HW2, OS

Dr Zali

Aban, 1403

Sepehr Ebadi

9933243

1)

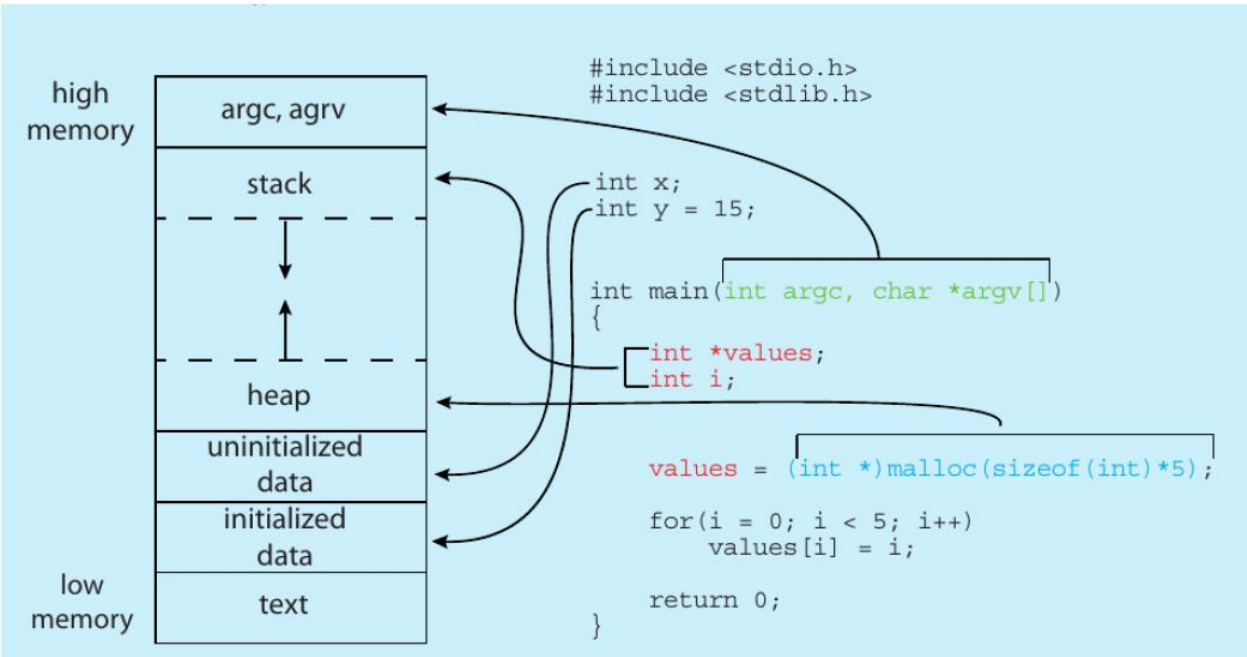
الف)

Text: کد اجرایی برنامه

Data : برای ذخیره‌سازی داده‌های مقداردهی شده‌ای است که در کل زمان اجرای برنامه ثابت می‌مانند global

Heap : برای تخصیص حافظه پویا (malloc)

Stack: برای ذخیره داده‌های مرتبط با فراخوانی توابع مثل متغیرهای محلی و آدرس‌های return



ب)

زمانی که سیستم عامل نیاز داشته باشد از یک پروسس به پروسس دیگر سوئیچ کند نیاز است تا عملیات context switch رخ بدهد که در این عملیات باید ابتدا وضعیت پروسس قبلی و یک سری اطلاعاتش مانند رجیستر هایی که مورد استفاده اش بوده در pcb اش آپدیت شود و سپس اطلاعات پروسس بعدی از pcb ازش لود شود.

پ)

فرآیندی است که در آن سیستم‌عامل زمانی که RAM پر است کل یک پروسس را از RAM به دیسک منتقل می‌کند تا پروسس جدیدی را وارد RAM کند یا برعکس پروسسی را از دیسک به حافظه اصلی برمی‌گرداند و معمولا پروسسی را انتقال میدهد که در حال اجرا نیست یعنی در صف waiting است.

2)

الف)

با استفاده از دستور kill میتواند حتی مشخص کند دقیقا کدام پروسس با کدام pid خاتمه یابد.

ب)

- Child has exceeded allocated resources
- Task assigned to child is no longer required
- The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

۱- اشغال کردن منابع زیادی توسط فرزند

۲- نیازی که داشته برآورده شده پروسس والد

۳- وقتی والد زنده است و سیستم عامل اجازه ندهد فرزند زنده بماند اگر والد تمام شده باشد

ث)

این برنامه باعث ایجاد فرآیند زامبی می‌شود و نه فرآیند یتیم.

پس از خاتمه یافتن فرآیند فرزند، چون فرآیند والد وضعیت خاتمه آن را نمی‌خواند، فرآیند فرزند در حالت زامبی باقی می‌ماند. در این حالت، فرآیند خاتمه‌یافته در جدول فرآیندها باقی می‌ماند تا زمانی که یا فرآیند والد وضعیت آن را بخواند یا خود فرآیند والد خاتمه یابد.

3)

ت)

۱- نیاز به اجرای یک دستور I/O
۲- عامل دیگری که می‌تواند فرآیند را بدون اختیارش به حالت انتظار ببرد، کمبود منابع یا درخواست دیگر فرآیندها برای همان منبع است. به عنوان مثال، وقتی فرآیندی به قفل نیاز دارد و قفل در اختیار فرآیند دیگری است، مجبور است بدون اختیار خود در حالت انتظار بماند تا منبع آزاد شود.

4)

الف)

حافظه مشترک (Shared Memory)

سربار:
حافظه مشترک از سربار کمتری برخوردار است؛ زیرا فرآیندها به طور مستقیم به بخشی از حافظه دسترسی دارند و نیازی به ارسال و دریافت پیام‌ها یا رفت و برگشت داده‌ها از طریق سیستم عامل نیست. این روش برای انتقال حجم زیادی از داده‌ها بسیار کارآمد است.

سهولت برنامه‌نویسی:
برنامه‌نویسی با حافظه مشترک پیچیدگی بیشتری دارد، زیرا برنامه‌نویس باید به صورت دستی مکانیزم‌های هماهنگی و همزمانی (مثل قفل‌ها و سازوکارهای جلوگیری از شرایط رقابت) را پیاده‌سازی کند تا از تداخل و مشکلات رقابتی جلوگیری شود. این باعث می‌شود برنامه‌نویسی با حافظه مشترک مشکل‌تر و پیچیده‌تر باشد.

ارسال پیام (Message Passing)

سربار:
ارسال پیام به دلیل نیاز به ارسال داده‌ها از یک فرآیند به فرآیند دیگر و استفاده از سیستم‌عامل برای مدیریت این پیام‌ها، سربار

بیشتری نسبت به حافظه مشترک دارد. در این روش، سیستم‌عامل باید پیام‌ها را مدیریت و انتقال دهد که زمان و منابع بیشتری را مصرف می‌کند.

سهولت برنامه‌نویسی:
ارسال پیام برنامه‌نویسی آسان‌تری دارد، زیرا خود سیستم‌عامل هماهنگی و مدیریت پیام‌ها را بر عهده دارد و نیاز به همزمانی دستی و پیچیده از سوی برنامه‌نویس نیست. این روش برای سیستم‌های توزیع‌شده که فرآیندها ممکن است در کامپیوترهای مختلف باشند نیز مناسب‌تر است.

ب)

ترمینال (Shell) در لینوکس

(Anonymous Pipe)

در ترمینال لینوکس، از pipeبرای انتقال خروجی یک فرمان به ورودی فرمان دیگر استفاده می‌شود. به عنوان مثال:

```
ls | grep ".txt"
```

در اینجا، خروجی دستور lsبه دستور grepارسال می‌شود تا تنها فایل‌هایی که پسوند txt.دارند، نمایش داده شوند. این نوع pipeناشناس است و به صورت موقت بین دو فرآیند ایجاد می‌شود.

Apache Nginx

Named Pipe (FIFO)

در سرورهای وب مانند Apache وNginx ، از Named Pipesبرای انتقال داده‌ها بین فرآیندهای مختلف استفاده می‌شود. این pipeها نام‌دار (FIFO) هستند و می‌توان از آن‌ها به عنوان یک فایل در سیستم فایل نام برد. این روش به فرآیندهای وب‌سرور اجازه می‌دهد تا با دیگر فرآیندهای سیستم (مثل فرآیندهای لاگ) به سادگی ارتباط برقرار کنند.

Docker

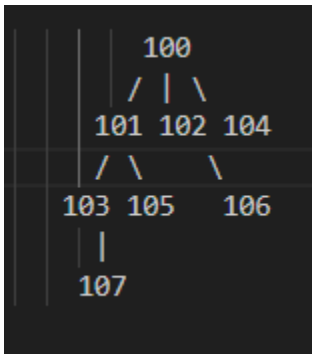
Named Pipe (FIFO)

در سیستم‌عامل‌های ویندوز و لینوکس، Dockerاز Named Pipesبرای ارتباط بین سرویس‌ها و کلاینت‌های Dockerاستفاده می‌کند. در سیستم‌عامل ویندوز، pipeها به‌ویژه برای ارتباط بین کلاینت Docker و دایمون (Docker

(Daemonمورد استفاده قرار می‌گیرند. این ارتباط به صورت پایدار است و کلاینت Docker می‌تواند با استفاده از این pipeدرخواست‌ها و دستورات خود را به دایمون ارسال کند.

5)

الف)



ب)

۱۰۰

۱۰۱

۱۰۲

۱۰۳

۱۰۴

۱۰۵

۱۰۶

۱۰۷

پ)

107, 106, 105, 104, 103, 102, 101, 100

6)

۱. اگر مقدار pidبرابر با ۰ باشد (یعنی کد در فرآیند فرزند اجرا شود)، مقدار متغیرهای global_varو local_varهر کدام به اندازه ۱۰ افزایش می‌یابد. سپس مقدار آن‌ها در خط printfبرای فرآیند فرزند چاپ می‌شود.
۲. اگر مقدار pidبزرگ‌تر از ۰ باشد (یعنی کد در فرآیند والد اجرا شود)، ابتدا فرآیند والد به مدت ۱۰ ثانیه صبر می‌کند (با استفاده از sleep(10)). سپس مقدار متغیرهای global_varو local_varرا در خط printfبرای فرآیند والد چاپ می‌کند.

در این کد، فرآیند فرزند و والد هر کدام نسخه‌ای مستقل از متغیرها را خواهند داشت ه دلیل مکانیزم forkکه یک کپی از فضای حافظه فرآیند اصلی می‌سازد. بنابراین:

فرآیند فرزند مقدار global_varو local_varرا به ترتیب به ۲۰ و ۳۰ تغییر می‌دهد و سپس مقدار این متغیرها را به صورت child -> global_var: 20, local_var: 30چاپ می‌کند.

فرآیند والد (پس از گذشت ۱۰ ثانیه) مقدار اولیه global_varو local_varرا که به ترتیب ۱۰ و ۲۰ است چاپ می‌کند. بنابراین خروجی آن به صورت parent -> global_var: 10, local_var: 20خواهد بود.

خروجی نهایی برنامه به این صورت خواهد بود:

child -> global_var: 20, local_var: 30

parent -> global_var: 10, local_var: 20

7)

الف)

پروسس:
پروسس، یک برنامه در حال اجرا است که شامل کد برنامه، داده‌ها و پشته (Stack) مخصوص به خود می‌باشد. هر پروسس محیط اجرای مستقل خود را دارد و به همین دلیل معمولاً به عنوان یک "واحد مستقل از اجرا" در نظر گرفته می‌شود.

ترد:

ترد، کوچکترین واحد قابل اجرا در یک پروسس است. یک پروسس می تواند شامل یک یا چند ترد باشد و هر ترد به صورت مستقل اجرا می شود. تردها حافظه مشترک دارند و در عین حال از منابع پروسس اصلی استفاده می کنند.

پروسس:

هر پروسس فضای حافظه جداگانه‌ای (مثل حافظه مجازی) دارد که شامل کد، داده و پشته خود است. ایجاد پروسس جدید نیازمند تخصیص منابع جدیدی از جمله حافظه و زمان‌بندی سیستم عامل است. در نتیجه، ایجاد و مدیریت پروسس‌ها هزینه‌برتر از تردها است.

ترد:

تردها از حافظه مشترک پروسس اصلی استفاده می کنند و به همین دلیل ایجاد آن‌ها بسیار کم‌هزینه‌تر از پروسس‌ها است. همه تردهای یک پروسس به کد و داده‌های پروسس اصلی دسترسی دارند و نیازی به تخصیص منابع جدید و مستقل ندارند.

پروسس:

هر پروسس از سایر پروسس‌ها جداست، به این معنا که یک پروسس نمی تواند مستقیماً به حافظه یا منابع دیگر پروسس‌ها دسترسی پیدا کند. این ویژگی امنیت بیشتری ایجاد می کند، زیرا در صورت وقوع خطا در یک پروسس، تاثیری روی پروسس‌های دیگر نخواهد داشت.

ترد:

تردها به دلیل اشتراک حافظه و منابع پروسس اصلی، از نظر امنیت و جداسازی ضعیف‌تر هستند. اگر یکی از تردها خطایی داشته باشد یا به درستی عمل نکند، می تواند کل پروسس را مختل کند، زیرا به داده‌ها و منابع مشترک دسترسی دارد.

پروسس:

پروسس‌ها به دلیل تخصیص منابع جداگانه، کندتر از تردها عمل می کنند. همچنین، ارتباط بین پروسس‌ها (Interprocess Communication) نیازمند سیستم‌های خاصی مانند Pipe ، Shared Memory و Message Passing است که باعث افزایش سربار و کاهش کارایی می‌شود.

ترد:

تردها به دلیل اشتراک منابع پروسس اصلی، سریع‌تر اجرا می‌شوند و ارتباط بین تردها ساده‌تر و کم‌هزینه‌تر است. به همین دلیل، تردها برای برنامه‌های با کارایی بالا و وظایف موازی که نیاز به دسترسی مشترک به داده‌ها دارند، مناسب‌تر هستند.

پروسس:

پروسس‌ها برای برنامه‌های جدا از هم استفاده می‌شوند که نیاز به جداسازی کامل دارند، مانند اجرای برنامه‌های مختلف روی سیستم عامل یا اجرای سرویس‌های مستقل (مانند سرویس‌های سیستم).

ترد:

تردها برای بهبود عملکرد برنامه‌هایی که نیاز به پردازش موازی دارند، استفاده می‌شوند، مانند پردازش تصویر، بازی‌های کامپیوتری، وب سرورها و برنامه‌هایی که نیازمند اجرای همزمان وظایف مختلف هستند.

پروسس:

اگر یک پروسس دچار مشکل یا خطا شود (Crash) ، معمولاً فقط همان پروسس از بین می‌رود و بر سایر پروسس‌ها تاثیری نمی‌گذارد. این ویژگی باعث می‌شود که پروسس‌ها در مقایسه با تردها انعطاف‌پذیرتر باشند.

ترد:

تردها به دلیل اشتراک منابع و حافظه، به شدت به یکدیگر وابسته‌اند. اگر یکی از تردها دچار مشکل شود، کل پروسس تحت تاثیر قرار می‌گیرد. این می‌تواند باعث ناپایداری بیشتر تردها نسبت به پروسس‌ها شود.

پروسس:

ایجاد و مدیریت پروسس‌ها هزینه زمانی بیشتری نسبت به تردها دارد، زیرا نیاز به تخصیص منابع مستقل و حافظه دارد. تغییر وضعیت بین پروسس‌ها (Context Switching) نیز به دلیل تغییر فضای آدرس مجازی، زمان بیشتری می‌برد.

ترد:

تردها از حافظه و منابع مشترک استفاده می‌کنند، بنابراین تغییر وضعیت بین تردها هزینه زمانی کمتری دارد. این ویژگی باعث می‌شود که تردها برای برنامه‌هایی که نیاز به پردازش موازی و کارایی بالا دارند، مناسب‌تر باشند.

پروسس:

پروسس‌ها واحدهای اصلی زمان‌بندی و مدیریت در سیستم‌عامل هستند. سیستم‌عامل به هر پروسس یک یا چند ترد اختصاص می‌دهد و پروسس‌ها را بر اساس اولویت و نیازهای منابع زمان‌بندی می‌کند.

ترد:

تردها معمولاً در داخل پروسس‌های موجود ایجاد می‌شوند و از آن‌ها برای پردازش موازی وظایف استفاده می‌شود. سیستم‌عامل می‌تواند تردها را به‌طور مستقل زمان‌بندی کند، اما آن‌ها تحت فضای آدرس پروسس اصلی باقی می‌مانند.

ب)

مزایای ایجاد ترد نسبت به پروسس:

سربار کمتر :تردها از منابع و فضای آدرس مشترک پروسس اصلی استفاده می‌کنند، بنابراین ایجاد آن‌ها نسبت به پروسس‌ها سربار کمتری دارد. تخصیص حافظه جدید و منابع مستقل نیاز نیست و به همین دلیل، سرعت ایجاد و اجرای تردها بیشتر است.

تغییر وضعیت سریع‌تر :تغییر وضعیت (Context Switching) بین تردها بسیار سریع‌تر از پروسس‌ها انجام می‌شود. به دلیل استفاده از فضای آدرس مشترک، نیازی به تغییر فضای آدرس مجازی نیست، که منجر به کاهش سربار زمانی تغییر وضعیت می‌شود.

اشتراک‌گذاری داده‌ها :تردها به طور مستقیم به داده‌های مشترک پروسس اصلی دسترسی دارند. این ویژگی باعث می‌شود که تردها به راحتی داده‌ها را بین خود به اشتراک بگذارند و نیازی به ارتباط بین‌پروسسی پیچیده مثل Pipe یا Shared Memoryنباشد. این ویژگی در برنامه‌هایی که نیاز به پردازش موازی داده‌های مشترک دارند، بسیار مفید است.

کارایی بالا در پردازش‌های موازی :به دلیل سربار کمتر و دسترسی سریع‌تر به داده‌ها، تردها برای پردازش‌های موازی و تسک‌های همزمان بسیار کارآمد هستند. در برنامه‌هایی مثل پردازش تصویر، بازی‌های کامپیوتری، و سرورهای وب که نیاز به پاسخ‌دهی سریع دارند، استفاده از تردها کارایی بالاتری دارد.

با وجود مزایای ذکر شده برای تردها، در برخی موارد استفاده از برنامه‌های چندپروسسی مزایای قابل توجهی دارد:

۱. **ایمنی و جداسازی** :پروسس‌ها فضای حافظه و منابع جداگانه‌ای دارند، بنابراین اگر یکی از پروسس‌ها با خطا مواجه شود یا دچار Crash شود، این خطا بر سایر پروسس‌ها تاثیری نمی‌گذارد. در برنامه‌های چندپروسسی، هر پروسس به صورت ایمن از پروسس‌های دیگر جدا است، که امنیت بیشتری را فراهم می‌کند. این ویژگی در سیستم‌های حساس که امنیت و پایداری اهمیت دارد (مثل سیستم‌های بانکی یا پردازش‌های سرور) بسیار مهم است.
۲. **پایداری بیشتر** :در برنامه‌های چندپروسسی، هر پروسس به طور مستقل اجرا می‌شود. اگر یک پروسس با مشکل مواجه شود، سایر پروسس‌ها همچنان می‌توانند به کار خود ادامه دهند. این ویژگی در مقایسه با تردها، که به دلیل اشتراک منابع به شدت به یکدیگر وابسته‌اند، پایداری بیشتری دارد.
۳. **استفاده از چند هسته پردازنده** :در برخی سیستم‌عامل‌ها، ممکن است اجرای چندین پروسس به صورت همزمان بر روی هسته‌های مختلف پردازنده انجام شود. این ویژگی باعث افزایش کارایی و بهره‌وری در سیستم‌های چند هسته‌ای می‌شود، در حالی که اجرای تردهای یک پروسس ممکن است به صورت محدودتری روی هسته‌های مختلف توزیع شود.
٤. **اجتناب از مشکلات همزمانی** :در برنامه‌های چندپروسسی، هر پروسس حافظه و فضای داده‌های جداگانه‌ای دارد و مشکلات همزمانی) مانند (Race Condition کمتر رخ می‌دهد. مدیریت همزمانی در برنامه‌های چندتردی پیچیده است و نیاز به ابزارهایی مثل Mutex و Semaphore دارد، اما در برنامه‌های چندپروسسی، این مشکلات به طور طبیعی کمتر هستند.

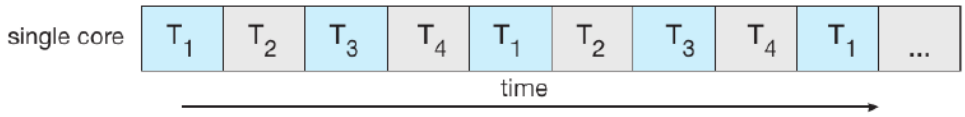
۵. **مناسب برای وظایف مستقل** :در مواردی که وظایف به طور کامل مستقل از یکدیگر هستند و نیازی به اشتراک داده‌ها ندارند، برنامه‌های چندپروسسی می‌توانند مناسب‌تر باشند. برای مثال، در سرورهایی که هر درخواست یک تسک مستقل دارد، استفاده از پروسس‌های جداگانه برای هر درخواست می‌تواند مدیریت و نظارت بهتری ایجاد کند.

8)

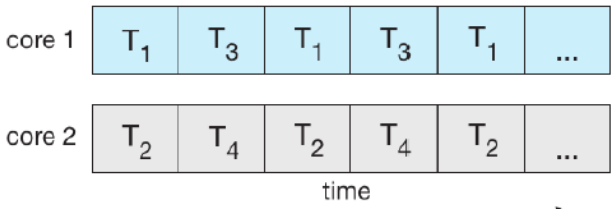
الف)

در حالت concurrency به صورت time sharing پروسس ها به صورت همزمان اجرا میشوند و در واقع به صورت مجازی و vitual همزمانی رخ میدهد ولی در حالت parallelism در صورتی که بیش از یک cpu core داشته باشیم دو تسک مختلف میتواند به صورت واقعی به صورت همزمان اجرا شود.

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



ب)

ابتدا، فرآیند والد و فرزند اول داریم.

در فرآیند فرزند اول، یک فرآیند دیگر ایجاد می‌شود که فرزندِ فرزند است.

در نهایت، هرکدام از این سه فرآیند (والد، فرزند اول، و فرزندِ فرزند) یک بار دیگر fork()را فراخوانی می‌کنند و هر کدام یک فرآیند جدید ایجاد می‌کنند.

در مجموع، **۸ فرآیند** خواهیم داشت.

نخ‌ها:

تنها در فرآیند فرزند اول، یک نخ ایجاد می‌شود. بنابراین، **۱ نخ** خواهیم داشت.

تعداد فرآیندهای منحصر به فرد ایجاد شده :۸

تعداد نخ‌های ایجاد شده :۱

9)

قسمت الف: اجرای دستورات پایپ(Pipe)

ابتدا یک پایپ با استفاده از تابع pipeایجاد می‌کنیم که دو طرف دارد: یک طرف برای نوشتن و طرف دیگر برای خواندن.

سپس دو فرآیند فرزند ایجاد می‌کنیم:

فرزند اول خروجی خود را به قسمت نوشتن پایپ (pipefd[1]) می‌فرستد. برای این کار، از dup2استفاده می‌کنیم که خروجی استاندارد (stdout) را به پایپ تغییر می‌دهد.

فرزند دوم ورودی خود را از قسمت خواندن پایپ (pipefd[0]) می‌گیرد و از dup2برای تغییر ورودی استاندارد (stdin)استفاده می‌کند.

هر دو طرف پایپ را در فرآیند والد می‌بندیم و صبر می‌کنیم تا هر دو فرآیند فرزند به پایان برسند.

قسمت ب: ریدایرکت خروجی به فایل

۱. اگر دستور شامل >باشد، آن را به دو بخش دستور و نام فایل تقسیم می‌کنیم.
۲. یک فرآیند فرزند ایجاد می‌کنیم و خروجی استاندارد را با استفاده از dup2به فایل تغییر می‌دهیم.
۳. سپس دستور را در این فرآیند فرزند اجرا می‌کنیم. اگر فایل وجود نداشته باشد، آن را ایجاد و در صورت نیاز، مجدداً بازنویسی می‌کنیم.
٤. فرآیند والد منتظر می‌ماند تا فرآیند فرزند اجرا را کامل کند و سپس به کار خود ادامه می‌دهد.

tt.c++ 3

tt.c++ > ...

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>

#define MAX_INPUT 1024

void execute_command(char *command) {
    char *args[10];
    int i = 0;
    args[i] = strtok(command, " ");
    while (args[i] != NULL) {
        args[++i] = strtok(NULL, " ");
    }
    execvp(args[0], args);
    perror("execvp failed");
    exit(1);
}

void handle_pipe(char *command1, char *command2) {
    int pipefd[2];
    pipe(pipefd);

    pid_t pid1 = fork();
    if (pid1 == 0) {
        close(pipefd[0]);
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[1]);
        execute_command(command1);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        close(pipefd[1]);
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[0]);
        execute_command(command2);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid1, NULL, 0);
    waitpid(pid2, NULL, 0);
```

tt.c++ > ...

```c++
void handle_pipe(char *command1, char *command2) {
}

void handle_redirect(char *command, char *filename) {
    pid_t pid = fork();
    if (pid == 0) {
        int file_fd = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
        if (file_fd < 0) {
            perror("Failed to open file");
            exit(1);
        }
        dup2(file_fd, STDOUT_FILENO);
        close(file_fd);
        execute_command(command);
    }
    waitpid(pid, NULL, 0);
}

int main() {
    char input[MAX_INPUT];

    while (1) {
        printf("shell> ");
        fgets(input, MAX_INPUT, stdin);
        input[strcspn(input, "\n")] = 0;

        if (strstr(input, "|")) {
            char *command1 = strtok(input, "|");
            char *command2 = strtok(NULL, "|");
            handle_pipe(command1, command2);
        } else if (strstr(input, ">")) {
            char *command = strtok(input, ">");
            char *filename = strtok(NULL, " ");
            handle_redirect(command, filename);
        } else {
            pid_t pid = fork();
            if (pid == 0) {
                execute_command(input);
            }
            waitpid(pid, NULL, 0);
        }
    }

    return 0;
}
```