SIGNALER - GENERERING

- Signaler som bara kan genereras av användare i applikationer (dvs reserverade för applikationer.
  - `SIGUSR1`
  - `SIGUSR2`

SIGNALER – ICKE TILLFÖRLITLIGA

- Vissa äldre unix-system implementerar signaler på ett sätt som inte är tillförlitligt !
- I dessa är signaler inte är garanterade att verkligen komma fram till ditt program utan de kan försvinna på vägen.
- Det kan också vara problem med att själv skriva koden så att olika problem inte uppstår.

SIGNALER – VAD KAN MAN GÖRA?

- Strunta i signalen
  - Funkar för alla signaler (utom `SIGKILL` och `SIGSTOP`), men beroende på vad som genererade signalen så kan man få problem senare. Kommer inte ens fram till processen.
- Fånga signalen (ej `SIGKILL` och `SIGSTOP`)
  - Skriv kod som tar hand om signalen och ser till att göra nåt (naturligtvis olika saker beroende på vad som har hänt).
- Default action
  - Låt den fördefinierade koden ta hand om saken.

SIGNALER

- Core dump
  - Fil med tillståndet för en process när den avslutas, kan användas för att ta reda på orsaken till varför processen avslutades. Genereras vid vissa signaler.
  - Kan begränsa storleken i sitt skal. För att se/sätta de gränser som finns används olika kommandon i olika skal.

    ```
    limit (i tcsh)
    ulimit (i bash)
    ```

    visar även saker som stack- och filstorlek

NÅGRA SIGNALER OCH DESS DEFAULT

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGWINCH	28,28,20	Ign	Window resize signal (4.3BSD, Sun)

FÅNGA SIGNALER

- Signalhanterare kan registreras mha funktionen (funkar ej tillförlitligt på alla system, se senare exempel för alt. impl)

```
void (*signal (int sig, void (*disp)(int)))(int);
```

eller med en `typedef`:

```
typedef void Sigfunc(int)
Sigfunc *signal(int, Sigfunc *);
```

FÅNGA SIGNALER

- Tre konstanter i signal.h
  - `#define SIG_DFL (void (*)(int))0`
    - Används för att sätta defaultbeteende
  - `#define SIG_ERR (void (*)(int))-1`
    - Returneras av signal om något går fel
  - `#define SIG_IGN (void (*)(int))1`
    - Används för att ignorera signalen
- Om signalhanteraren returnerar kommer koden att fortsätta där den avbröts av signalen.
- Även funktionen `sigaction` kan användas i detta syfte. Mer krånglig (och också mer uttrycksfull), men funkar alltid.

SPECIELLA SIGNALER

- Det finns några signaler som är lite speciella då de inte uppträder som alla andra.
- `SIGABRT`
  - är speciell i den meningen att du kan definiera en avbrottshanterare men om ett avbrott inträffar så kommer programmet att avbrytas. Det vill säga: efter det att din kod har exekverats så kommer inte kontrollen tillbaka till programmet utan programmet avslutas.
- `SIGSTOP, SIGKILL`
  - är speciella i den meningen att de inte går att ta hand om, om du försöker så lyckas du inte.

SIGNALER UNDER SYSTEMANROP

- Vad händer om man får en signal medan man gör ett systemanrop?
- Principen: om man har gjort ett "långsamt" systemanrop så avbryts detta anrop och man får tillbaka ett fel och felkoden (`errno`) är satt till `EINTR`
- Varför: jo, om man fångar en signal så är det antagligen något man är intresserad av och man bör därför hoppa tillbaka så fort som möjligt
- Vissa system startar numera om ett flertal av de långsamma anropen som standard
- POSIX tillåter att de startas om, men har det ej som ett krav

SIGNALER – LÅNGSAMMA ANROP

- Vad är då långsamma anrop? Jo, det är systemanrop som kan blockeras "för evigt":
  - Filläsning
    - De som kan blockeras för evigt om data inte finns tillgängligt (pipe, terminaler och nätverk)
  - Filskrivning
    - Samma som ovan

SIGNALER – LÅNGSAMMA ANROP

- Vad är då långsamma anrop? Jo, det är systemanrop som kan blockeras "för evigt":
  - `open`
    - Open till filer som kan blockeras för evigt (t.ex. modem).
  - `pause` och `wait`
  - `ioctl`
  - en del interprocess grejor
- Disk IO-anrop gäller inte
  - De kommer alltid tillbaka

REENTRANT FUNCTIONS

- Kan en funktion anropas igen samtidigt som den redan exekveras? Ibland, men ibland inte och gör man ett andra anrop på en funktion som inte är reentrant så ligger man pyrt till.
- Tabell 10.4 beskriver de funktioner som garanterat är reentrant och därmed med säkerhet kan anropas i signalhanterare.
- Följande måste gälla för att en funktion ska vara reentrant:
  - Håller inte reda på statiskt data
  - Skyddar globalt data genom att ta en lokal kopia av det
  - Får inte anropa andra funktioner som inte är reentrant
  - Får ej returnera referens till static-deklarerat data, och allt data tillhandahålls av anroparen.

NÅGRA FUNKTIONER SOM ÄR REENTRANT

_Exit()	fcntl()	mkdir()	setpgid()	socketpair()
_exit()	fork()	mkfifo()	setsid()	stat()
abort()	fstat()	open()	setuid()	sysconf()
accept()	getegid()	pause()	shutdown()	time()
access()	geteuid()	pipe()	sigaction()	times()
alarm()	getgid()	poll()	sigaddset()	umask()
bind()	getgroups()	pselect()	sigdelset()	uname()
chdir()	getpid()	raise()	sigemptyset()	unlink()
chmod()	getppid()	read()	sigfillset()	utime()
chown()	getsockname()	recv()	sigismember()	wait()
close()	getsockopt()	recvmsg()	signal()	waitpid()
connect()	getuid()	rename()	sigpending()	write()
creat()	kill()	rmdir()	sigprocmask()	
dup()	link()	select()	sigset()	
dup2()	listen()	send()	sigsuspend()	
execle()	lseek()	sendmsg()	sleep()	
execve()	lstat()	setgid()	socket()	

SKICKA SIGNALER

- `kill`

  `int kill(pid_t pid, int sig);`

  – Skickar signalen sig till PID (en process eller grupp av processer)

  – PID måste vara en process vars real eller effective UID överensstämmer med anroparens effective eller real UID

- `raise`

  `int raise(int sig);`

  – Med hjälp av denna funktion så kan en process skicka en signal till sig själv.

- `alarm`

  – Används ofta i kombination med `pause`

SIGNALER – ALARM

`unsigned int alarm(unsigned int sec)`

- Ställer alarmklockan som tillhör den anropande processen att skicka signalen `SIGALRM` till processen efter *sec* sekunder.
- Endast ett larm kan vara inställt för en process.
- Om *sec* är 0, kommer ev. tidigare larm att tas bort.
- `fork` stänger av alla ev. alarm för nya processer. Exec-funktionerna förändrar ej ev. alarm som är satta för processen som anropar `exec`.
- Man får nödvändigtvis inte en alarmsignal efter 5 sekunder om man gör anropet alarm(5)! Det enda man kan vara säker på man får en alarmsignal någon gång efter 5 sekunder

SIGNALER - PAUSE

`int pause(void);`

– Stoppar upp den anropande processen tills den mottar en signal. Signalen måste vara en som inte ignoreras.

– Om signalen får processen att terminera så kommer `pause` ej att returnera.

– Om signalen fångas av den anropande processen och kontrollen återgår till programmet från signalhanteringsfunktionen, kommer den att fortsätta exekvera koden efter `pause`-anropet.

– Med `alarm` och `pause` kan man låta en process somna och sedan vakna så att den kan göra nåt speciellt.

- `sigsuspend` lite mer avancerat systemanrop som kan åstadkomma samma sak

BLOCKERA SIGNALER

- Det finns funktioner för att förhindra signaler (man får välja vilka man vill förhindra) från att inträffa.
- Bör göras för viktiga partier i koden där någon viss signal inte får inträffa
- Signal sets
  - Det finns möjlighet att hantera flera signaler samtidigt genom att definiera ett "signal set", se kap 10.11. Genom att sedan använda `sigprocmask`, 10.12 kan du sätta vilka signaler som för tillfället ska blockeras, etc. Om du vill kolla vilka signaler som väntar på att bli "levererade" så kan du använda `sigpending`, 10.13.

sigaction

- Efterföljare till `signal`-funktionen
- `int sigaction(int signo, const struct sigaction *restrict act, struct sigaction *restrict oact);`
- `struct sigaction` har ett fält för signalhanteraren, ett med ett signal set som sätts innan signalhanteraren körs (efteråt återställs den till det den var innan). OS:et lägger automatiskt till den inträffade signalen här (så att den blockeras)
- boken har en implementation av `signal` mha `sigaction` (10.18)

signal() MHA sigaction()

```
Sigfunc *signal(int signo, Sigfunc *func) {
    struct sigaction act, oact;
    act.sa_handler = func;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (signo == SIGALRM) {
#ifdef SA_INTERRUPT
        act.sa_flags |= SA_INTERRUPT;
#endif
    } else {
#ifdef SA_RESTART
        act.sa_flags |= SA_RESTART;
#endif
    }
    if (sigaction(signo, &act, &oact) < 0)
        return(SIG_ERR);
    return(oact.sa_handler);
}
```

REALTIDSASPEKTER

- Som ni har sett så finns det en hel del problem som hör ihop med signaler. Det gäller att vara noggrann och se till att man inte har några missar i koden, tyvärr är detta inte så enkelt utan det är enkelt att missa.
- Dessa problem är inte något som enbart har med signalhantering att göra utan det ingår i det mer allmänna problemet med realtidsprogrammering.
- Det är mycket svårt att göra ett "säkert" system utan att ha tillgång till atomiska operationer. Om man tittar på de instruktioner som finns för olika processorer så finns det ofta en operation som är just odelbar, kanske "test-and-set", som sedan kan användas för att implementera odelbara funktioner på en högre nivå.

KOMMUNIKATION

SYSTEM

- kör ett shell-kommando

```
#include <stdlib.h>
int system(const char *string);
```

- system-funktionen skickar string till shellet, som om string hade skrivits som ett kommando i terminalen. Anroparen väntar tills kommandot avslutats.
- Returnerar exit-statusen av shellet i samma format som `waitpid`.

NAMNGIVNA PIPES – FIFO

- Ett annat namn för namngivna pipes är FIFOs. Den begränsning som finns för pipes är att de fungerar bara om processerna har en gemensam förfader som skapat pipan. Om så inte är fallet så måste man istället använda FIFOs.
- En FIFO har en sökväg och genom denna så kan flera processer öppna 'pipen'
- Processer som ska kommunicera måste vara på samma maskin fast FIFO:n är synlig i filsystemet.

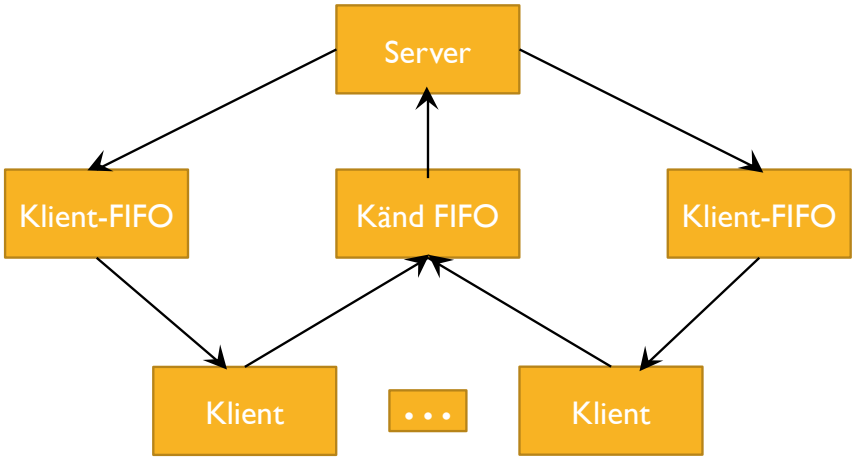
NAMNGIVNA PIPES – FIFO

- Skapas med

```
mkfifo(const char *path, mode_t mode);
```

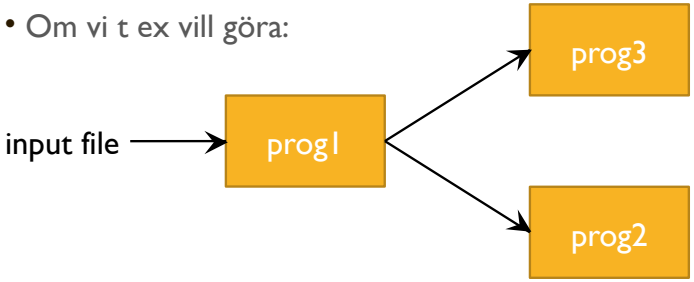
  - Returnerar `0` om ok, annars `-1` (`errno` sätts)
- Kan även skapas med kommandot `mkfifo`
- Öppnas och används som en vanlig fil (vanligen med `open, read, write, close`)
- OBS finns kvar efter körningen om den inte explicit tas bort med t ex `unlink`
- Har liksom en vanlig pipe en begränsad buffert (ligger i minnet, ej på disk)
- När den sista skrivaren till FIFO:n stängs genereras `EOF`

FIFO – CLIENT/SERVER



FIFO – "PIPELINE"

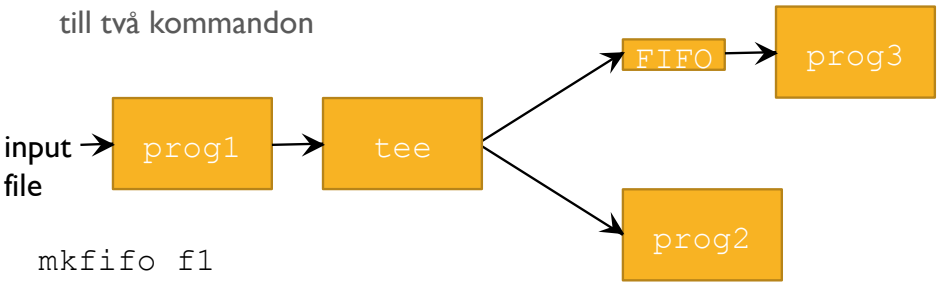
- Används av skal-kommandon för att skicka data från en pipeline till en annan, utan temp-filer.
- Om vi t ex vill göra:




```
prog1 < inputfile | prog2   ??prog3??
```

FIFO – "PIPELINE"

- Använd en `FIFO` och kommandot `tee` för att åstadkomma spliten till två kommandon




```
mkfifo f1
prog3 < f1&
prog1 < inputfile | tee f1 | prog2
```