سیستم‌های عامل - دکتر ابراهیمی‌مقدم

امیرحسین منصوری - ۹۹۲۴۳۰۶۹

تمرین سری دوم

سوال ۱

الف) *نادرست*؛ PCB در فضای آدرس کرنل نگهداری می‌شود تا کرنل بتواند به آن دسترسی داشته باشد.

ب) *نادرست*؛ الزاما چنین نیست، و ممکن است پروسس فرزند بتواند از سیستم‌عامل منابع دریافت کند.

ج) *نادرست*؛ تعداد پروسس‌هایی که می‌توانند به صورت همروند روی یک هسته پردازنده اجرا شوند را Long-term Scheduler مشخص می‌کند. Short-term Scheduler صرفا بین پروسه‌های موجود که Ready هستند، یکی را برای اجرا انتخاب می‌کند.

سوال ۲

پروسه پس از ساخته شدن در وضعیت new است. پس از پذیرش توسط Long-term scheduler، پروسه در وضعیت ready قرار می‌گیرد. زمانی که Short-term scheduler به پروسه CPU اختصاص دهد، پروسه در وضعیت running قرار می‌گیرد. با فرض این که پروسه تا جایی که بتواند CPU را برای خودش نگه می‌دارد، تا اجرای خط ۴ کد، پروسه در همین وضعیت running باقی می‌ماند.

در خط ۵، عملیات IO رخ می‌دهد؛ بنابراین در این خط، پروسه به وضعیت waiting می‌رود. وقتی که عملیات IO به پایان برسد، پروسه به وضعیت ready رفته و بعد از مدتی توسط Short-term scheduler به وضعیت running باز می‌گردد و اجرای برنامه ادامه پیدا می‌کند.

تا پایان اجرای برنامه، پروسه در همین وضعیت باقی می‌ماند، و در نهایت پس از اتمام برنامه، به وضعیت terminated می‌رود.

سوال ۳

- **پیچیدگی پیاده‌سازی:** استفاده از Message passing برای برنامه‌نویس ساده‌تر است؛ چون سازوکار ارسال و دریافت پیام در کرنل پیاده‌سازی شده است و برنامه‌نویس تنها از آن استفاده می‌کند. اما در روش Shared memory، برنامه‌نویس باید خودش سازوکار موردنظر را پیاده کند.
- **سرعت:** در روش Shared memory، تنها نوشتن در حافظه انجام می‌شود و نیازی به استفاده از System-callهای سیستم‌عامل نیست. اما برای Message passing معمولا از System-callهای سیستم‌عامل استفاده می‌شود که کندتر است. بنابراین روش Shared memory سریعتر است.

برای پیاده‌سازی Message passing، پروسه‌ای که می‌خواهد داده‌ای به اشتراک بگذارد، آن را با استفاده از System call مربوطه برای پروسه با pid مشخص می‌فرستد:

```
send(pid, msg);
```

سپس پروسه مربوطه، پیام مورد نظر را با System call مربوطه دریافت می‌کند:

```
receive(msg);
```

این ارسال و دریافت می‌تواند به صورت مستقیم یا غیر مستقیم انجام شود.

در روش Shared memory، پروسه‌ای که می‌خواهد داده به اشتراک بگذارد، این داده را در یک بافر مشترک در حافظه می‌نویسد، و پروسه‌ای که می‌خواهد داده را دریافت کند، آن را از این بافر مشترک می‌خواند. یک سازوکار برای مدیریت دسترسی به این بافر مشترک، پیاده کردن یک صف FIFO است. پر بودن این صف نشانه وجود داده است و در این صورت، پروسه دریافت کننده می‌تواند داده را از سر صف بردارد. در صورت خالی بودن صف، پروسه باید منتظر داده بماند.

سوال ۴

- Short-term scheduler: از صف پروسه‌های Ready (یا همان Ready queue)، در هر زمان و بنا به معیارهایی، یک پروسه را برای اجرا روی پردازنده انتخاب می‌کند و به آن CPU Time می‌دهد. در بازه‌های زمانی کوتاه اجرا می‌شود و بنابراین باید سریع باشد.
- Long-term scheduler: مشخص می‌کند کدام پروسه‌ها می‌توانند وارد Ready queue شوند تا بعدا بتوانند توسط پردازنده اجرا شوند. صرفا در زمان درست‌شدن پروسه جدید و در بازه‌های زمانی بلندتری اجرا می‌شود. همچنین درجه Multi-programming را نیز مشخص می‌کند.
- Medium-term scheduler: پروسه‌هایی که در انتظار هستند (مثلا منتظر عملیات IO) را از Ready queue خارج می‌کند. همچنین در تعیین درجه Multi-programming نقش دارد.

سوال ۵

سیستم‌عامل برای Context switch، اطلاعات PCB پروسه را مثل مقدارهای فعلی رجیسترهای پردازنده، وضعیت فعلی پروسه، و... ذخیره می‌کند.

زمانی که پروسه یک عملیات IO انجام می‌دهد، به وضعیت waiting می‌رود و این باعث می‌شود که یک Context-switch رخ دهد؛ و به این ترتیب یک پروسه می‌تواند خودش باعث Context-switch شود. همچنین درصورتی که Scheduler تصمیم بگیرد پردازنده را از یک پروسه بگیرد، Context-switch بدون اختیار پروسه رخwt می‌دهد.

سوال ۶

الف)

در پروسس فرزند، `fork` مقدار صفر برمی‌گرداند. بنابراین مقدار `x` همان مقدار اولیه خود که برابر ۷ است را حفظ می‌کند و عدد ۷ چاپ می‌شود.

ب)

بله؛ زیرا ممکن است (و البته طبق فرض سوال حتما این اتفاق رخ می‌دهد) که پروسه والد کار خود را تمام کند و terminate شود، و پروسه فرزند همچنان تمام نشده باشد. با اضافه کردن خط زیر، بعد از صدا کردن `fork` (مثلا دقیقا بعد از خط ۲)، این مشکل برطرف می‌شود.

```
wait(NULL);
```

خط بالا باعث می‌شود تا تمام شدن همه‌ی پروسه‌های فرزند، اجرای پروسه‌ی فعلی (والد) متوقف شود.

ج)

در سیستم‌عامل‌های یونیکسی، در هنگام orphan شدن یک پروسه، کرنل یک پروسه از پیش مشخص‌شده را به عنوان والد این پروسه انتخاب می‌کند. این پروسه در هر پیاده‌سازی متفاوت است؛ اما پروسه init (که اولین پروسه اجرا شده هنگام روشن شدن سیستم است و تا آخر خاموش‌شدن آن اجرا می‌شود) یکی از انتخاب‌های محبوب بوده است.

سوال ۷

سوال اصلی

به چند نکته دقت می‌کنیم:

۱. اولویت عملگر && از || بالاتر است.

۲. هر دو عملگر از چپ به راست محاسبه می‌شوند.

۳. این دو عملگر خاصیت اتصال کوتاه دارند؛ یعنی در صورت `true` بودن سمت چپ عملگر || یا `false` بودن سمت چپ عملگر &&، عملوند سمت راست دیگر محاسبه نمی‌شود.

با توجه به نکته اول، می‌توان خط اول تابع `main` را به شکل خواناتر زیر بازنویسی کرد:

```
fork() || (fork() && fork()) || (fork() && fork()) || fork()
```

سیستم‌کال `fork`، باعث تولید یک پروسه جدید می‌شود. هر پروسه جدید، یک بار عبارت `"yo"` را چاپ می‌کند. پس باید تعداد پروسه‌های ایجاد شده را بشماریم. در ادامه فرض می‌کنیم PID پروسه والد (که از هیچ پروسه دیگری `fork` نشده است) برابر ۱ است.

روند اجرای این خط از کد برای هر پروسه ایجاد شده با PID مشخص شده به صورت زیر است.

پروسه ۱: اولین `fork` اجرا می‌شود و مقداری غیر صفر برمی‌گرداند. فرض می‌کنیم PID پروسه جدید ۲ است و مقدار ۲ برگردانده می‌شود.

```
2 || (fork() && fork()) || (fork() && fork()) || fork()
```

به دلیل اتصال کوتاه، `fork` دوم و سوم اجرا نمی‌شوند.

```
1 || (fork() && fork()) || fork()
```

به طور مشابه، `fork` چهارم و پنجم نیز اجرا نمی‌شوند.

```
1 || fork()
```

همچنین `fork` آخر نیز اجرا نمی‌شود. در نهایت تنها پروسه ۲ ایجاد می‌شود.

پروسه ۲: اولین `fork` صفر برمی‌گرداند؛ زیرا در پروسه فرزند هستیم.

```
0 || (fork() && fork()) || (fork() && fork()) || fork()
```

در نتیجه `fork` دوم و سوم اجرا می‌شوند و پروسه‌های ۳ و ۴ را ایجاد می‌کنند.

```
0 || (3 && 4) || (fork() && fork()) || fork()
1 || (fork() && fork()) || fork()
```

مشابه روند پروسه ۱، می‌توان دید که بقیه `fork`ها اجرا نمی‌شوند. در نهایت پروسه‌های ۳ و ۴ ایجاد می‌شوند.

پروسه ۳: وضعیت اجرا در ابتدا به صورت زیر است.

```
0 || (0 && fork()) || (fork() && fork()) || fork()
```

به دلیل اتصال کوتاه، `fork` سوم اجرا نمی‌شود.

```
0 || (fork() && fork()) || fork()
```

fork چهارم و پنجم اجرا شده و پروسه‌های ۵ و ۶ را ایجاد می‌کنند.

```
(5 && 6) || fork()
1 || fork()
```

باز به دلیل اتصال کوتاه، fork آخر اجرا نمی‌شود. در نهایت پروسه‌های ۵ و ۶ ایجاد می‌شوند.

پروسه ۴: روند اجرای این پروسه بسیار مشابه پروسه ۳ است. در اینجا نیز fork چهارم و پنجم اجرا شده و پروسه‌های ۷ و ۸ را ایجاد می‌کنند.

پروسه ۵: وضعیت اجرا به صورت زیر است.

```
(0 && fork()) || fork()
```

به دلیل اتصال کوتاه، fork پنجم اجرا نمی‌شود. به سادگی می‌توان دید که fork آخر اجرا می‌شود و پروسه ۹ را ایجاد می‌کند.

پروسه ۶: مشابه پروسه ۵، پروسه ۱۰ نیز در اینجا ایجاد می‌شود.

پروسه ۷: کاملا مشابه پروسه ۵، پروسه ۱۱ نیز در اینجا ایجاد می‌شود.

پروسه ۸: کاملا مشابه پروسه ۶، پروسه ۱۲ نیز در اینجا ایجاد می‌شود.

پروسه ۹ تا ۱۲: در این پروسه‌ها اجرای تمام forkها تمام شده است. بنابراین پروسه جدیدی ایجاد نمی‌شود.

در نتیجه در کل ۱۲ پروسه اجرا می‌شوند و عبارت yo، ۱۲ بار چاپ می‌شود.

سوال اصلاح‌شده

مشابه سوال قبل، وضعیت هر پروسه را بررسی می‌کنیم.

پروسه ۱: fork اول و دوم اجرا می‌شوند و پروسه‌های ۲ و ۳ را ایجاد می‌کنند.

```
2 && !fork() || fork();
2 && !(3) || fork();
0 || fork();
```

fork آخر نیز اجرا شده و پروسه ۴ را ایجاد می‌کند. در نهایت پروسه‌های ۲، ۳ و ۴ ایجاد می‌شوند.

پروسه ۲: fork اول مقدار صفر برمی‌گرداند.

```
0 && !fork() || fork();
```

به دلیل اتصال کوتاه، fork دوم اجرا نمی‌شود.

```
0 || fork();
```

در نهایت، fork آخر هم اجرا می‌شود و پروسه ۵ را ایجاد می‌کند.

پروسه ۳ تا ۵: در این پروسه‌ها، اجرای تمام forkها تمام شده و پروسه جدیدی ایجاد نمی‌شود.

در کل ۵ پروسه اجرا می‌شوند؛ پس عبارت yo، ۵ بار چاپ می‌شود.

سوال ۸

exec

خانواده system-callهای `exec` (مثل `execve`، `execv`، و...) یک برنامه ذخیره‌شده در دیسک را اجرا می‌کنند و آن را جایگزین برنامه اجرا شده در پروسه فعلی می‌کنند؛ بدون این که پروسه جدیدی ایجاد کنند. با استفاده از این system-call، می‌توان یک برنامه دیگر را اجرا کرد. مثلاً shellهایی مثل bash یا zsh، با استفاده از این system-call دستورات کاربر را که در بسیاری از موارد شامل اجرای یک برنامه روی دیسک است اجرا می‌کنند.

fork

یک پروسه جدید درست می‌کند که یک کپی از پروسه فعلی است؛ به این معنی که دقیقا دارای همان کد و همان محتویات حافظه است و در پروسهٔ جدید، اجرای برنامه دقیقا از همان جایی که `fork` صدا زده شده ادامه پیدا می‌کند. در صورت اجرای موفقیت‌آمیز، این تابع در پروسه والد pid پروسهٔ ایجاد شده، و در پروسهٔ فرزند مقدار صفر را برمی‌گرداند. با بررسی مقدار بازگشتی این تابع می‌توان پی برد که ادامهٔ کد در کدام‌یک از پروسه‌های والد یا فرزند اجرا می‌شود.

هر زمان که نیاز داشته باشیم یک پروسه جدید ایجاد کنیم، از `fork` استفاده می‌کنیم. برنامه‌های shell نیز هنگام اجرای دستورات کاربر، ابتدا یک fork ایجاد می‌کنند، و سپس دستور کاربر را در پروسهٔ فرزند اجرا می‌کنند تا پروسه‌ای که خود shell را اجرا می‌کند از بین نرود.

wait

تا تمام شدن یک پروسهٔ فرزند (یا همه پروسه‌های فرزند)، پروسهٔ والد را در حالت انتظار (waiting) نگه می‌دارد و اجرای آن را متوقف می‌کند.

در موارد بسیاری، نیاز داریم تا تمام شدن کار پروسهٔ فرزند صبر کنیم. مثلا در یک برنامه shell، پس از اجرای یک دستور، باید اجرای آن (که در پروسهٔ فرزند به طور جداگانه اجرا می‌شود) تمام شود تا shell دوباره منتظر وارد شدن دستور بعدی باشد.