

Sharif-OS-Lab /
session-5-6

<> Code

Issues 4

Pull requests

Actions

Projects

Security

Insights

session-5-6 / session5.md



sina-imani Fix typos, change signal to sigaction

1ba0ff3 · last year



171 lines (132 loc) · 9.43 KB

آزمایش ۵ - ارتباط بین پردازش‌ها

۵.۱ مقدمه

در این جلسه از آزمایشگاه مکانیزم‌های مربوط به ارتباط و تبادل پیام بین پردازش‌ها در سیستم عامل لینوکس را خواهیم آموخت.

۵.۱.۱ پیش‌نیازها

انتظار می‌رود که دانشجویان با موارد زیر از پیش آشنا باشند:

1. نحوه‌ی ایجاد پردازش‌ها در سیستم عامل لینوکس (مطالب جلسه‌ی چهارم)
2. برنامه‌نویسی به زبان C++
3. دستورات پوسته‌ی لینوکس که در جلسات قبل فرا گرفته شده‌اند.

۵.۲ ارتباط بین پردازش‌ها

در جلسات قبل نحوه‌ی ایجاد پردازش‌های جدید را آموختیم. در این جلسه سعی داریم روش‌های ارتباط میان این پردازش‌ها را بررسی کنیم. مکانیزم‌های متعددی برای تبادل پیام بین پردازش‌ها وجود دارد که در این جلسه دو روش استفاده از Pipe و Signal را بررسی خواهیم کرد. از جمله کاربردهای ارتباط بین پردازش‌ها می‌توان به همگام‌سازی و انتقال اطلاعات اشاره کرد.

دستورات Pipe برای کاربران پوستره ی لینوکس آشنا هستند. برای مثال شما می‌توانید برای مشاهده ی لیست پردازیهایی که در آن‌ها کلمه ی init وجود دارد از دستور `ps aux | grep init` استفاده کنید. در اینجا دو پردازیه به کمک یک Pipe به هم متصل شده‌اند. نکته‌ای که در اینجا مهم است آن است که این Pipe ایجاد شده تنها در یک جهت (از پردازیه ی اول به پردازیه ی دوم) اطلاعات را جابه‌جا می‌کند. به کمک فراخوانی‌های سیستمی می‌توان Pipe های دو سویه و حلقوی نیز ایجاد کرد.

۵.۳ شرح آزمایش

۵.۳.۱ ایجاد یک Pipe یک‌سویه

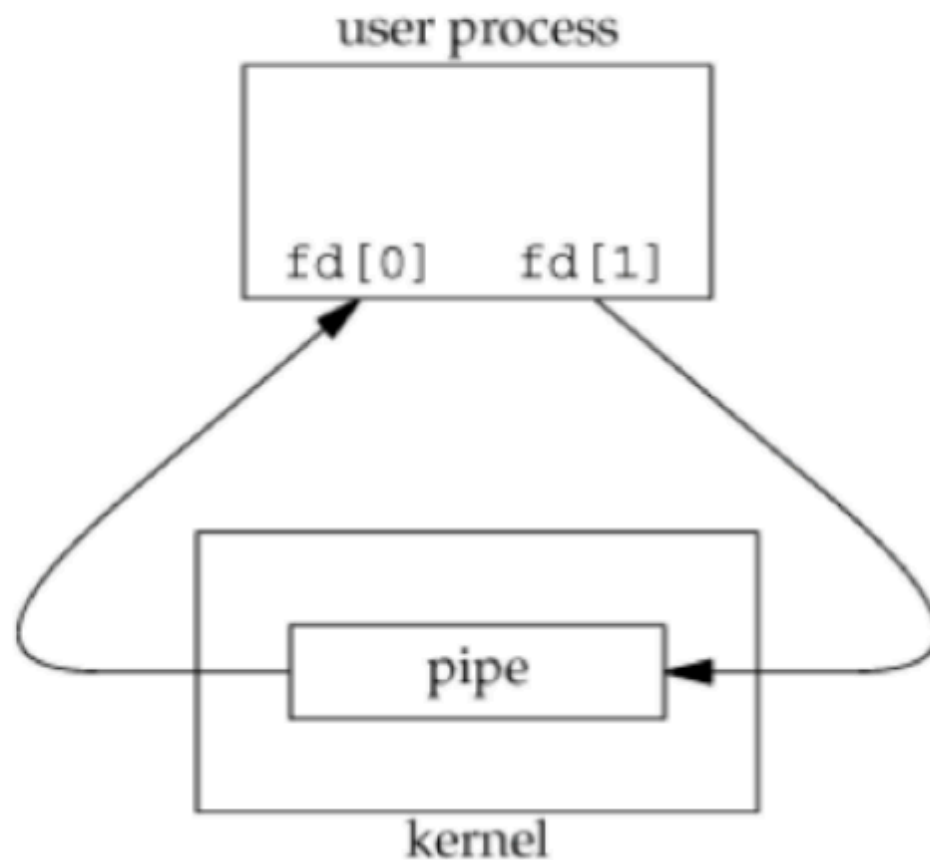
- (آ) برای ایجاد Pipe یک سویه در سیستم عامل لینوکس از فراخوانی سیستمی `pipe` استفاده می‌شود. به کمک دستور `man 2 pipe` خلاصه‌ای از نحوه‌ی کار آن را ملاحظه کنید.
- (ب) به کمک کد زیر یک Pipe ایجاد کنید.

```
int fd[2];  
int res = pipe(fd);
```



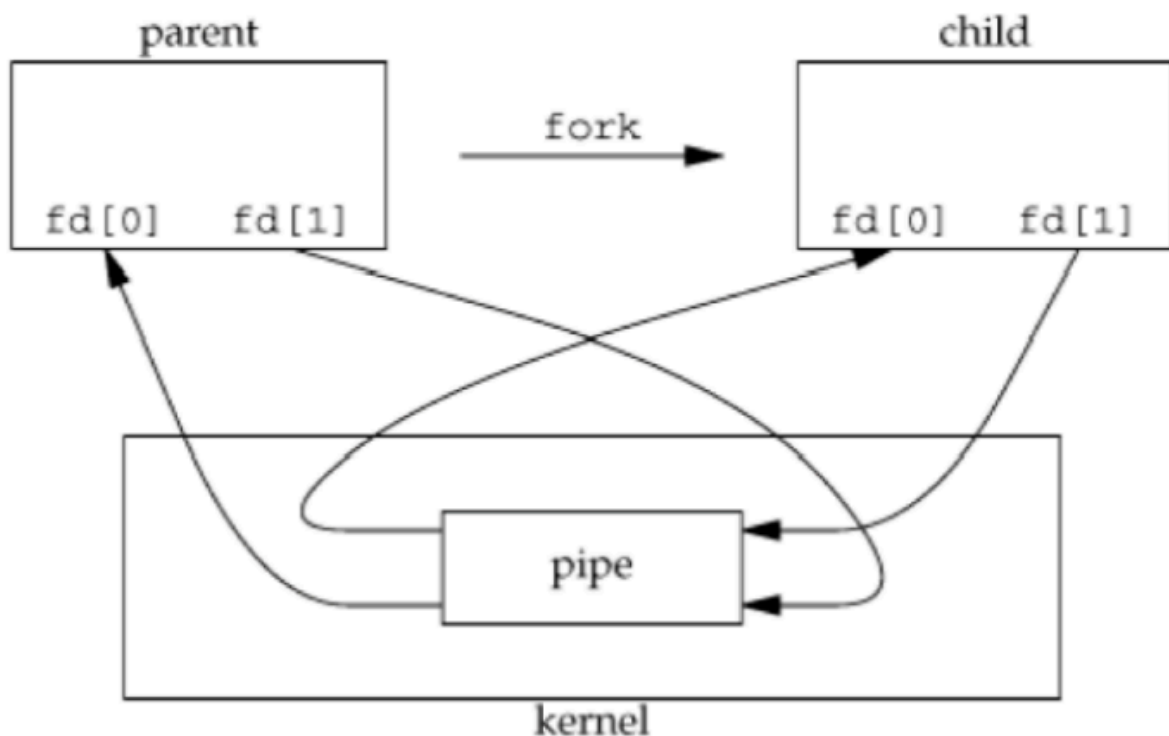
دستور `pipe` در اینجا دو File Descriptor ایجاد می‌کند (آرایه ی `fd`). یکی از آن‌ها برای خواندن و دیگری برای نوشتن مورد استفاده قرار خواهد گرفت. `fd[0]` برای خواندن و `fd[1]` برای نوشتن خواهد بود.

- (ج) تا اینجا تنها یک پردازیه داریم و می‌توان شمای کلی `fd`های ایجاد شده را در شکل ۵.۱ نشان داد:



شکل ۱۰.۵: Pipe یک سویه

هر چیزی که بر روی `fd[1]` نوشته شود، قابل خواندن با `fd[0]` خواهد بود. حال توجه کنید که در صورتی که عملیات `fork` انجام گیرد، پردازشی فرزند، `File Descriptor` های پدر را به ارث خواهد برد. بنابراین، بعد از انجام شدن عملیات `fork` و ایجاد پردازشی فرزند، ساختار بالا به شکل ۵.۲ در خواهد آمد.



شکل ۲.۵: Pipe دو سویه

مشکل مهمی که در اینجا با آن مواجه هستیم آن است که در صورتی که هر دو پردازش بخواهند بر روی Pipe بنویسند و یا از آن بخوانند، به دلیل اینکه تنها یک بافر مشترک داریم، رفتار سیستم قابل پیش بینی نخواهد بود. در این حالت یک پردازش ممکن است داده‌ای که خودش بر روی Pipe قرار داده است را بخواند! بنابراین نیاز است که یک طرف تنها بر روی Pipe بنویسد و یک طرف تنها از آن بخواند. برای مثال فرض کنید پردازش فرزند قصد خواندن از Pipe و پردازش والد قصد نوشتن بر روی آن را دارد. به کمک فراخوانی سیستمی `close`، پردازش والد `fd[0]` خود را می‌بندد (زیرا قصد خواندن ندارد) و پردازش فرزند نیز `fd[1]` را خواهد بست. به این ترتیب یک ارتباط Half-Duplex بین این دو پردازش ایجاد می‌شود.

فعالیت‌ها

session-5-6 / session5.md

↑ Top

Preview

Code

Blame

Raw

📄

⬇️

✎

⌵

☰

می‌توان یک برنامه، مثلا `ls` را اجرا نمود. به کمک دستورات `dup/aup2` برنامه‌ای بنویسید که پردازش والد دستور `ls` و پردازش فرزند دستور `wc` را اجرا کند و خروجی پردازش والد (دستور `ls`) به عنوان ورودی به پردازش فرزند داده شود.

○ (راهنمایی: یک Pipe ایجاد کنید به نحوی که خروجی پردازش ی والد ورودی Pipe باشد و خروجی Pipe به عنوان ورودی پردازش ی فرزند باشد. اینکار با استفاده از دستورات `dup/dup2` ممکن است.)

○ (راهنمایی ۲: خروجی برنامه ی `ls` در `stdout` قرار می‌گیرد که یک File Descriptor با شماره ی ۱ است. ورودی برنامه `wc` از `stdin` است که یک File Descriptor با شماره ی ۰ است.)

- بررسی کنید که چطور ارتباطات تمام دو طرفه بین پردازش‌ها داشته باشیم.

۵.۳.۲ سیگنال‌ها

بعضی اوقات نیاز است که برنامه‌ها بتوانند با برخی از شرایط غیر قابل پیش بینی مواجه شده، آن‌ها را کنترل کنند. برای مثال:

- درخواست بستن برنامه توسط کاربر به وسیله ی `Ctrl + C`
- رخ دادن خطا در محاسبات Floating Point
- مرگ پردازش ی فرزند

این رخدادها توسط سیستم عامل لینوکس شناخته می‌شوند و سیستم عامل با ارسال یک سیگنال، پردازش را از وقوع آن‌ها آگاه می‌سازد. برنامه نویس می‌تواند این سیگنال‌ها را نادیده بگیرد، یا در عوض با نوشتن کد آن‌ها را مدیریت و کنترل نماید.

- به کمک دستور `man 7 signal` لیستی از سیگنال‌های موجود در سیستم عامل لینوکس را مشاهده کنید. سیگنال‌های زیر را توضیح دهید.

SIGINT, SIGHUP, SIGSTOP, SIGCONT, SIGKILL

- [Information about signals](#)
- [Nore information about Two signals: SIGSTOP and SIGCONT](#)
- یک سیگنال ساده، سیگنال Alarm (SIGALRM) است. به کمک دستور `man` در مورد آن توضیح کوتاهی ارائه دهید.
- کد زیر به کمک این سیگنال نوشته شده است. آن را اجرا کرده و در مورد کارکرد آن توضیح دهید.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    alarm (5);
    printf ("Looping forever . . . \n");
    while (1);
    printf("This line should never be executed\n");
}
```



```
    return 0;  
}
```

- به طور پیش فرض پردازش بعد از دریافت یکی از سیگنال‌های تعریف شده، کشته می‌شود. به کمک تابع `sigaction` می‌توان این رفتار را تغییر داد و کد مورد نظر برنامه نویس را اجرا کرد. همچنین یک تابع دیگر به نام `pause` وجود دارد که پردازش را تا زمانی که یک سیگنال دریافت کند، متوقف می‌سازد (واضح است که این توابع از فراخوانی‌های سیستمی برای انجام کار خود استفاده می‌کنند). به کمک این دو تابع، برنامه‌ی بالا را به گونه‌ای ویرایش کنید که بعد از دریافت سیگنال `SIGALRM` از توقف خارج شود و خط آخر را در خروجی چاپ کند.
- برنامه‌ای بنویسید که در صورتی که کاربر کلیدهای `Ctrl + C` را فشار دهد، برای بار اول خارج نشود و پیامی برای فشار دادن دوباره ی آن‌ها چاپ کند. در دفعه ی دوم رخ دادن سیگنال، برنامه به پایان برسد.