سوال ۱)

برای قسمت

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

با توجه به موارد زیر درخت تصمیم را رسم و تعداد fork ها را به دست می آوریم:

- در دستورات شرطی اگر چندین statement باهم & شده باشند در صورت اولین مشاهده 0، condition های بعدی دیگر بررسی نمی شود.
- در دستورات شرطی اگر چندین statement باهم || شده باشند در صورت اولین مشاهده 1، condition های بعدی دیگر بررسی نمی شود.

(تعداد پردازه ها تا این مرحله 5)
در ادامه ی کد یک حلقه داریم، که بدون در نظر گرفتن هیچ شرط خاصی دستور سیستمی fork را صدا می زند. بنابراین در هر بار اجرای حلقه تعداد fork ها 2 برابر می شود. با توجه به اینکه 3 پیمایش در حلقه انجام می هیم در کل 40=5*8 پردازه در آخر داریم.

سوال ۲)

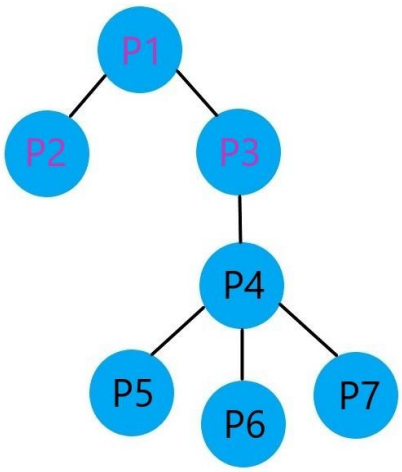
میدانیم که متغیر value از نوع global است. یعنی تنها میان threadها (و نه پردازه‌ها) مشترک است. در خط چهارم تابع main، پردازه فرزند جدید ساخته میشود.
پردازه فرزند: وارد بلاک if شده و یک pthread ساخته میشود. Control thread منتظر terminate شدن thread جدید میماند. Thread جدید مقدار value را به ۵ تغییر می‌دهد. همانطور که در بالا گفته شد، این متغیر بین دو thread یکسان است. پس در خط C، مقدار ۵ پرینت میشود.
پردازه والد: والد بلاک else میشود و با دستور wait منتظر exit شدن پردازه فرزند میمانیم. دقت شود که value میان دو پردازه، share نمیشود. درنتیجه، مقدار صفر در خط P پرینت میشود.

سوال ۳)

بیشترین مقدار: thread اول مقدار count = 0 را دریافت کند و در تابع test، آن را به مقدار MAX افزایش دهد. سپس thread دوم مقدار count = MAX را دریافت کرده و مقدار آن را به 2MAX افزایش دهد (درواقع عملیات خواندن و نوشتن بدون تداخل انجام شود). در نهایت مقدار 2MAX چاپ شود.
کمترین مقدار: هر دو thread مقدار count = 0 را با هم بخوانند. thread اول به سرعت محاسبه را انجام دهد تا جایی که $MAX - 1$ بار حلقه ایجاد شده و در حال حاضر همین مقدار در متغیر count نوشته شده است. حالا thread دوم مقدار اولیه برابر صفر که خوانده بود را با ۱ جمع کرده و نتیجه که عدد ۱ است را در متغیر count ذخیره می‌کند (بر روی مقدار قبلی که $MAX - 1$ است). مجددا هر دو مقدار متغیر count که برابر ۱ شده است را می‌خوانند. این بار، thread اول در آخرین بار اجرای حلقه گیر میکند ولی دومی تا انتهای حلقه میرود و مقدار MAX را در متغیر count

مینویسد. در نهایت thread اول نیز آخرین مرحله را انجام میدهد و ۱+۱=۲ را در این متغیر میریزد. در نتیجه مینیمم برابر عدد ۲ است.

سوال ۴) فرض کنید پردازه‌های موجود در درخت فرآیند فوق را مانند زیر نامگذاری کنیم.



حال می‌توان با شبکه کد زیر این درخت را با شرایط گفته شده به وجود آورد.

```
int pid = fork();
if (pid > 0) {   // P1
    pid = fork();
    if (pid == 0) {  // P3
        pid = fork();
        if (pid == 0) {  // P4
            pid = fork();
            if (pid == 0)  // P5
                exec("ls")
            pid = fork();
            if (pid == 0)  // P6
                exec("sort")
            pid = fork();
            if (pid == 0)  // P7
                exec("search")
            wait();
            wait();
            wait();
        }
    }
```

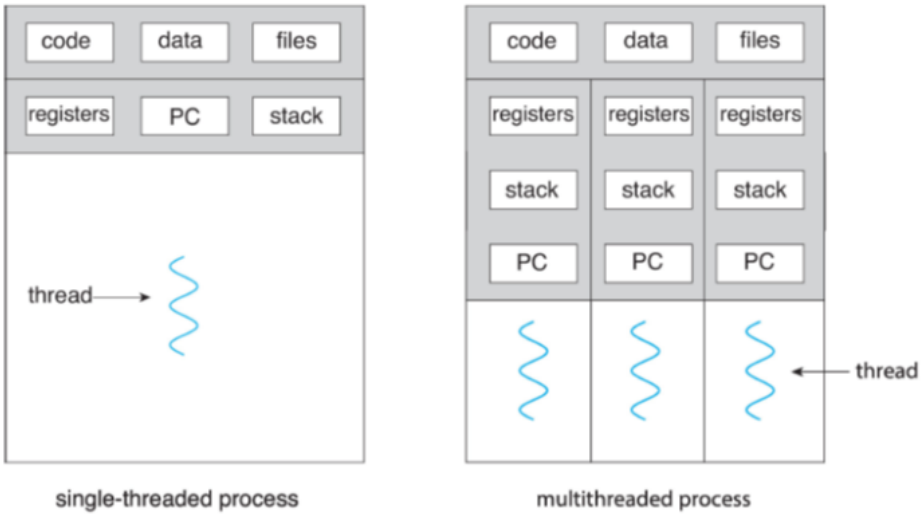
}

* برنامهٔ فوق یک نمونه از جواب صحیح بوده و هر شبه کدی که عملکرد مشابه داشته باشد قابل قبول است.

سوال ۵)

در هنگام ساخت یک پردازه قسمتی از حافظه برای ساخت PCB به آن اختصاص می یابد. هر ریسمان یک TCB دارد که درون آن یک پوینتر به PCB پردازنده ی آن است و code و data و file ها بین تمامی ریسمان های یک پردازه مشترک است و از طریق PCB پردازه به آن دسترسی دارند. پس منابع جدید مورد نیاز برای ساخت یک ریسمان نسبت به ساخت یک پردازه بسیار کمتر است.

سوال ۶) در شکل زیر اطلاعات ذخیره شده برای یک پردازهٔ مجزا و تعدادی ریسمان مربوط به یک پردازه نمایش داده شده‌است.



همانطور که در این شکل مشاهده می‌شود، چیزی که ریسمان‌های مربوط به یک پردازه را از یکدیگر مجزا می‌سازد مقادیر درون رجیسترها، فضای استک مربوط به ریسمان و Program Counter برنامه است. بنابراین زمانی که عمل Context Switch میان دو ریسمان مربوط به یک پردازه انجام می‌شود، تنها کافیست همین اطلاعات ذخیره و بازیابی شوند.

از طرفی یک پردازهٔ مجزا نسبت به پردازهای دیگر، به طور کلی فضای حافظه و منابع مجزایی دارد. بنابراین در زمان Context Switch میان دو پردازه علاوه بر موارد اشاره شده در بالا، فضای آدرس برنامه نیز تغییر خواهد کرد و باید کل PCB برنامه از جمله کل منابع تخصیص داده شده به پردازه (مانند لیست فایل‌هایی که باز هستند و I/O device ها)، داده‌های مربوط به مدیریت حافظه و ... نیز ذخیره و بازیابی شوند.

می‌دانیم عمل Context Switch خود برای سیستم سربار خالص است. از طرفی با توجه به توضیحات داده شده می‌توان گفت سربار این عمل زمانی که میان دو ریسمان مربوط به پردازه انجام می‌شود بسیار کمتر از زمانیست که میان دو پردازهٔ مجزا انجام شود. بنابراین استفاده از ریسمان‌ها به جای پردازه‌ها (تا جای ممکن) می‌تواند به بهبود سرعت عملکرد سیستم کمک کند.

(یکی از تفاوت‌های دیگر عمل Context Switch در این دو حالت این است که در برخی از معماری‌ها (که از ASID پشتیبانی نمی‌کنند) زمانی که میان دو پردازه سوئیچ می‌کنیم بافر TLB خالی خواهد شد، اما در زمان سوئیچ میان دو ریسمان خیر. این قضیه باعث می‌شود سرعت دسترسی به حافظه پس از هر Context Switch میان پردازه‌ها به شدت کاهش یابد. در ادامهٔ درس با TLB آشنا خواهید شد اما فعلاً آن را یک حافظهٔ cache فرض کنید که سرعت ترجمهٔ آدرس‌های برنامه به آدرس فیزیکی حافظه را بهبود می‌بخشد.)