Oppgave 1

Prosesser er utstyrt med sine egne virtuelle adresserom; deres virtuelle minner holdes privat – og dét med god grunn: Prosesser som tuklet med hverandes data helt vilkårlig ville medført kaos. For at prosesser likevel skal være i stand til å kommunisere og samarbeide, må det finnes en måte for dem å utveksle meldinger med hverandre eller, mer generelt, å få tilgang til et delt område hvor felles data er lagret. Dette kan oppnås på forskjellige måter, som postbokser, piper, delt minne og signaler.

Forestill deg at du skriver et program som skal multiplisere store matriser. For å gjøre programmet mer effektivt parallelliserer du algoritmen slik at flere gaffelprosesser kan arbeide med deler av oppgaven samtidig. Spesifikt: Hver parallell prosess må være i stand til å rapportere sitt resultat til foreldreprosessen, som beregner den endelige summen. Hva slags IPC vil du velge for å få dette til å fungere? Gi en kort redegjørelse for algoritmen (ingen kode påkrevet).

I dette tilfellet ville jeg valgt å bruke en *pipe*, siden det dekker behovet for funksjonalitet vi trenger fra vår IPC-metode. Vi bestemmer oss for at vi for eksempel ikke trenger *mailboxen* sin typing av meldinger, men at en FIFO-liste i en pipe fungerer helt greit for å kommunisere resultatene til foreldreprosessen.

En enkel algoritme er som følger:

1. Foreldreprosessen, la oss kalle den main(), vil opprette barneprosesser med kall på pipe(), la oss kalle de pc_0, pc_1, pc_2... pc_n.

2. Barneprosessene vil starte å fylle opp sin pipe med resultater fra beregningene de gjør. Dersom pipen blir full, eller write() er blokkert fordi main() leser fra den, må prosessen vente på tur til å skrive til pipen.

3. Underveis vil main() prøve å lese resultatene som ligger i pipene for å beregne den endelige summen. Selv om alle barneprosessene i teorien kunne brukt samme pipe i kommunikasjonen med main(), så er det kanskje lurt å ha flere pipes, slik at når main() leser fra pc_0 sin pipe så kan f.eks. pc_1 og pc_2 skrive inn resultater i sin pipe til main().

Ikke en god besvarelse, kanskje noen poeng men husker ikke begrunnelsen bak dette.

I. Vi ville valgt shared memory/delt minne, da hver gaffelprosess trenger å kunne hente ut hva som skal regnes ut, og etterpå sende resultatet til foreldreprosessen.

Vi lar her alle prosessen dele et segment minne hvor vi har to input matriser og en output matrise. Man lager en ny prosess for hver rad i venstre matrisen som hver regner ut en rad i output matrise. Dette går ved at hver gaffelprosess skriver til en forskjellig rad i output matrisen. Når alle gaffelprosessene har terminert, er foreldre prosessen fri til å ta bruk output matrisen.

Fullt score besvarelse

Forestill deg at du skriver tre programmer som skal arbeide i tandem: Det første programmet fanger live video fra et kamera. Det andre programmet konverterer videostrømmen til flere forskjellige kvaliteter. Det tredje programmet serverer de konverterte videostrømmene til internett. Hva slags IPC vil du velge for å få dette til å fungere og hvorfor?

Live videostrømming i original kvalitet krever typisk store minneområder, og krever også god throughput i sanntid for å levere den kvaliteten som brukerne ønsker. Postboks og pipes er alternativer dersom store nok bufre er tilgjengelig. Hvis bufrene er små, vil det gi mer overhead og flere systemkall med kontekstbytte ned i kjernen, noe som er kostbart. Pipes kan brukes mellom første og andre program. Andre program genererer ouput med tre forskjellige kvaliteter eller typer, og postboks kan brukes til å håndtere denne strømmen fordi den har mulighet for typer. Det tredje programmet vil da hente meldinger med ønsket kvalitet fra postboksen. Dersom bufrene blir for små med pipes og postboks er Shared Memory et alternativ. Da kan to og to programmer (hhv. første og andre pluss andre og tredje) dele et fysisk minneområde som er mappet inn i programmenes adresseområde. Andre program leser fra området det deler med første program, og skriver til området det deler med tredje program. Det vil kreve mer overhead til synkronisering og concurrency for å ivareta minneområdet som en delt ressurs.

Denne besvarelsen var meget god, manglet en konklusjon til å besvare hvilken IPC som skulle velges, men gruppen hadde god refleksjon ved å reflektere fordeler og ulemper ved de ulike kommunikasjonsmetodene.

Oppgave 2

Signaler brukes til å utløse bestemt atferd i en prosess, for eksempel avslutning eller feilhåndtering. De utgjør en enkel måte å samhandle med prosesser som er felles for alle UNIX-lignende operativsystemer, som OpenBSD, Linux og MacOS. (Windows bruker ikke signaler, i hvert fall ikke den POSIX-kompatible typen, men bruker andre former for interprosesskommunikasjon for å oppnå de samme målene.)

Du har allerede lært om avbrudd og unntak i forbindelse med CPU. Funksjonelt ligner signaler avbrudd ved at begge får prosessen til å avvike fra dens normale kjøreflyt. Hva er de viktigste forskjellene mellom dem?

Interrupts håndteres i kernelen som dispatcher riktig interrupt handler som fremgår av IDTen. De forårsakes av asynkrone hardware hendelser.

Signaler blir sendt fra kernelen til en spesifikk prosess (ofte først fra en annen prosess og så til kernelen). De kan forårsakes av både synkrone og asynkrone hendelser. Det er opp til prosessen selv å bestemme seg for hvordan den skal håndtere signalet.

Meget god besvarelse, viktigste forskjellen er at signaler kan hentes og redefineres.

Avbrudd håndteres av en interrupt handler, og kan kun maskeres eller legges i en kø dersom en oppgave med høyere prioritet kjører i kjernen. Signaler behandles av en signal handler og har en default utførelse, men kan fanges og redefineres.

Signaler er opprettet av software, men kan sendes basert på hendelser i både hardware, software, IO, osv. Avbrudd sendes ofte som elektroniske signal med opprinnelse i hardware, men de kan genereres av prosessoren.

Også en meget god besvarelse

Vanligvis kan en kjørende prosess tilknyttet terminalen avsluttes ved å trykke Ctrl+C på tastaturet. Dette sender SIGINT-signalet til prosessen. Imidlertid kan prosessen håndtere dette signalet på en annen måte og dermed overstyre standardoppførselen, som er å avslutte. Det samme gjelder SIGTERM, som vanligvis ikke har en hurtigtast. Generelt kan signal-systemanropet brukes til å sette egne signalbehandlere, som er vanlige funksjoner, på bestemte signaler. Er det på denne måten mulig å lage en prosess som nekter å dø av noe signal? Forklar svaret ditt.

Nei, fordi det finnes signaler som ikke kan redefineres. Ifølge gnu.org kan ikke signalet SIGKILL redefineres eller blokkeres.

Alice har oppdaget at en segmenteringsfeil faktisk får kjernen til å sende SIGSEGV-signalet til den skyldige prosessen. Dette signalet, som andre, kan overstyres. Prosessen trenger ikke krasje, argumenterer hun: Når signalet mottas, kan signalbehandleren ganske enkelt la prosessen fortsette sin glade ferd. Fra nå av vil segmenteringsfeil være en bekymring fra fortiden. Bob er skeptisk. Han vet ikke helt hvordan han skal sprekke Alices boble, men synes ideen høres for god ut til å være sann. Hvordan vil du argumentere?

I et tilfelle hvor man har satt SIGSEGV-signalet til å ignoreres vil den delen av koden som forårsaket segmenteringsfeilen kjøre på nytt. Siden man har ignorert signalet i stedet for å håndtere feilen vil man få segmenteringsfeil også når koden kjører igjen. Da vil programmet krasje med segmenteringsfeil, som det ville gjort i utgangspunktet.

Selvom “SIGSEGV-signalet” ignoreres, så er ikke feilen blitt unnagjort. Feilen vil allikevel skje og det kan være dumt å ignorere en slik signal siden det gjør det vanskeligere å dekode programmet.

Oppgave 3

IPC er et viktig konsept i operativsystemer; det går også foran det mer omfattende emnet som er nettverkskommunikasjon. Visse former for nettverkskommunikasjon kan egentlig sies å være en annen slags IPC – med det konseptuelle forbeholdet at flerprosesssystemer kan være distribuert på flere maskiner i et nettverk.

En pipe forbinder en prosess´ utdata med en annen prosess´ inndata. Den virker mellom prosesser på samme operativsystem gjennom fildeskriptorer for lesing og skriving, som peker til en delt minneside. For eksempel: Kommandoen ls | sort leder utdataene fra ls til inndatene til sort gjennom en pipe og gir en alfabetisk sortert liste over innholdet i gjeldende katalog.

Gi en oversikt over trinnene som skallet utfører når kommandoen kjøres.

1. `main`-prosessen startes og shellet leser standard input skrevet inn av brukeren ved bruk av `getline` funksjonen.
2. Streng tokenisering kalles og splitter kommandolinjen til tokens og tolker kommandoen `ls | sort`.
3. `main`-prosessen kaller `fork` funksjonen og lager `exec`-prosessen `ls`.
4. `main`-prosessen kaller `fork` funksjonen og lager `exec`-prosessen `sort`.
5. `main` kaller `pipe` funksjonen:
   a. Det allokeres plass i det virtuelle filsystemet til en midlertidig inode. Denne peker på en fysisk minneside.
   b. Pipen har en fildeskriptor for å lese filer som tilhører `sort`-prosessen, og en for skriving fra `ls` -prosessen, `fd[1]`.
   c. `ls`-prosessen sitt standard output endres til å skrive til pipen.
   d. `sort`-prosessen sin standard input omdirigeres til å lese input fra pipen.
6. `ls`-prosessen eksekveres og henter listen over filer og directories og bruker fd[1] til å skrive dataen over til den delte minnesiden.
7. `fd[0]` blir notifisert om at `fd[1]` har skrevet til minneområdet.
8. `sort` eksekveres og leser data fra delt minneområde ved bruk av `fd[0]`.
9. `write`-fildeskriptoren lukkes.
10. Filnavnene sammenlignes og sorteres linje for linje.
11. `read`-fildeskriptoren lukkes.
12. Sortert liste av filnavnene returneres.

Punkt 5 burde komme før punkt 3.) siden hvis rekkefølgen var slik over, så ville pipen ikke bli arvet. Problemet var at gruppen skrev mer enn hva oppgaven spurte om, siden man skulle kun skrive de stegene som gjøres ved shell. Pål sa at de ikke hadde fått noe bonuspoeng så husk å svar etter oppgavebeskrivelsen.

For å lage pipen kaller skallet pipe-funksjonen, som åpner to fildeskriptorer: pipefd[0], brukt til lesing, og pipefd[1], brukt til skriving. Hvorfor, tror du, brukes to filbeskrivelser, i stedet for bare én? Tross alt holder som regel en enkelt fildeskriptor for både lesing og skriving til vanlige filer.

Først og fremst er ikke en pipe lik en fil da den bare oppretter en FIFO-kø i minne, og man har kun muligheten til å lese køen fra "starten" og da uten å ignorere eller hoppe over data.

Når det gjelder vanlig filer kan man begynne lesingen mer eller mindre hvor man vil, og hoppe over så mye data i en fil man vil (f.eks. med fseek).

Med andre ord så er det fordi ved piping så skal man ikke hoppe over noe data, inkludert at man kan skrive til / lese fra fil-deskriptorene asynkront uten å måtte tenke alt for mye på concurrency og problemer rundt det. I tillegg så gjør det at man kan lukke fil deskriptorene henholdsvis etter når de når EoF.

Bonusoppgave: Forsøk å komme opp med en enkel måte å splitte kommandoen ved hjelp av piper slik at hver del kan kjøres på separate maskiner. Altså, du vil se listen fra den ene maskinen på den andre. Et hint er gitt i figur 1.

Figure 1: Hint til oppgave 3.

Første maskinen kan kjøre ls og pipe dette til netcat, som sender det som blir pipet til IP-adressen til maskin 2 på en oppgitt port. På den andre maskinen bruker vi netcat til å lytte på den oppgitte porten, og piper det som kommer inn videre til sort.

M1: ls | nc <ip-adresse> <port>

M2: nc -L -p <port> | sort