Oppgave 1:

I følgende program ser vi bruken av en named pipe (FIFO) mellom en forelderprosess og barneprosess.

Det er først to funksjoner, én for skriving og én for lesing. Inne i første funksjon «writer()» finner vi en prefefinert melding «Music from Big Pink!» som sendes/skrives til minnet, for å senere kunne bli lest. Etter å ha lagt meldingen i minnet vil funksjonen skrive ut «Message sent : Music from Big Pink!». Deretter vil funksjonen avsluttes ved at FIFO-en avsluttes med «close(fd)»

I neste funksjon «reader()» leses det som først ble sendt i «writer()». Her vil den lese meldinger fra FIFO-en med «read()» og lagrer det den leser i buffer (maksgrense på 99 tegn). Når meldingen er mottatt, vil den skrive det ut med en print, som vil se slik ut: «Message received: Music from Big Pink». Deretter vil funksjonen avsluttes med «close(fd)».

I selve programkoden, opprettes det først en variabel for pid for å lagre prosess-Idene. Deretter opprettes det en named pipe (FIFO) med filsti og filrettigheter som parametere. Videre opprettes det et barneelement av pid, før det rett etterpå sjekkes med et if-statement om det er et barneelement eller ikke. Hvis pid == 0 (barneprosess) skal «writer()»-funksjonen begynne.

Videre sjekkes det om pid ikke er lik 0. Er dette tilfellet skal «reader()»-funksjonen begynne. Etterpå finner vi systemtakket «wait(NULL)». Denne får foreldreprosessen til å vente til barneelementet er ferdig. I dette tilfellet venter forelderprosessen på at barneelementet skal skrive ferdig til FIFO-en og avslutte. Til slutt finner vi unlink. Denne brukes til å fjerne en fil fra systemet. Her vil den fjerne FIFO filen som ble brukt til å sende og lese meldingen. Ettersom FIFO-filer ikke slettes automatisk, må dette gjøres manuelt hver gang.

I konsollen vil det derfor bli skrevet ut to linjer:

Message sent : Music from Big Pink!

Message received: Music from Big Pink!

Oppgave 3:

Deloppgave A:

```
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ emacs write_shm.c
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ emacs read_shm.c
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ gcc -o write_shm write_shm.c -lrt
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ gcc -o read_shm read_shm.c -lrt
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ ./read_shm ; ./write_shm
sum1 = 0, sum2 = 50000005000000
```

Ved å kjøre begge programmene sekvensielt, får vi først og fremst samme resultatet som om vi hadde kjørt de hver for seg i samme rekkefølge. Grunnen til at vi får resultatet som vi gjør, er fordi har «write_shm» ikke har skrevet noe som «read_shm» kan lese. Grunnet dette, vil sum1 alltid forbli 0, ettersom denne rekkefølgen leser, før det skrives, og det er derfor ingenting å lese. Programmet vil likevel beregne sum2 riktig, ettersom telleren til sum2 er definert i «read_shm».

Deloppgave B:

```
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ ./write_shm ; ./read_shm
sum1 = 50000005000000, sum2 = 50000005000000
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$
```

Ved å kjøre programmene sekvensielt, men i «omvendt» rekkefølge oppnår vi det «riktige» resultatet. Siden write_shm har initialisert minnet, vil sum1 være riktig, og det vil samsvare med sum2.

Deloppgave C:

```
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ (./write_shm &) ; ./read_shm
sum1 = 149472248922565, sum2 = 50000005000000
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ (./write_shm &) ; ./read_shm
sum1 = 24824695066508, sum2 = 50000005000000
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$ (./write_shm &) ; ./read_shm
sum1 = 14687138723008, sum2 = 50000005000000
emilbe@itstud:~/htdocs/Intro2OS/oblig/7$
```

Denne måten å kjøre programmene på, kan kalles for parallell kjøring. Det som skjer her, er at «write_shm» vil kjøre i bakgrunnen, og «read_shm» vil umiddelbart kjøre deretter. Resultatet kan derfor variere avhengig om «write_shm» har rukket å skrive verdiene til minnet før «read_shm» begynner å lese, og også hvor langt «write-shm» har kommet. Denne måten å kjøre programmene på, støter derfor på problemet «race conditions»