشنا می‌شوید:

به صفحه درس سیستم عامل دکتر محمد رضا حیدرپور ارجاع داده می‌شود [۱۱].

۱۰ ضمیمه

برای آشنایی بیشتر با \LaTeX با جست و جو در اینترنت منابع مفیدی خواهید یافت.

منابع

[1] http://mrheidar.ir/courses/operating_system.html