Lösningsförslag

Systemprogrammering, 5p
31 oktober 2001

Uppgift 1 (4 p)

Nonblocking I/O I unblocked mode returnerar I/O-anrop omedelbart, med ett felmeddelande om operationen inte kunde genomföras. Kan t ex användas om man inte vet om I/O kan genomföras och man kan utföra annat arbete (kan prova att utföra I/O senare) om det inte gick.

Core dump En "bild" av processens minne sparas på en fil (core), tex när en process får vissa signaler. Denna fil kan användas av många debuggers för att undersöka tillståndet hos processen när den avbröts.

Set-user-ID bit En bit i en fils "mode-word" som säger att "effective UID" ska vara filägarens och inte exekverarens. Används om man tillfälligt vill att en process ska ha andra rättigheter. (Programmet kan växla mellan filägarens och exekverarens rättigheter.)

Semafor En semafor är en heltalsvariabel som bara kan anta icke-negativa värden. Variabeln kan *endast* kommas åt via två *atomära* operationer:

wait Om semaforens värde är större än noll minskas dess värde med ett, annars fördröjs den anropande processen tills dess att semaforens värde är större än noll.

signal Ökar semaforens värde med ett. Om det finns fördröjda processer så väcks en.

Används t ex för att synkronisera processer.

Uppgift 2 (2 + 2 p)

(a) Föräldern kan inte i efterhand få reda på sina barns pids utan måste spara returvärdet från *fork*. Barnen kan anropa *getppid()* för att få fram nuvarande förälders pid (barnet kan ha ärvts av *init*).

(b) När en förälderprocess avslutas händer inget speciellt för barnprocesserna som fortsätter att exekvera. Barnet ärvs av *init* som tar hand om returresultatet från barnet för att ingen zombie ska skapas.

När en barnprocess avslutas före sin förälderprocess så meddelas föräldern via en signal. Om föräldern anropat någon *wait*-funktion så returneras dessutom status till föräldern och barnet försvinner. Om föräldern inte gjort det så sparas statusinformationen (barnet blir en zombie) tills dess att föräldern

anropar en av *wait*-funktionerna eller avslutas och init-processen tar hand om statusen.

Uppgift 3 (4 p)

Till exempel:

Signaler Skicka mjukvaruinterrupt mellan processer som då kan ha signalhaterare som utför olika saker. Väldigt begränsad mängd data som kan överföras, dvs egentligen inget data förutom att man får en signal.

Pipes Via pipe() skapas två fildescriptorer som kan läsas från ena änden och skrivas till den andra. Måste vara skapad av en gemensam "anfader", och har också ett begränsat utrymme.

Shared memory Kan skapa en del i primärminnet som kan vara gemensamt för flera processer. Har ofta en ganska begränsad storlek och processerna måste finnas på samma maskin.

Filer Kan skapa filer som man kan läsa/skriva från/till. Här måste processerna känna till filnamnet och dessutom komma åt samma filsystem (dock behöver de inte finnas på sammma maskin).

Uppgift 4 (2 + 1 + 1 p)

(a) Göra `return` från `main`-funktionen eller anrop av funktionen `exit`. Dessa två metoder är ekvivalenta och innebär till skillnad från det tredje sättet (anrop av funktionen `_exit`) att man dessutom stänger alla standard I/O och anropar funktioner som registrerats av `atexit`.

(b) Processen avslutas om den får vissa signaler. T ex `^C` om man inte har sagt att annat ska göras.

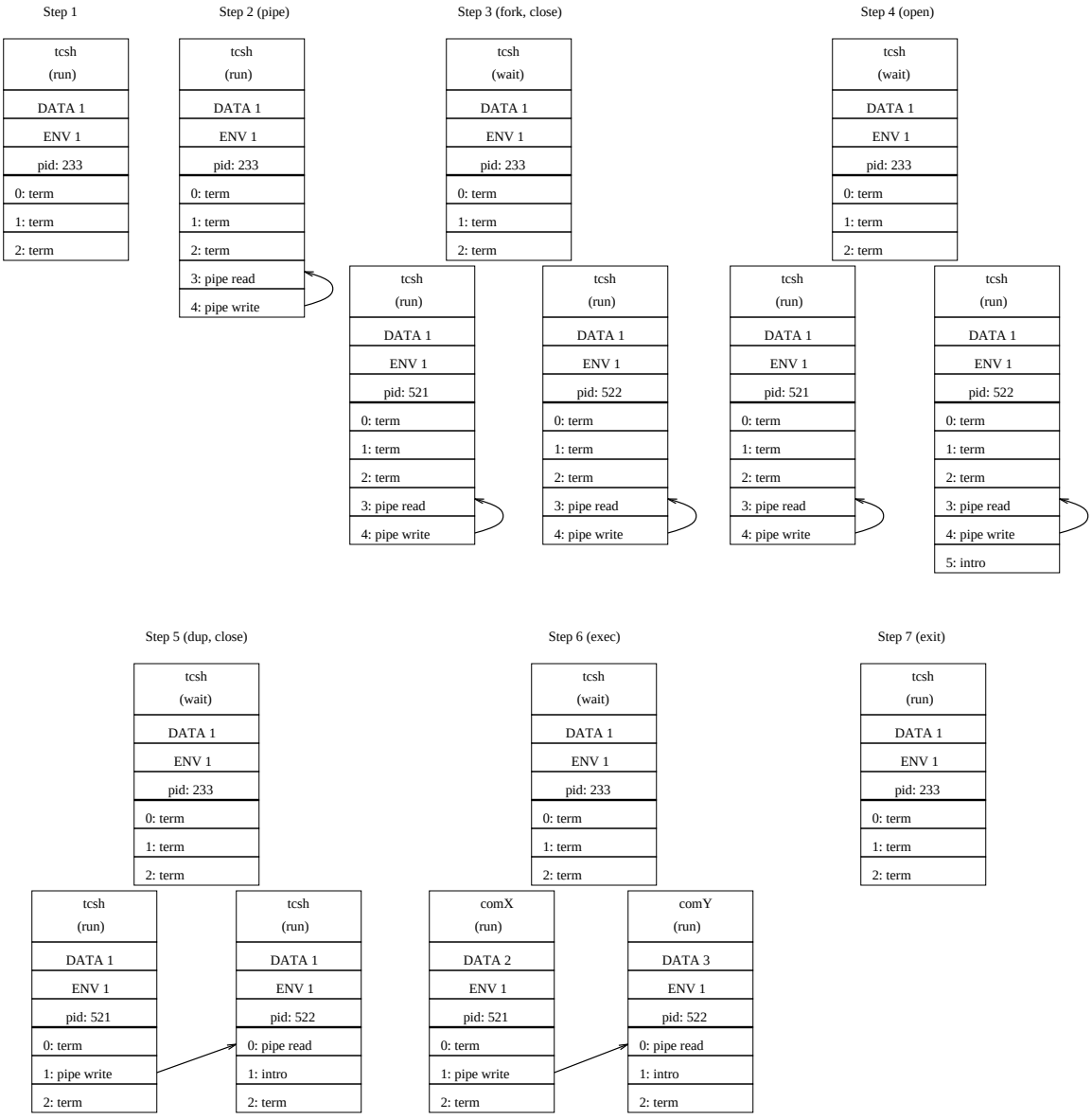
(c) T ex stänga alla öppna filer, lämna tillbaka primärminnet mm.

Uppgift 5 (2 + 3 p)

(a)

(b)
```
QBC
CIA
```

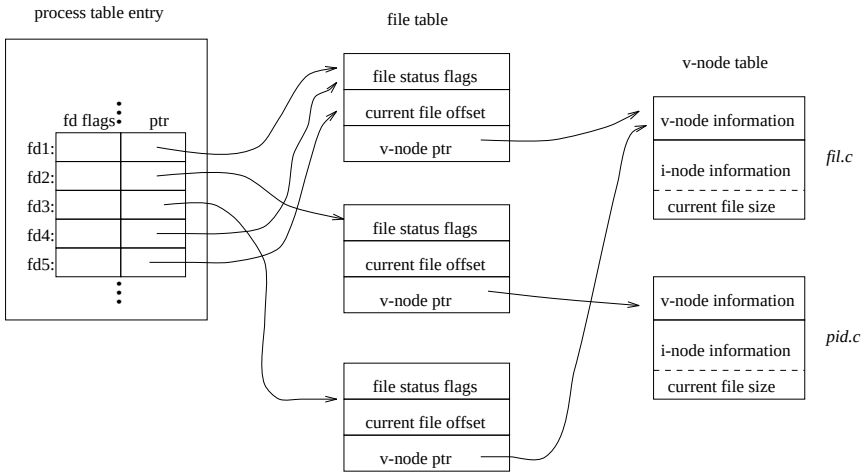
Uppgift 6 (4 p)



Lite mer text krävs också.

Uppgift 7 (3 + 2 p)

(a)



(b)

```
buf1: ABC
buf2: ABC
buf3: DEF
buf4: GHI
```

Uppgift 8 (4 p)

```
pid = 8183, ppid = 8182, var = 10, p = 0
pid = 8182, ppid = 8181, var = 11, p = 8183
pid = 8181, ppid = 8180, var = 8, p = 8182
pid = 8184, ppid = 8180, var = 8, p = 0
pid = 8180, ppid = 4622, var = 9, p = 8184
```

Uppgift 9 (6 p)

```
#include    <stdio.h>
#include    <sys/types.h>
#include    <unistd.h>

int main(int argc, char *argv[])
{
    int i;
    pid_t   pid;

    if (argc < 3) {
        fprintf(stderr, "Usage: %s file command [command ...]\n", argv[0]);
        exit(1);
    }
    for (i = 2; i < argc; i++)
        if ( (pid = fork()) < 0) {
            perror("fork error");
            exit(1);
        } else if (pid == 0) {
            execlp(argv[i], argv[i], argv[1], (char *) 0);
            perror(argv[i]);
            exit(1);
        }
    exit(0);
}
```