Licenciatura em Engenharia Informática, Faculdade de Ciências e Tecnologias

Universidade do Algarve

Ano letivo 2018/2019

Sistemas Operativos

Relatório Prático
Guião 2: "C: Processos, Fork, ..."

Docente:

Amine Berquia

Aluno:
Guilherme *"Matrix"* Dias, nº 61237

Índice

Introdução

Neste relatório intenciono apresentar soluções para os problemas colocados no guião PL2, e as tecnicas usadas para os resolver. Entre estas estão Processos, *Pipes*, *Threads*, Semáforos.. Será utilizada a linguagem C, visto que os APIs oferecidos pelo sistema operativo a utilizar – Ubunto – foram escritos também em C, e têm maior compatibilidade e facilidade de *linkage* com este.

Por falar em C e Ubuntu, vamos escavar um pouco mais fundo. Ubuntu é uma distribuiçãode linux, e o objetivo geral do linux é ser o mais compativél possível entre diferentes tipos de hardware. É por isto programado com a linguagem C. O que dita que desde que hexista um compilador para a arquitetura em questão, irá também haver (após se ser ter lidado com específicos da arquitetura como paging ou segmentation, i/o reads ou writes, etc..) uma destribuição de linux.

O linux e as suas distribuições seguem porém um conjunto de regras no que toca aos APIs que disponibilizam aos programadores da sua plataforma. Essas regras ditam que funções devem hexistir e o que devem fazer, para que código em C de uma plataforma funcione em qualquer outra que siga este conjunto de regras. Este conjunto de regras chama-se POSIX (Portable Operating System Interface). Por norma todos os sistemas operativos derivados de Unix usam este “standard”.

Hexistem porém outros sistemas operativos que não o fazem. Temos o exemplo do Windows, que, apesar de uma quantidade razoavel da sua funcionalidade ter sido derivada e/ou retirada do Unix, não respeita estes standards. O windows tem standards completamente diferentes no que toca aos conceitos a ser abordados, que, apesar de fazerem a mesma coisa, têm nomes e linkage diferente. É por então esta razão que muitas pessoas vindas de plataformas como o linux para o windows, designam a programação neste como “um cancro completo”.

Vamos abordar Forks, Pipes, Threads e muitos outros conceitos, que, de uma maneira ou de outra estão implementados em todos os sistemas operativos. Há uma razão para isto. As arquiteturas e os instruction sets que estão por baixo todos funcionam (se não da mesma maneira,) de uma maneira parecida. Hexiste virtualização de adereços, feita com paging. Paging é o mapeamento de adereços físicos (que representam localizações na RAM) a adereços virtuais que podem ser utilizados por processos.

Mas estamos a ir um pouco rápido de mais. Processos.

Processos são, de uma certa maneira, carrinhas. Sei que provavelmente não é a melhor analogia, mas carrinhas (e partindo do principio que são modelos idênticos acabados de sair da fábrica), todas têm as mesmas peças, nos mesmos sítios, e todas funcionam da mesma maneira. Estas têm porém espaço disponível dentro delas. De uma forma abstrata, podemos pensar assim. Mas estas carrinhas têm uma particularidade. Lá dentro é posto um macaco. Imaginando que esta carrinha é uma carrinha de encomendas, vamos assumir que este macaco pode estar a organizar as encomendas, ou a desorganiza-las, ou a fazer castelos, dependendo do macaco.

Na realidade há muita coisa má com esta analogia, visto que por exemplo originalmente muito pouco espaço pertence efetivamente ao processo, e esta carrinha está completamente sobre a influência deste ser omnisciente e todo poderoso chamado kernel, que com ela pode fazer virtualmente qualquer coisa sem qualquer dificuldade.

Mas vamos continuar. Cada vez que uma carrinha é posta ao serviço, é posto um macaco lá dentro. Este nosso macaco é na realidade um programa. Programas estão guardados por norma em armazenamento (storage em inglês, discos, SSDs, pendrives, etc..). Quando o processo é começado, este não executa código por magia. O sistema operativo vai ao disco buscar o programa, que está no formato de ELF (Executable and Linkable Format) no linux, ou PE (Portable Executable), no Windows, etc.. Este programa é então mapeado dentro deste processo. A este ponto a analogia quebra completamente, porque este macaco tem o poder de rezar ao referido ser todo poderoso, e fazer aumentar o espaço da carrinha, colonar a carrinha, fazer aparecer outros macacos dentro da carrinha, etc..

É então para isto que serve o standard do POSIX, para que o nosso macaquinho, mesmo estando numa carrinha de um modelo diferente, continue a saber pegar numa ak-47 e disparar pela janela.

Já estou a exagerar.

A razão pela qual surgiram processos foi uma de compatibilidade e segurança. Se todos os processos tivessem o mesmo address space (o físico), eventualmente, alguns desses processos iriam começar a rescrever memória dos outros, e ao fim de algum tempo inevitavelmente uma exceção iria acontecer que iria “matar” todo o sistema operativo. Para além disto, todos os processos teriam fácil acesso ao address space de outros (visto que seria o mesmo), o que faria com que alguém com más intenções podesse aceder a dados sensíveis. Por exemplo, alguém poderia facilmente fazer uma calculadora que roubava passwords de facebook. Não quer dizer que não seja possível com a implementação de processos, mas é mais difícil.

Address space, para o caso de ainda haver dúvidas é o espaço ao qual um determinado programa pode aceder. Num address space físico (onde o kernel “reside”), cada adereço corresponde a uma série de 8 transístores. Um byte, 8 bits, oito vezes 0 ou 1. Um address space virtual é o nível de abstração acima. Em que um adereço virtual corresponde a um adereço físico. Um adereço virtual pode corresponder ao mesmo número que um adereço físico, mas não é comum. Estes address spaces fazem parte e são implementados diretamente no processador na forma do (referido acima) paging. Em que o sistema operativo comunica ao processador que quer que por exemplo, cada vez que uma instrução no contexto x use um adereço de 0 – 100, devolva a memória física 420 – 430.

Agora que já sabemos isto, e já temos uma boa mão no que são processos e address spaces, podemos definir e/ou explicar o que são alguns dos conceitos que vamos utilizar na resolução dos exercícios.

Já alguma vez pensou(aram) que pode ser útil pegar num processo, a meio de ele estar a correr, e colona-lo? Com colonar eu quero dizer fazer uma cópia exata para outro address space. Não? Compreensivel, eu também não. Mas alguém pensou. E fizeram isso.

Mas isto que eu estou a dizer não vem do nada, quando eu mais atrás falei de o macaco poder colonar a carrinha, não era só uma piada. Nada daquilo foi **só** uma piada. Hexiste uma função Fork.

Fork. Garfo? Não. Se formos ao google e procurarmos por fork verb definition, vai-nos aparecer duas coisas: “Spooning leads to Forking”, e “(especially of a route) divide into two parts”, a primeira tem piada mas nós só precisamos da segunda neste momento. A função fork divide o processo em dois. Faz o que eu descrevi no parágrafo acima. Isto é util, porque podemos ter múltiplos processos, a realizar várias tarefas, mas apenas 1 programa a correr.

Ou seja teriamos assim um programa que dividiria o seu processo em dois ou mais.

Porque é que isto pode ser util? O google chrome fa-lo. Nessa situação concreta é por razões de segurança. Devido ao que eu disse anteriormente sobre a falha de segurança que ter vários processos a correr no mesmo address space pode apresentar. Por exemplo, se um hacker “hackar” uma tab do chrome através de um website, não vai ter qualquer tipo de acesso às outras porque estão num address space completamente diferente.

Antes de darmos a explicação final, vamos falar sobre o PID e o PPID. Já referi que um sistema operativo opera com processos, e tem por norma múltiplos a correr de cada vez. Estes processos precisam de identificação, porque senão ninguém sabe o que é o quê. Se tentarmos imaginar um mundo em que os processos não têm identificaçao, estariamos também a tentar imaginar um mundo em que ninguém saberia quais dos processos estariam a correr como administrador. Se não se sabe quem uma pessoa é, não se sabe a idade ou trabalho dela por exemplo.

Agora que já sabemos que um processo pode gerar outros processos, faz algum sentido que os deixêmos de certa maneira associados. Para isto foi inventado o esquema de parent-child. Em que, se um processo fizer fork, o processo criado vai ter como processo “parent” (pai) o que o criou.

PPID é assim o identificador do processo pai. Todos os processos têm um PPID, até os que aparentam ser um pai. Por exemplo quando executamos um processo na consola, o PPID deste vai ser o PID do processo da consola. O processo da consola vai ter PPID do PID do processo do UI que abriu a consola, e por aí em diante. Mas sim, obviamente tem um fim. Os primeiros processos a serem criados (pelo menos no linux) têm um PPID de 1. Este 1 representa o “processo” init. O processo init não é bem bem um processo como é mais uma parte do kernel em si que é responsável por criar processos.

Por isso podemos agora definir o FORK. Fork é chamado apenas uma vez. O Fork comunica com o kernel por via de um interrupt(que não vou explicar o que é porque acredito que o fiz no último relatório em detalhe), e o kernel vai colonar, aka, fazer uma cópia binária idêntica do processo que chamou o fork, e começar a execução desse a partir daí. Temos assim uma função que tecnicamente, e por definição, dá return duas vezes. (em processos diferentes, sim, mas duas vezes *nonetheless*).

THREADS.

Threads, ok.

Threads. Lembra(m)-se quando eu falei sobre o macaco poder fazer aparecer outros macacos dentro da carrinha? É isso. Bem fixe não? Quem me dera a mim poder spawnar mais 3 versões de mim para lavar a loiça. (se bem que se fossem verdadeiramente eu continuavam todas a não lavar a loiça).

A vantagem das threads, é que ao contrário de criar outros processos que podem correr em simultâneo, estas podem criar "sub-processos" dentro do mesmo address space. Isto dá-nos a possibilidade de processar vários dados ou eventos ao mesmo tempo.

Sinto que num leitor com menos conhecimento isto poderia suscitar a pergunta de "mas não tinhas dito que isso era menos seguro que usar outros processos?". Sim claro. Mas isso também tira muitas possibilidades. Se quiseres aceder à mesma variável (não importante como uma password) dos dois processos, não podes. (poder podes, mas isso tem que ver com pipes, depois já falamos sobre o assunto).

Num programa tem-se múltiplas instruções e o normal é o nosso processador ir percorrê-las uma a uma, executando assim o programa. Mas e se nós tivessemos vários processadores? Podiamos pô-los a executar outro sítio qualquer independente, e assim podiamos ter o dobro da velocidade certo? É mais ou menos este o conceito. Threads, na sua abstração podem ser intrepertadas desta maneira. Mas eu vou dar um pouco de background. O sistema operativo tem este elemento chamado schedueler, em que vai olhar para as instruções de todos os processos que estão a ser executados, e vai executar um deles em específico com o processador chama-se transferir controlo ao processo. Mas, eventualmente, esse processo já vai estar a receber demasiada atenção e os outros vão começar a chorar, por isso o sistema operativo vai reavaliar, e transferir controlo a outro que já não o tenha à algum tempo. Um pouco como mamar. E é assim que, por exemplo num CPU só com uma core, se gera a ilusão de multi-tasking. Mas isto não acontece ao nível dos processos como tenho estado a fazer acreditar, acontece com as threads. As threads recebem mama, não os processos. Como é obvio, o schedueler tem alguma atenção aos processos também, caso contrário, um processo com 9001 threads teria uma quantidade absurda de tempo de execução comparado aos outros, o que não é desejável.

Já vimos então que threads são como que "Linhas de execução" dentro dos processos, em que de um ponto de vista abstrato, executam todas ao mesmo tempo, fazendo o processo muito mais rápido.

Agora que já sabemos o que são threads, podemos analisar algumas funções.

pthread_create. Thread e create são consideravelmente self-explanatory, mas aquele p? Faz lembrar alguma coisa? Provavelmente não. Mas e se o metermos maiúsculo? Hmm.. Nada ainda? E um "o" maiúsculo à frente? Isso mesmo, POSIX. Posix, thread e create. Como já vimos posix é só um

standard, para que possamos criar threads em qualquer sistema operativo que lhe obedeça com apenas pthread_create.

Específicas. Isto é só mais C na realidade, há só um parâmetro que nos interessa e que pode não ser demasiado evidente à primeira vista. start_routine. Start routine é a função na qual a thread vai começar. Simples quanto isso. Não se pode começar uma dita cuja "Linha de execução" sem nada para executar certo? Por isso sim, é a função na qual a thread vai começar.

Temos ainda outra função que não é demasiado óbvia à primeira vista: pthread_join. Mas é simples também. Como a thread principal do programa pode terminar (e não só frequentemente como tamb+em idealmente acontece) a sua execução antes das outras, dá-nos jeito que esta não pare e as outras fiquem penduradas a fazer sabe-se lá o quê. Se pensarmos mais uma vez nos macacos, um pthread_create é um "cria um macaco dentro da carrinha que faz uma tarefa x". Se imaginarmos que um desses macacos tem a função de conduzir, temos que admitir que não podemos simplesmente guardar a carrinha e deixar o maluco do macaco lá dentro. Ou seja pthread_join simplesmente espera que a thread argumento pare. Por outras palavras, que o macaco decida sair de lá de dentro. Como é obvio isto não é a solução perfeita, porque ele pode resolver que lhe apetece ficar a dar cabeçadas no volante mais 15 minutos, ou que quer passar o resto da vida lá dentro.

Uma boa solução para parar a thread é à força. Podíamos tira o macaco da carrinha à força, e para isso existem funções como pthread_cancel, mas são má prática, porque imaginando que a função do do macaco era arrumar as caixas, podiamos ficar com aquilo tudo desarrumado, ou ainda pior, ele deixava as caixas numa posição em que caíam em cima da próxima pessoa que abrisse a porta. O que pode parecer engraçado, mas se as caixas tivessem blocos de cimento lá dentro perdia a piada rapidamente.

Por isso temos semáforos.

Eu sei, eu sei. Macacos, carrinhas, e agora semáforos!? Infelismente (ou não, preferência pessoal) foi o nome que os sujeitos do POSIX resolveram dar a este elemento. Semáforos são até bastante mais simples (eu acho) do que semáforos na vida real. Podem ser mais complexos, mas o contrário de simples é difícil, e difícil isto não é.

Semáforos são utilizados para esperar por eventos entre threads. Vamos imaginar que a thread principal (a primeira que é executada quando se corre um programa), quer fazer a tabuada do 1, mas até 4294967295. E vamos imaginar que queremos também fazer a tabuada do 2 até 1000. É portanto lógico utilizar threads. Uma para a tabuada do 1, outra para a do 2. Usamos então a thread principal para criar uma thread segundária e fazer a tabuada do 1. Utilizaremos a thread secundária para fazer a tabuada do 2. MAS Nós só queremos fazer a tabuada do 1 até termos acabado a do 2. Isto em macacos é: macaco 1 pede ajuda ao macaco 2 para fazer tabuadas. Depois ambos fazem as tabuadas do 1 e do 2. Obviamente o primeiro macaco vai demorar mais tempo a fazer a tabuada do 1 do que o outro. Eles combinaram que quando ambos acabassem tabuada, iam beber umas ricas jolas. Para isto ser possivél o primeiro teria que avisar o segundo quando acabasse, caso contrário, ficavam os dois sentados sem fazer nada. (vamos assumir que o segundo não estava a olhar).

Semáforos são portanto isto. O avisar de outra thread que um trabalho foi acabado. Os semáforos são “objetos” que ambas as threads podem aceder com segurança.

sem_wait espera que um semáforo fique “verde” - 1, e sem_post faz com que o semáforo fique verde. Em macacos, isto é: sem_wait é o macaco que fica a olhar para a parede à espera que o outro acabe a tabuada do 1, e sem_post, é o macaco que acaba a tabuada do 1 dar um calduço ao segundo (e irem beber as ditas cujas jolas).

Não esquecer que antes de um semáforo poder ser utilizado, tem que se inicializar com sem_init e terminar com sem_close.

DEADLOCKS. Deadlocks são a coisa mais obvia do mundo assim que se começa a trabalhar com semáforos ou multi-processos, e também que eu já referi. Noutras palavras ainda, deadlock é quando por exemplo o macaco não avisa o outro macaco porque assume que acabou primeiro, e ficam infinitamente um à espera um do outro.

Deadlocks são um problema sério quanto mais baixo o nível é. Um deadlock no kernel ou drivers pode, e muitas vezes efetivamente causa o famoso “freeze”. Que é quando o sistema operativo pura e simplesmente deixa de responder.

Agora vamos não falar de coisas tristes, e passar ao último tópico. Pipes.

Pipes. Tubos? Hm?

Ok, vou ser rápido. Pipes servem para comunicação entre processos. Quando eu disse previamente que uma das vantagens de threads sobre processos era a partilha do mesmo address space, e que os processos não conseguiam comunicar entre si? Meti lá uns parêntices a dizer que não era bem esse o caso.

Pipes podem ser interpertadas como ficheiros. As funções usadas para se interagir com estas são as mesmas que são utilizadas para ficheiros. read e write. De uma certa forma, pipes podem ser consideradas ficheiros. No linux existe este conceito de ficheiros de memória, em que são ficheiros representados exatamente como os de disco, mas cuja localização é em memória, e se não forem guardados e o sistema operativo desligar, pura e simplesmente deixam de existir. Um bom exemplo é tudo em /dev/shm. Por isso as pipes acabam por ser como um espécie de ficheiros, que residem em memória, e são por isso rápidos (e não de todo um dano a performance).

Pipes são criadas com a função pipe, em que tem como argumento um array de dois ints. Estes vão ser utilizados como file descriptors (como por exemplo quando se abre um ficheiro ou o stdin/out da consola). As pipes devem ser criadas antes de um fork ser feito, pois assim ambos os address spaces terão uma copia dos file descriptors. Depois, do ponto de vista de cada processo, desses dois file descriptors, o primeiro vai ser o de leitura, e o segundo o de escrita. Ou seja, para ler de uma pipe, deve-se fazer “read(fds[0], …”, e para escrever “write(fds[1], …”. Se tiver(em) um olho atento, vão reparar que processo A vai estar a ler do file descriptor 0 apesar de o processo B estar a escrever no file descriptor 1. Isto meus amigos é magia do kernel.

Pipes são porem unidirecionais. Ou seja, para que comunicação possa ser feita, do processo A para o processo B, o processo A tem que chamar a função close no seu file descriptor de leitura, e o processo B no seu file descriptor de escrita.

Para comunicação bi-direcional devem ser criadas duas pipes.

Chegamos assim ao fim desta aventura, espero ter sido útil. Agora a resolução dos exercícios deve ser bastante fácil de compreender.

Exercícios

Resolverei agora os exercícios do guião, tendo como objetivo **demonstrar as minhas capacidades** e o meu código.

Dado que já expliquei não só as funções que vou utilizar mas também o conceito por detrás das mesmas na minha introdução teórica, a resolução dos exercícios será dedicada ao código, output, e notas sobre a resolução bem como um resumo geral de cada exercício.

P1

A

A quantidade de processos **iniciados** pelo programa é **3**.

Se quisermos contar com o processo pai, teremos **4 processos no total**.

Como falei previamente na minha introdução teórica, os processo, no meu caso da consola, **iniciou** o processo deste programa. O programa em si, após estar a correr num/em múltiplos processos **inicia** os seus forks.

Para este programa completar, **foram iniciados 4 processos** no total.

Visto que eu já expliquei o fork, faz sentido pois:

-no primeiro fork, o processo pai, cria um processo filho.

-ambos resumem execução no return desse fork

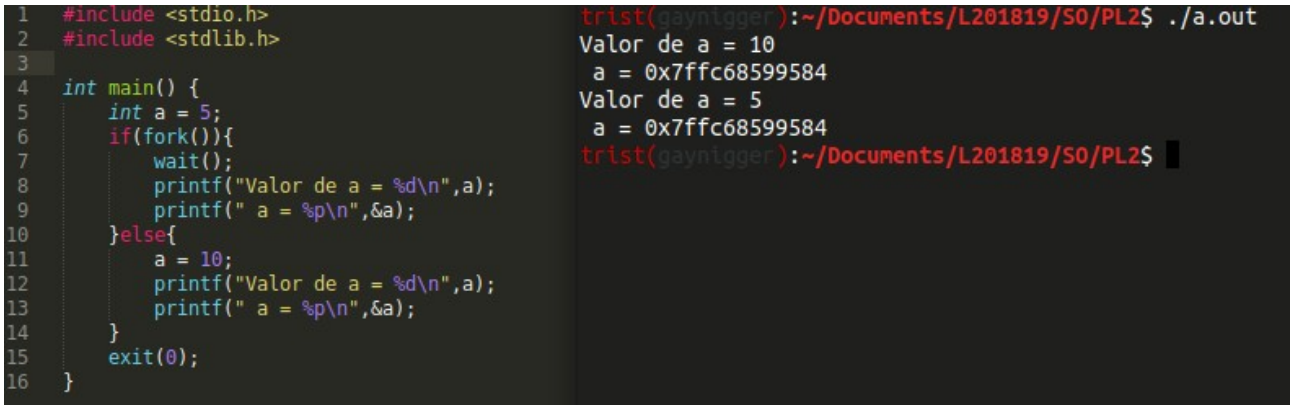
-temos agora 2 processos que vão executar o 2º fork

-criando assim 4 processos

Por isso, e mais uma vez, podemos concluir que o processo gerado por este programa, cria 3 sub-processos.

Dois filhos e um neto.

B



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a = 5;
    if(fork()){
        wait();
        printf("Valor de a = %d\n",a);
        printf(" a = %p\n",&a);
    }else{
        a = 10;
        printf("Valor de a = %d\n",a);
        printf(" a = %p\n",&a);
    }
    exit(0);
}
```

```
trist(gaynigger):~/Documents/L201819/SO/PL2$ ./a.out
Valor de a = 10
 a = 0x7ffc68599584
Valor de a = 5
 a = 0x7ffc68599584
trist(gaynigger):~/Documents/L201819/SO/PL2$
```

Os resultados são simples, o pai vai fazer um fork, o que quer dizer que vai também passar a condição do if, visto que nele, o fork dá return de um valor não zero.

Temos depois um wait, que vai esperar que o correspetivo filho morra (acabe execução). Por isso agora vamos ao filho.

O filho atribui à variável A o valor 10. Imprime depois o seu valor e adereço e acaba execução.

O pai pode então resumir execução e fazer estes últimos dois passos.

Como podemos ver pelo output (e como esperado) no filho, a tem valor 10, e no pai, a tem valor 5.

Porém ambos têm o mesmo adereço. Isto deve-se ao facto de que o fork clona o processo pai quando é chamado.

Como já vimos, o fork vai ser chamado uma vez, e vai dar return 2 vezes, em processos binariamente idênticos, com a unica diferença sendo a register na qual tem o return (eax para x86, rax para x86_64 por exemplo).

Como o processo é idêntico, o seu address space está mapeado também de maneira igual, logo, as variáveis vão ter adereços virtuais iguais, apesar de terem localizações físicas diferentes. Isto é possível com paging e segmentation (estando paging presente em todas as arquiteturas de propósito geral).

P2

A

Este exercício sofre de uma ambiguidade lexical, em "Depois de ver cada mensagem, para o processo ativo com a instrução Sleep.". É difícil identificar se para é a preposição para, ou para do verbo parar.

É também confuso visto que a esse ponto do enunciado, supostamente ambos os processos estão ativos.

Vou porém imaginar que a última frase ("O pai espera a morte do processo filho.") está antes da anteriormente referida, em cujo caso já faz sentido, visto que o pai esperando a morte do filho poderia ser considerado "inativo".

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <wait.h>

int branchout(void(*child)()){
    int c = fork();
    if(!c)child();
    return c;
}

void child_proc(){
    int ppid = getppid();
    for (int i = 0; i < 3; ++i)printf("Eu sou o filho, meu pai é %i\n", ppid);
    sleep(2);
}


int main(int argc, char const *argv[]){
    if(!branchout(child_proc))return 0;

    int p = getpid();
    for (int i = 0; i < 5; ++i)printf("Eu sou o pai, minha identificação é %i\n", p);
    wait(0);

    return 0;
}
```

```
trist(gaynigger):~/Documents/L201819/SO/PL2$ ./a.out
Eu sou o pai, minha identificação é 12676
Eu sou o pai, minha identificação é 12676
Eu sou o pai, minha identificação é 12676
Eu sou o pai, minha identificação é 12676
Eu sou o pai, minha identificação é 12676
Eu sou o filho, meu pai é 12676
Eu sou o filho, meu pai é 12676
Eu sou o filho, meu pai é 12676
trist(gaynigger):~/Documents/L201819/SO/PL2$
```

Tenho então uma função branchout, que dado um ponteiro para uma função child, controla a execução conforme o resultado do fork, de maneira em que acabe sempre sem nunca ser preciso um “exit”, que eu pessoalmente acho mau design.

A criança imprime o PID do pai 3 vezes, espera dois segundos, e retorna.

O pai faz uma criança, imprimee o seu PID 5 vezes, espera que a criança morra, e retorna.

B

Este exercicio é também um pouco confuso, visto que seria talvez lógico que fosse mencionado um neto?

```c
int branchout(void(*child)()){
    int c = fork();
    if(!c)child();
    return c;
}

void grandchild_proc(){
    int ppid = getppid();
    for (int i = 0; i < 3; ++i)printf("Eu sou o neto, meu pai é %i\n", ppid);
    sleep(2);
}

void child_proc_first(){
    if(!branchout(grandchild_proc))return;
    if(!branchout(grandchild_proc))return;
    int ppid = getppid();
    int pid = getpid();
    for (int i = 0; i < 3; ++i)printf("Eu sou o filho, eu sou %i, e meu pai é %i\n", pid, ppid);
    wait(0);
    sleep(2);
}

void child_proc_latter(){
    int ppid = getppid();
    for (int i = 0; i < 3; ++i)printf("Eu sou o filho, meu pai é %i\n", ppid);
    sleep(2);
}


int main(int argc, char const *argv[]){
    if(!branchout(child_proc_first)) return 0;
    if(!branchout(child_proc_latter))return 0;
    if(!branchout(child_proc_latter))return 0;

    int p = getpid();
    for (int i = 0; i < 5; ++i)printf("Eu sou o pai, minha identificação é %i\n", p);
    wait(0);

    return 0;
}
```

```
trist(gayniggers):~/Documents/L201819/SO/PL2$ ./a
Eu sou o pai, minha identificação é 12816
Eu sou o pai, minha identificação é 12816
Eu sou o pai, minha identificação é 12816
Eu sou o pai, minha identificação é 12816
Eu sou o pai, minha identificação é 12816
Eu sou o filho, meu pai é 12816
Eu sou o filho, meu pai é 12816
Eu sou o filho, meu pai é 12816
Eu sou o filho, meu pai é 12816
Eu sou o neto, meu pai é 12817
Eu sou o filho, eu sou 12817, e meu pai é 12816
Eu sou o filho, meu pai é 12816
Eu sou o neto, meu pai é 12817
Eu sou o filho, eu sou 12817, e meu pai é 12816
Eu sou o filho, meu pai é 12816
Eu sou o neto, meu pai é 12817
Eu sou o neto, meu pai é 12817
Eu sou o filho, eu sou 12817, e meu pai é 12816
Eu sou o neto, meu pai é 12817
Eu sou o neto, meu pai é 12817
trist(gayniggers):~/Documents/L201819/SO/PL2$
```

Fiz também uma versão alternativa com apenas um print em cada, e fiz com que o filho esperasse pelo fim do pai, para demonstrar flow control.

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <wait.h>

int branchout(void(*child)()){
    int c = fork();
    if(!c)child();
    return c;
}

void grandchild_proc(){
    int ppid = getppid();
    printf("Eu sou o neto, meu pai é %i\n", ppid);
}

void child_proc_first(){
    printf("Eu sou o filho, eu sou %i, e meu pai é %i, e terei 2 filhos:\n", getpid(), getppid());
    if(!branchout(grandchild_proc))return;
    if(!branchout(grandchild_proc))return;
    wait(0);
}

void child_proc_latter(){
    printf("Eu sou o filho %i, meu pai é %i\n", getpid(), getppid());
}


int main(int argc, char const *argv[]){
    printf("Eu sou o pai, minha identificação é %i, terei 3 filhos e 2 netos:\n", getpid());
    if(!branchout(child_proc_first)) return 0;wait(0);
    if(!branchout(child_proc_latter))return 0;wait(0);
    if(!branchout(child_proc_latter))return 0;wait(0);

    return 0;
}
```

```
trist(gayniggers):~/Documents/L201819/SO/PL2$ ./a.out
Eu sou o pai, minha identificação é 12886, terei 3 filhos e 2 netos:
Eu sou o filho, eu sou 12887, e meu pai é 12886, e terei 2 filhos:
Eu sou o neto, meu pai é 12887
Eu sou o neto, meu pai é 12887
Eu sou o filho 12890, meu pai é 12886
Eu sou o filho 12891, meu pai é 12886
trist(gayniggers):~/Documents/L201819/SO/PL2$
```

P3

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

#define piperead  0
#define pipewrite 1
int pipeChildWrite[2] = {0};
int pipeParntWrite[2] = {0};

int branchout(void(*child)()){
    int c = fork();if(!c)child();return c;
}

void child_proc(){
    close(pipeChildWrite[piperead]);
    close(pipeParntWrite[pipewrite]);

    void kbInterruptHandlerChild(int dummy){}
    signal(SIGINT, kbInterruptHandlerChild);

    char currProcessing = 0;
    while(read(pipeParntWrite[piperead], &currProcessing, 1), currProcessing != '\n'){
        if     (currProcessing <= 'z' && currProcessing >= 'a')currProcessing -= 'a'-'A';
        else if(currProcessing <= 'Z' && currProcessing >= 'A')currProcessing += 'a'-'A';
        write(pipeChildWrite[pipewrite], &currProcessing, 1);
    }
    close(pipeChildWrite[pipewrite]);
    close(pipeParntWrite[piperead]);
    //puts("deadkid");
    return;
}

void endIt(){
    write(pipeParntWrite[pipewrite], "\n", 1); //this signals the child to kill itself
    wait(0);
    close(pipeParntWrite[pipewrite]);
    close(pipeChildWrite[piperead]);
}

int main(int argc, char const *argv[]){
    pipe(pipeParntWrite);pipe(pipeChildWrite);

    if(!branchout(child_proc))return 0;

    ////////////////////////////////////////////////////////////////////////////////////////////////////
    ////////////////////////////////////////////////////PARENT//////////////////////////////////////////
    ////////////////////////////////////////////////////////////////////////////////////////////////////
    close(pipeParntWrite[piperead]);
    close(pipeChildWrite[pipewrite]);

    void kbInterruptHandlerParent(int dummy){endIt(); exit(0);}
    signal(SIGINT, kbInterruptHandlerParent);

    char buffer[2000] = {0}; int bufferlen = 0;
    for (;fgets(buffer, 2000, stdin);){
        bufferlen = strlen(buffer);
        buffer[--bufferlen] = '\0'; //remove \n at the end

        write(pipeParntWrite[pipewrite], buffer, bufferlen);

        for (int i = 0; i < bufferlen; ++i){
            read (pipeChildWrite[piperead],  &buffer[i], 1);
        }

        puts(buffer);
    }

    endIt();

    return 0;
}
```

Aqui temos dois processos e duas pipes. Uma pipe para o pai comunicar com o filho e uma para o filho comunicar com o pai. O pai lê da consola, manda para o filho, e o filho, letra a letra, “inverte

os caracteres. Como é linha a linha, reservei o caracter ‘\n’ para ser o “sinal” para o filho terminar, visto que em mais nenhum senário o receberia.

```
trist(gaynigger):~/Documents/L201819/SO/PL2$ ./a.out
abcABC!"?
ABCabc!"?
asd jsd ASDdasD dEEF__ . !"  $#$$#$ aA4 !#$ 12 rRS
ASD JSD asdDASd Deef__ . !"  $#$$#$ Aa4 !#$ 12 Rrs
trist(gaynigger):~/Documents/L201819/SO/PL2$ ./a.out
^Ctrist(gaynigger):~/Documents/L201819/SO/PL2$
```

Temos agora aqui o teste e as duas opções de saida.

Os testes como dá para ver estão funcionais.

A primeira saida foi uma de EOF (ctrl+d), em que o pai manda um ‘\n’ ao filho, o que o faz terminar, e depois o pai sai também.

A segunda é uma de ctrl+C, em que utilizo a função signal(tambem POSIX) para dar override ao default behavior desse interrupt (que é terminar o programa abruptamente) para mandar primeiro uma mensagem ‘\n’ ao filho, esperar que o filho morra, e depois sair.

P4

Visto que tive problemas ao intrepertar o pedido no enunciado, resolvi desenvolver o senário que achei mais propício ao mostrar das minhas capacidades.

Um senário em que hexiste matéria-prima (raw materials), neste caso ilimitada, de onde ambos os produtores retiram. Os produtores iram de seguida, asyncronamente, produzir produto dessas matérias-primas, e, quando requirido pelo consumidor, enviar (também ao consumidor claro).

O consumidor neste caso tem um “stockpile”, que vai ter que encher com base nos produtores aos quais tem preferência.

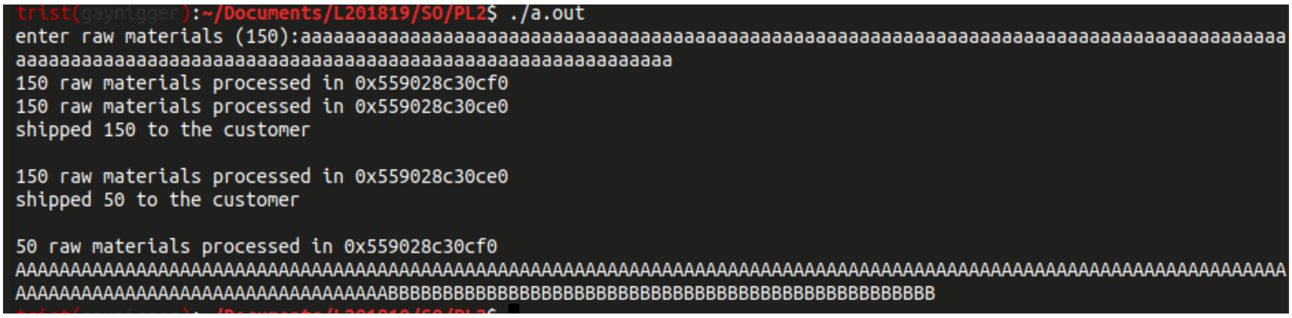
Se os produtores não tiverem quantidade suficiente para satisfazer o consumidor (eu por exemplo consumo uma quantidade absurda de atum e esparguete), passa ao próximo consumidor, até o cliente estar satisfeito.

Os produtores têm também armazenamento limitado, o que quer dizer que assim que o armazem fique cheio, ficam à espera que o consumidor tenha necessidade de mais. Quando o consumidor tem necessidade e o armazem não está cheio, o produtor para brevemente a produção para enviar para o consumidor, e resume imediatamente produção.

```c
#include <pthread.h>
#include <semaphore.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned maxProdQ = 200;
unsigned producerMaxStock = 150;

unsigned char* productsBuffer = 0;
unsigned maxProductsSp = 0;
unsigned productsSp = 0;

typedef void*(*sr_proto)(void*);


struct Producer{
    char (*processRawMaterial)(char);
    char* rawMaterialBuffr;
    unsigned rawMaterialAmount;
    char* product;
    unsigned productAmount;
    unsigned maxProductAmount;

    sem_t finishedWriting;
    sem_t demand;
    unsigned demandAmount;
    sem_t* ProdBuffer;
};void Producer_init(struct Producer* p){
    sem_init(&p->demand, 0, 0);
    sem_init(&p->finishedWriting, 0, 0);
}void Producer_term(struct Producer* p){
    sem_close(&p->demand);
    sem_close(&p->finishedWriting);
}

struct ProducerThread{
    struct Producer prod;
    pthread_t thread;
};
```

```c
void* ProducerWork(struct Producer* me){while(1){
    int loopTryWait = 0;            //will be used later
    unsigned i, oldpa = me->productAmount;
    //while we haven't filled up our capacity, and we don't have any requests
    while(me->productAmount != me->maxProductAmount && (loopTryWait = sem_trywait(&me->demand)) == -1){
        me->product[me->productAmount] = me->processRawMaterial(me->rawMaterialBuffr[me->productAmount]);   //create
        me->productAmount++;                                                                                //product
    }
    printf("%u raw materials processed in %p\n", me->productAmount - oldpa, me->processRawMaterial);
    if(loopTryWait == -1)sem_wait(&me->demand); //here if we exited the loop on purpose to push, we won't have to wait again
    if(me->demandAmount == 0)return 0;          //this is exit scenario

    sem_wait(me->ProdBuffer);                   //wait in case someone else is writing to the product buffer
    for (i = 0; i < me->demandAmount && me->productAmount; ++i) //write all we have or the demand amount
        productsBuffer[productsSp++] = me->product[--me->productAmount];    //depending on which is smallest
    sem_post(me->ProdBuffer);                   //make the buffer available for usage again
    printf("shipped %u to the customer\n\n", i);
    sem_post(&me->finishedWriting);             //make it so anyone can know we're done with "shipping" for now
}}                                              //and that are going back to producing

char prod1(char a){return 'A';}
char prod2(char a){return 'B';}
int main(int argc, char const *argv[]){
    if(argc > 1){
        maxProdQ = atoi(argv[1]);
    }if(argc > 2){
        producerMaxStock = atoi(argv[2]);
    }
    productsBuffer = malloc(maxProdQ);
    maxProductsSp = maxProdQ;
    memset(productsBuffer, '_', maxProductsSp);

    char* rawprod = malloc(producerMaxStock);
    printf("enter raw materials (%u):", producerMaxStock);
    scanf("%s", rawprod);

    sem_t ProdBuffer;
    sem_init(&ProdBuffer, 0, 1);

    struct ProducerThread producers[2] = {
        {{
            prod1, rawprod, producerMaxStock, malloc(producerMaxStock), 0, producerMaxStock, {0}, {0}, 0, &ProdBuffer
        }, 0},
        {{
            prod2, rawprod, producerMaxStock, malloc(producerMaxStock), 0, producerMaxStock, {0}, {0}, 0, &ProdBuffer
        }, 0}
    };

    Producer_init(&producers[0].prod);Producer_init(&producers[1].prod);
    pthread_create(&producers[0].thread, 0, (sr_proto)ProducerWork, &producers[0].prod);
    pthread_create(&producers[1].thread, 0, (sr_proto)ProducerWork, &producers[1].prod);

    while(productsSp != maxProductsSp){
        for (int i = 0; i < 2; ++i){
            producers[i].prod.demandAmount = maxProductsSp - productsSp;
            sem_post(&producers[i].prod.demand);
            sem_wait(&producers[i].prod.finishedWriting);
            //sleep(1);
            if(productsSp == maxProductsSp)break;
        }
    }

    printf("%*.*s\n", maxProdQ, maxProdQ, productsBuffer);
    sem_post(&ProdBuffer);

    producers[0].prod.demandAmount = 0;
    sem_post(&producers[0].prod.demand);
    producers[1].prod.demandAmount = 0;
    sem_post(&producers[1].prod.demand);

    pthread_join(producers[0].thread, 0);
    pthread_join(producers[1].thread, 0);

    free(rawprod); free(productsBuffer);
    free(producers[0].prod.product);free(producers[1].prod.product);
    sem_close(&ProdBuffer);
    Producer_term(&producers[0].prod);Producer_term(&producers[1].prod);
    return 0;
}
```

E aqui está a correr:



Em que ele pede os materiais, processa-os.

Temos dois produtores, vamos-lhes chamar f0 e e0.

O produtor f0 produz 150 (maximo de armazenamento), depois o produtor e0 faz o mesmo.

Como o cliente tem preferência pelo produtor e0, esse manda todo o seu produto para o consumidor (que tem uma procura de 200).

O cliente ainda não está satisfeito, por isso passa ao próximo produtor, f0. f0 manda 50 do seu produto para o consumidor.

Ao mesmo tempo que f0 envia para o consumidor, e0 está a produzir mais.

O consumidor espera que ambos os produtores acabem de lhe dar os produtos, e depois imprime de quem são. Temos 150 As do produtor e0, e 50 Bs do produtor f0.

Ao mesmo tempo o produtor estava a criar os 50 de produto que lhe faltavam.

Quando o consumidor manda uma procura de 0 para o produtor, ele “vai à falência” (a thread acaba). No fim de imprimir, ambos os produtores acabam, e o programa acaba.

Não consegui demonstrar os produtores a pararem a produção para enviar ao consumidor pois o meu processador é demasiado rápido, mas uma boa revisão do código indicará que tal acontece devido ao uso de sem_trywait.

Conclusão

Podémos assim não só ganhar uma apreciação geral sobre como o sistema operativo, comunicação entre processos, e multi-tasking funciona, mas também pelas várias formas como são e podem ser utilizadas estas técnicas.

Enterti-me a fazer os exercícios e espero por uma próxima.

Referências

Neste guião não houve referências pois todo o conhecimento presente na minha introdução teórica foi conhecimento que depositei de cabeça, e com outras cadeiras que me apanharam de surpresa e a minha própria irresponsabilidade, não tive tempo para ir procurar por todos os sítios onde fui buscar a quantidade razoável de conhecimento que sei por gosto. A bibliografia do meu guião prévio (que incluirei asseguir), explica melhor o meu background e o porquê de eu não ter feito uso da internet durante a realização deste trabalho.

Bibliografia

A grande maioria dos meus conhecimentos sobre sistemas operativos adequiri há uma quantidade consideravel de tempo quando fiz um kernel module para linux e um mini-sistema operativo para x86, por isso não irá estar aqui tudo, mas aqui vai a minha tentativa:

- wiki.osdev.org
- stackoverflow.com
- unix.stackexchange.com
- askubuntu.com
- guidedhacking.com
- man7.org/linux/man-pages/