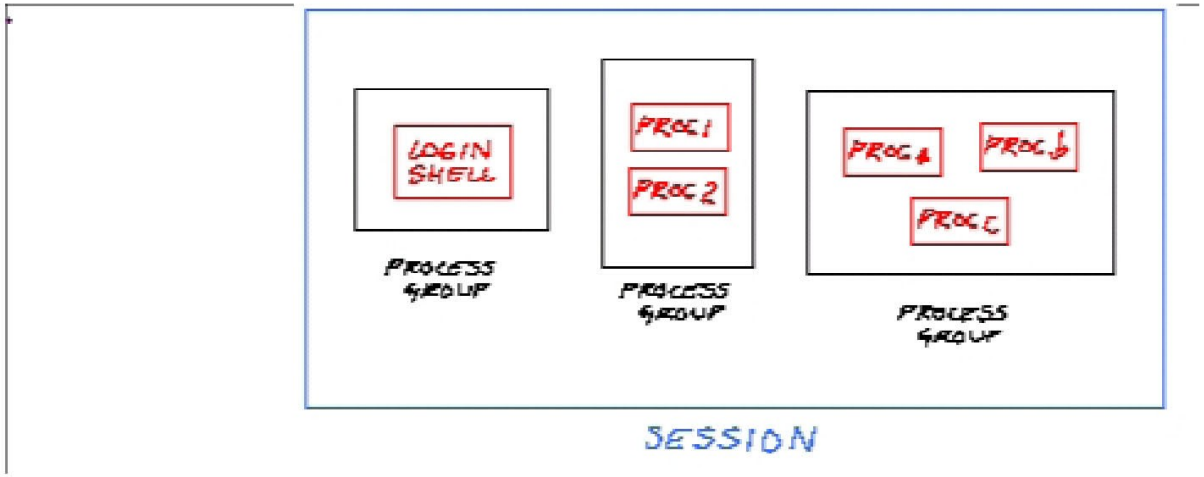
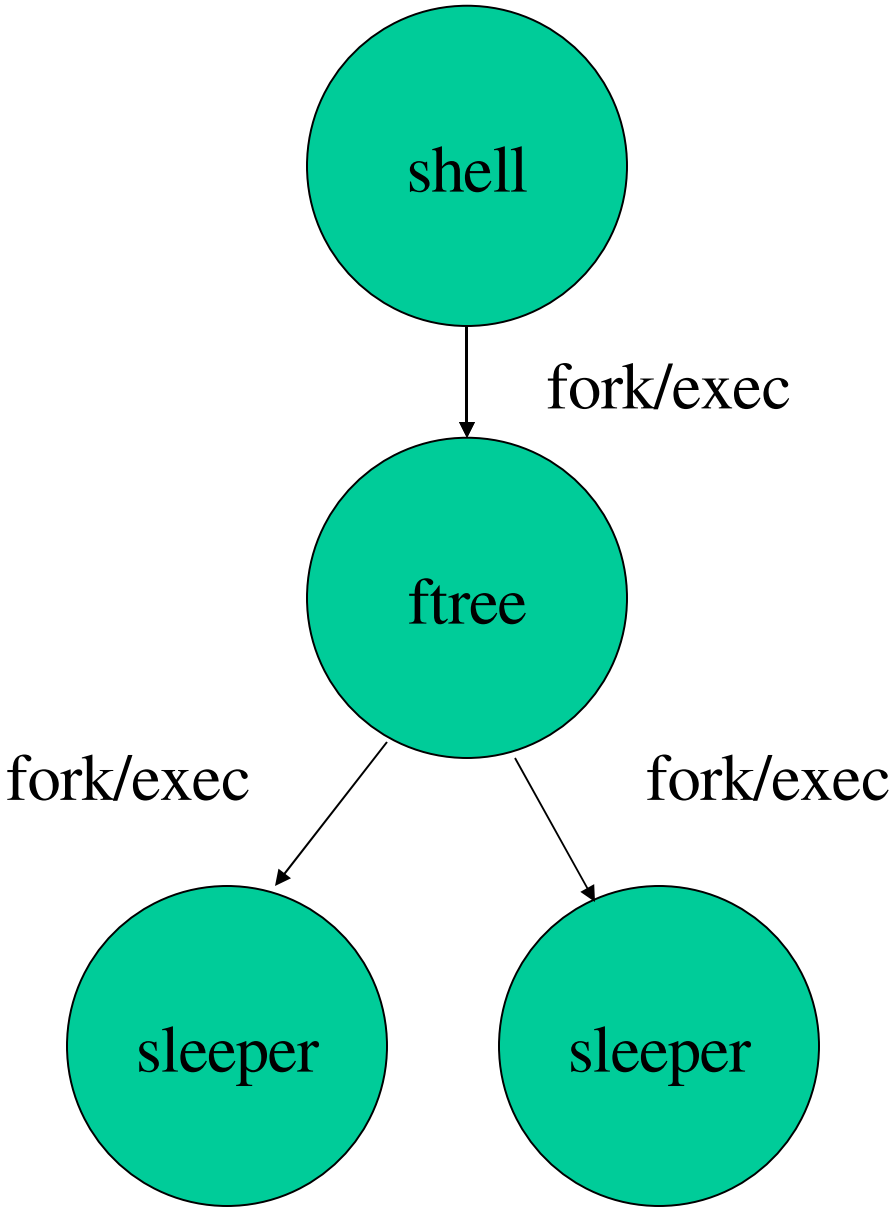
Process Groups

- Job control acheived using *process groups* and *sessions*
  - ⋆ Control access of multiple processes to single terminal
- Process group is collection of one or more processes
  - ⋆ Each process in some group; each group has unique id
  - ⋆ Group can have leader (leader PID= process group ID)
  - ⋆ Group exists as long as some process in group
  - ⋆ Process can set process group id of itself or one of its children
  - ⋆ Cannot change process group id of child once child calls `exec`

Sessions

- Session is collection of one or more process groups
- Process (not group leader) creates new session via `setsid`
  - ⋆ Process becomes session leader; only process in session
  - ⋆ Process becomes process group leader of a new group
  - ⋆ Process has no controlling terminal





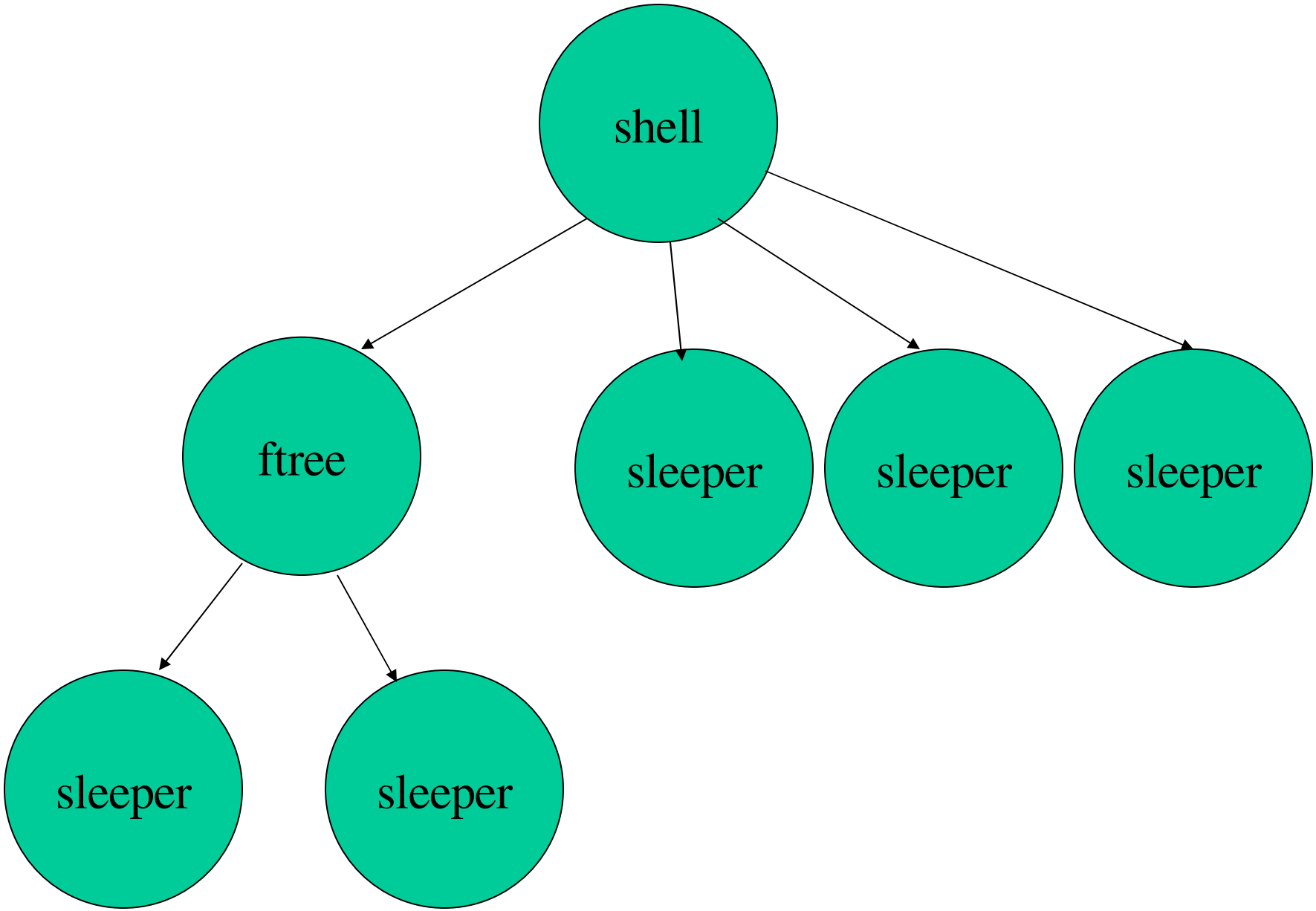
Session ID ?
Process Group ?

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

main()
{

  int ret;
  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper","5",NULL);}

  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper","5",NULL);}

  wait(NULL);
  wait(NULL);
}
```

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main(int argc,char **argv){

  printf("Process <%d> sleeping <%d>\n",getpid(),atoi(argv[1]));
  sleep(atoi(argv[1]));
  printf("Process <%d> exits\n",getpid());
}
```

```
[jmayo@asimov termio]$ ./ftree
Process <30061> sleeping <5>
Process <30062> sleeping <5>
Process <30061> exits
Process <30062> exits
[jmayo@asimov termio]$
```

```
[jmayo@asimov ~]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
17548 17549 17549 17549 pts/3    30060 S    26095   0:00 -tcsh
23051 23052 23052 23052 pts/4    30072 S    26095   0:00 -tcsh
17549 30060 30060 17549 pts/3    30060 S    26095   0:00 ./ftree
30060 30061 30060 17549 pts/3    30060 S    26095   0:00 sleeper 5
30060 30062 30060 17549 pts/3    30060 S    26095   0:00 sleeper 5
23052 30072 30072 23052 pts/4    30072 R    26095   0:00 ps -jx
[jmayo@asimov ~]$
```


shell
ftree
sleeper
sleeper
sleeper
sleeper
sleeper

```
[jmayo@asimov termio]$ ./sleeper 60 &
[1] 14003
[jmayo@asimov termio]$ Process <14003> sleeping <60>
./sleeper 60 &
[2] 14004
[jmayo@asimov termio]$ Process <14004> sleeping <60>
./sleeper 60 &
[3] 14005
[jmayo@asimov termio]$ Process <14005> sleeping <60>
./ftree
Process <14007> sleeping <5>
Process <14008> sleeping <5>
Process <14007> exits
Process <14008> exits
[jmayo@asimov termio]$ Process <14003> exits
[1]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$ Process <14004> exits
[2]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$ Process <14005> exits
[3]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$
```

```
[jmayo@asimov termio]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
13900 13901 13901 13901 pts/0    14006 S    26095   0:00 -tcsh
13948 13949 13949 13949 pts/2    14009 S    26095   0:00 -tcsh
13901 14003 14003 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14004 14004 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14005 14005 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14006 14006 13901 pts/0    14006 S    26095   0:00 ./ftree
14006 14007 14006 13901 pts/0    14006 S    26095   0:00 sleeper 5
14006 14008 14006 13901 pts/0    14006 S    26095   0:00 sleeper 5
13949 14009 14009 13949 pts/2    14009 R    26095   0:00 ps -jx
[jmayo@asimov termio]$
```

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

main(int argc,char **argv)
{

  int ret;
  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper",argv[1],NULL);}

  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper",argv[1],NULL);}

  wait(NULL);
  wait(NULL);
}
```

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

main(int argc, char **argv)
{

  int cpid;

  cpid=fork();
  if (cpid==0){ execlp("ftree","ftree",argv[1],NULL);}

  cpid=fork();
  if (cpid==0){ execlp("ftree","ftree",argv[1],NULL);}

  wait(NULL);
  wait(NULL);
}
```

```
[jmayo@asimov termio]$ ./ftreeNgrp 20
Process <17599> sleeping <20>
Process <17601> sleeping <20>
Process <17600> sleeping <20>
Process <17602> sleeping <20>
Process <17599> exits
Process <17601> exits
Process <17600> exits
Process <17602> exits
[jmayo@asimov termio]$
```

```
[jmayo@asimov termio]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
13900 13901 13901 13901 pts/0    17596 S    26095   0:00 -tcsh
13948 13949 13949 13949 pts/2    17603 S    26095   0:00 -tcsh
13901 17596 17596 13901 pts/0    17596 S    26095   0:00 ./ftreeNgrp 20
17596 17597 17596 13901 pts/0    17596 S    26095   0:00 ftree 20
17596 17598 17596 13901 pts/0    17596 S    26095   0:00 ftree 20
17597 17599 17596 13901 pts/0    17596 S    26095   0:00 sleeper 20
17597 17600 17596 13901 pts/0    17596 S    26095   0:00 sleeper 20
17598 17601 17596 13901 pts/0    17596 S    26095   0:00 sleeper 20
17598 17602 17596 13901 pts/0    17596 S    26095   0:00 sleeper 20
13949 17603 17603 13949 pts/2    17603 R    26095   0:00 ps -jx
[jmayo@asimov termio]$
```

NAME

setpgid, getpgid - set/get process group

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);
```

DESCRIPTION

setpgid sets the process group ID of the process specified by pid to pgid. If pid is zero, the process ID of the current process is used. If pgid is zero, the process ID of the process specified by pid is used. If setpgid is used to move a process from one process group to another (as is done by some shells when creating pipelines), both process groups must be part of the same session. In this case, the pgid specifies an existing process group to be joined and the session ID of that group must match the session ID of the joining process.
getpgid returns the process group ID of the process specified by pid. If pid is zero, the process ID of the current process is used.

Process groups are used for distribution of signals, and by terminals

to arbitrate requests for their input: Processes that have the same process group as the terminal are foreground and may read, while others will block with a signal if they attempt to read. These calls are thus used by programs such as csh(1) to create process groups in implementing job control. The TIOCGPGRP and TIOCSPGRP calls described in termios(3) are used to get/set the process group of the control terminal.

If a session has a controlling terminal, CLOCAL is not set and a hangup occurs, then the session leader is sent a SIGHUP. If the session leader exits, the SIGHUP signal will be sent to each process in the foreground process group of the controlling terminal.

If the exit of the process causes a process group to become orphaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal will be sent to each process in the newly-orphaned process group.

RETURN VALUE

On success, setpgid returns zero. On error, -1 is returned, and errno is set appropriately.

getpgid returns a process group on success. On error, -1 is returned, and errno is set appropriately.

ftree.c

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

main(int argc,char **argv)
{

  int ret;

  setpgid(0,0);
  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper",argv[1],NULL);}

  ret=fork();
  if (ret==0){ execlp("sleeper","sleeper",argv[1],NULL);}

  wait(NULL);
  wait(NULL);
}
```

```
[jmayo@asimov termio]$ ./ftreeNgrp 20
Process <17660> sleeping <20>
Process <17663> sleeping <20>
Process <17662> sleeping <20>
Process <17661> sleeping <20>
Process <17660> exits
Process <17663> exits
Process <17662> exits
Process <17661> exits
[jmayo@asimov termio]$
```

```
[jmayo@asimov termio]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
13900 13901 13901 13901 pts/0    17657 S    26095   0:00 -tcsh
13948 13949 13949 13949 pts/2    17665 S    26095   0:00 -tcsh
13901 17657 17657 13901 pts/0    17657 S    26095   0:00 ./ftreeNgrp 20
17657 17658 17658 13901 pts/0    17657 S    26095   0:00 ftree 20
17657 17659 17659 13901 pts/0    17657 S    26095   0:00 ftree 20
17658 17660 17658 13901 pts/0    17657 S    26095   0:00 sleeper 20
17658 17661 17658 13901 pts/0    17657 S    26095   0:00 sleeper 20
17659 17662 17659 13901 pts/0    17657 S    26095   0:00 sleeper 20
17659 17663 17659 13901 pts/0    17657 S    26095   0:00 sleeper 20
13949 17665 17665 13949 pts/2    17665 R    26095   0:00 ps -jx
```

NAME

kill - send signal to a process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>

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

DESCRIPTION

The kill system call can be used to send any signal to any process group or process.

If pid is positive, then signal sig is sent to pid.

If pid equals 0, then sig is sent to every process in the process group of the current process.

If pid equals -1, then sig is sent to every process except for process 1 (init), but see below.

If pid is less than -1, then sig is sent to every process in the pro-

cess group -pid.

If sig is 0, then no signal is sent, but error checking is still performed.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

```
void handler(int sig){
  printf("Process <%d> exits on signal <%d>\n",
         getpid(),sig);
  fflush(stdout);
  exit(0);
}

main() {
  int cpid,i
  signal(SIGUSR1,handler);

  for (i=0;i<5;i++){
    cpid=fork();
    if (cpid==0){while(1==1);}
    printf("Parent spawns pid <%d>\n",cpid);
    fflush(stdout);
}

  printf("Parent pid is <%d>\n",getpid());
  sleep(5);   /* Give kids time to spawn */
  kill(0,SIGUSR1);
}
```

```
[jmayo@asimov ~/codeExamples]$ ./killAll
Parent spawns pid <31317>
Parent spawns pid <31318>
Parent spawns pid <31319>
Parent spawns pid <31320>
Parent spawns pid <31321>
Parent pid is <31316>
Process <31316> exits on signal <10>
Process <31319> exits on signal <10>
Process <31320> exits on signal <10>
Process <31317> exits on signal <10>
Process <31318> exits on signal <10>
Process <31321> exits on signal <10>
```

NAME

setsid - creates a session and sets the process group ID

SYNOPSIS

```
#include <unistd.h>

pid_t setsid(void);
```

DESCRIPTION

setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session, the process group leader of the new process group, and has no controlling tty. The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session.

RETURN VALUE

The session ID of the calling process.

Controlling Terminal

- Session can have single *controlling terminal*
  - Terminal device, pseudo-terminal device
- Session leader that establishes connection to controlling terminal is *controlling process*
- Session process groups divided into: single foreground group, one or more background groups
- Process group id of foreground group associated with terminal is TPGID

```
[jmayo@asimov termio]$ ./sleeper 60 &
[1] 14003
[jmayo@asimov termio]$ Process <14003> sleeping <60>
./sleeper 60 &
[2] 14004
[jmayo@asimov termio]$ Process <14004> sleeping <60>
./sleeper 60 &
[3] 14005
[jmayo@asimov termio]$ Process <14005> sleeping <60>
./ftree
Process <14007> sleeping <5>
Process <14008> sleeping <5>
Process <14007> exits
Process <14008> exits
[jmayo@asimov termio]$ Process <14003> exits
[1]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$ Process <14004> exits
[2]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$ Process <14005> exits
[3]    Exit 22                       ./sleeper 60
[jmayo@asimov termio]$
```

```
[jmayo@asimov termio]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
13900 13901 13901 13901 pts/0    14006 S    26095   0:00 -tcsh
13948 13949 13949 13949 pts/2    14009 S    26095   0:00 -tcsh
13901 14003 14003 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14004 14004 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14005 14005 13901 pts/0    14006 S    26095   0:00 ./sleeper 60
13901 14006 14006 13901 pts/0    14006 S    26095   0:00 ./ftree
14006 14007 14006 13901 pts/0    14006 S    26095   0:00 sleeper 5
14006 14008 14006 13901 pts/0    14006 S    26095   0:00 sleeper 5
13949 14009 14009 13949 pts/2    14009 R    26095   0:00 ps -jx
[jmayo@asimov termio]$
```

Direct Access to Terminal

- Operating system connects stdin, stdout, stderr to terminal
  - Descriptors 0, 1, 2
- Can open terminal directly to get access
  - Work around redirection

NAME

ctermid - get controlling terminal name

SYNOPSIS

```
#include <stdio.h>

char *ctermid(char *s);
```

DESCRIPTION

ctermid() returns a string which is the pathname for the current controlling terminal for this process. If s is NULL, a static buffer is used, otherwise s points to a buffer used to hold the terminal pathname. The symbolic constant L_ctermid is the maximum number of characters in the returned pathname.

RETURN VALUE

The pointer to the pathname.

CONFORMING TO

POSIX.1

Example

- Want to access terminal directly
  - Allow access regardless of redirection of stdin/stdout
- Example program has two functions
  - Prompt user for input - accessing terminal through file descriptor created by opening return from ctermid; echo user input via same descriptor
  - Read/echo from descriptor 0/1

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(){

  int fd;
  char ch;

     fd=open(ctermid(NULL),O_RDWR);
     if (fd < 0){perror("open");exit(1);}

     write(fd,"Enter word:",14);
     read(fd,&ch,1);
     while (ch != 'q'){ write(fd,&ch,1);read(fd,&ch,1);}
     read(fd,&ch,1);

     while (read(0,&ch,1)!=0) write(1,&ch,1);
}
```

```
[jmayo@asimov termio]$ cat datafile
FILE DATA
[jmayo@asimov termio]$ ./wcterm < datafile
Enter word:adfas
adfas
q
FILE DATA
[jmayo@asimov termio]$
```

UNIX Job Control

- Signals from keyboard go to all in foreground group (TPGID)
- Background process that attempts to read from terminal gets SIGTTIN
  1. Job is stopped
  2. Shell detects and notifies user
  3. Shell command fg causes shell to put job in foreground; sends SIGCONT
- Action on output from background process specified via stty (more later)
  - Can get SIGTTOU, process stopped
  - Can allow access to terminal

Example

- Create multiple foreground processes
- Send signal from keyboard
- All should terminate

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void handler(){
  printf("Sleeper process <%d> caught SIGINT\n");
  exit(1);
}

main(int argc,char **argv){

  signal(SIGINT,handler);
  printf("Process <%d> sleeping <%d>\n",getpid(),atoi(argv[1]));
  sleep(atoi(argv[1]));
  printf("Process <%d> exits\n",getpid());
}
```

```
[jmayo@asimov ~]$ ./ftree 60
Process <9987> sleeping <60>
Process <9988> sleeping <60><CTRL-C>
Sleeper process <9986> caught SIGINT
Sleeper process <9986> caught SIGINT
```

Example

- Start process *P* in background
- When *P* receives SIGINT, try to read from STDIN
- Expect that:
  - *P* starts off in other than terminal process group
  - On signal, *P* will be stopped (default action for SIGTTIN, SIGTTOU)
- Put *P* in foreground, will change terminal process group id to *P*'s

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

char overflow[100];

void handler(){
  puts("Enter string: ");
  gets(overflow);
}
main(int argc, char **argv){

  signal(SIGINT,handler);
  sleep(atoi(argv[1]));
}
[jmayo@asimov ~]$
```

```
[jmayo@asimov ~]$ ./sigEntry 120&
[2] 7064
[jmayo@asimov ~]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 6981  6982  6982  6982 pts/0     7065 S    26095   0:00 -tcsh
 6982  7064  7064  6982 pts/0     7065 S    26095   0:00 ./sigEntry 120
 6982  7065  7065  6982 pts/0     7065 R    26095   0:00 ps -jx
[jmayo@asimov ~]$
[jmayo@asimov ~]$ Enter string:

[2]  + Suspended (tty input)         ./sigEntry 120
[jmayo@asimov ~]$ fg
./sigEntry 120
1234
[jmayo@asimov ~]$
```

```
[jmayo@asimov ~]$ kill -INT 7064
[jmayo@asimov ~]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 6981  6982  6982  6982 pts/0     6982 S    26095   0:00 -tcsh
 6982  7064  7064  6982 pts/0     6982 T    26095   0:00 ./sigEntry 120
 7069  7070  7070  7070 pts/2     7111 S    26095   0:00 -tcsh
 7070  7111  7111  7070 pts/2     7111 R    26095   0:00 ps -jx
```

```
[jmayo@asimov ~]$ ps -jx
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
 6981  6982  6982  6982 pts/0     7064 S    26095   0:00 -tcsh
 6982  7064  7064  6982 pts/0     7064 S    26095   0:00 ./sigEntry 120
 7069  7070  7070  7070 pts/2     7112 S    26095   0:00 -tcsh
 7070  7112  7112  7070 pts/2     7112 R    26095   0:00 ps -jx
```

Example

- Start processes in background
- Have them write to stdout
- Determine default behavior
- Modify the default via stty
- Observe change

simpleBg.c

```
main(){
  printf("Process <%d>\n",getpid());
}
```

```
[jmayo@asimov ~/codeExamples]$ ./simpleBg&; ./simpleBg&
[1] 14379
[2] 14380
[jmayo@asimov ~/codeExamples]$ Process <14379>
Process <14380>

[2]    Exit 16                          ./simpleBg
[jmayo@asimov ~/codeExamples]$
[1]    Exit 16                          ./simpleBg
[jmayo@asimov ~/codeExamples]$
```

```
[jmayo@asimov ~/codeExamples]$ stty tostop
[jmayo@asimov ~/codeExamples]$ ./simpleBg & ; ./simpleBg &
[1] 14420
[2] 14421
[jmayo@asimov ~/codeExamples]$
[1]  + Suspended (tty output)         ./simpleBg
[jmayo@asimov ~/codeExamples]$
[2]  + Suspended (tty output)         ./simpleBg
[jmayo@asimov ~/codeExamples]$ fg
./simpleBg
Process <14421>
[jmayo@asimov ~/codeExamples]$ fg
./simpleBg
Process <14420>
[jmayo@asimov ~/codeExamples]$ stty -tostop
```

```
[jmayo@asimov ~/codeExamples]$ stty -tostop
[jmayo@asimov ~/codeExamples]$ ./simpleBg & ; ./simpleBg &
[1] 14442
[2] 14443
[jmayo@asimov ~/codeExamples]$ Process <14442>
Process <14443>

[1]    Exit 16                          ./simpleBg
[jmayo@asimov ~/codeExamples]$
[2]    Exit 16                          ./simpleBg
```

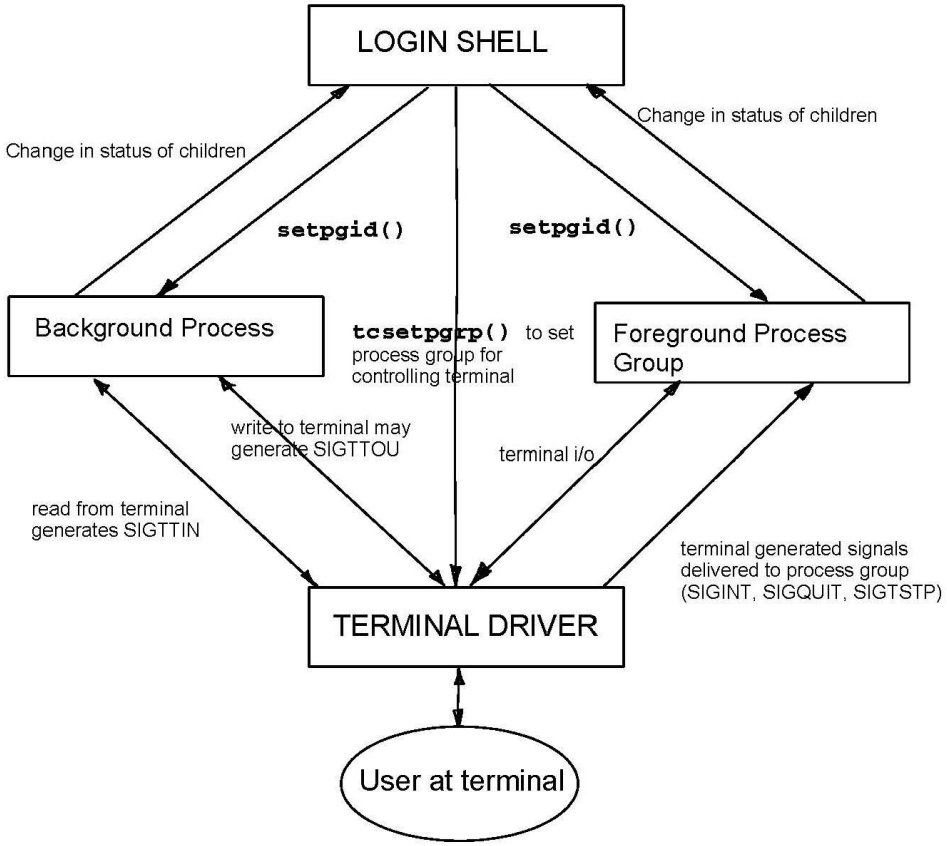


Figure based on one that appears in "Advanced Programming in the UNIX Environment", by W. Richard Stevens

NAME

tcgetpgrp, tcsetpgrp - get and set terminal foreground process group

SYNOPSIS

```
#include <unistd.h>

pid_t tcgetpgrp(int fd);

int tcsetpgrp(int fd, pid_t pgrp);
```

DESCRIPTION

The function tcgetpgrp() returns the process group ID of the foreground process group on the terminal associated to fd, which must be the controlling terminal of the calling process.

The function tcsetpgrp() makes the process group with process group ID pgrp the foreground process group on the terminal associated to fd, which must be the controlling terminal of the calling process, and still be associated with its session. Moreover, pgrp must be a (nonempty) process group belonging to the same session as the calling process.

If tcsetpgrp() is called by a member of a background process group in ts session, and the calling process is not blocking or ignoring SIGTTOU, a SIGTTOU signal is sent to all members of this background process group.

RETURN VALUE

When fd refers to the controlling terminal of the calling process, the function tcgetpgrp() will return the foreground process group ID of that terminal if there is one, and some value larger than 1 that is not presently a process group ID otherwise. When fd does not refer to the controlling terminal of the calling process, -1 is returned, and errno is set appropriately.

When successful, tcsetpgrp() returns 0. Otherwise, it returns -1, and errno is set appropriately.

Example

- Fork process
- Child calls setpgrp to create new process group
- Parent group unchanged
- See who is foreground, who is background

```
main(){

   int cpid;

   cpid=fork();
   if (cpid==0){
       setpgid(0,0);
       sleep(20);
   }
   wait(NULL);
}
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
15668	15669	15669	15669	pts/4	15775	Ss	26095	0:00	-tcsh
15669	15775	15775	15669	pts/4	15775	S+	26095	0:00	./toggle
15775	15776	15776	15669	pts/4	15775	S	26095	0:00	./toggle

Example

- Demonstrate control of foreground group within application
  1. Shell throws P into foreground group
  2. P creates C; C put into background as before
  3. P writes to stdout; gives C terminal
  4. C writes to stdout – should not block now
  5. C gives terminal to P
  6. P writes to stdout; should not block
  7. Plus some synchronization ….

```
#include <stdio.h>
int x=1;
void handler(int sig){x=0;}
main(){

  int cpid;
  signal(SIGUSR1, handler);
  cpid=fork();

  if (cpid==0){
    setpgid(0,0);
    while (x==1);      /* Blocked until SIGUSR1 */
    printf("Child's write does not block\n");
    fflush(stdout);
    tcsetpgrp(0,getppid());
    kill(getppid(),SIGUSR1);
    sleep(10);  _exit(0);
  }
```

```
  printf("Parent's first write does not block.\n");
  fflush(stdout);
  tcsetpgrp(0,cpid);
  kill(cpid,SIGUSR1);
  while (x==1);          /* Blocked until SIGUSR1 */
  printf("Parent's second write does not block. \n");
}
```

```
[jmayo@asimov ~/codeExamples]$ stty tostop
[jmayo@asimov ~/codeExamples]$ ./toggle2
Parent's first write does not block.
Child's write does not block
Parent's second write does not block.
[jmayo@asimov ~/codeExamples]$ stty -tostop
```

Terminal I/O

- Terminal I/O used for many things
  - Terminals, hardwired lines between computers, modems, printers, etc.
- Numerous attributes associated with terminal
- Messy to document and use
- Two modes:
  1. Canonical: terminal input processed as lines
     - Line terminated by newline character, eof character, eol character
  2. Noncanonical: terminal input not assembled into lines

Example

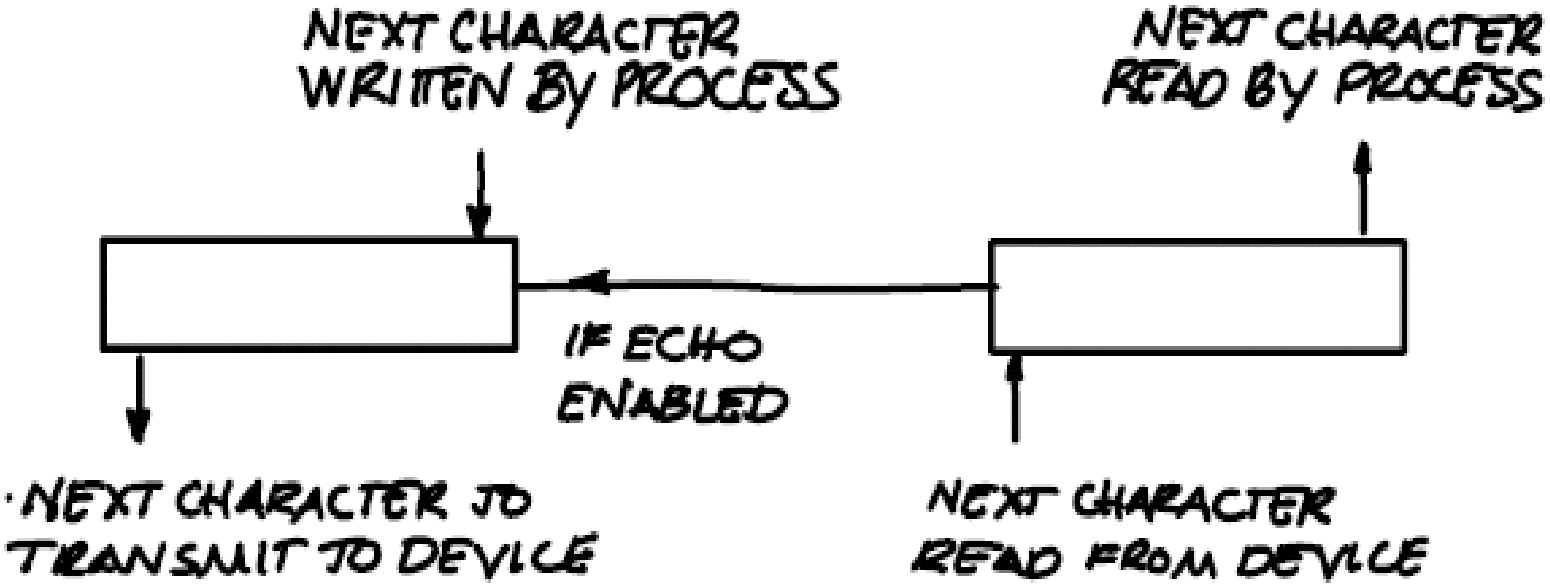
- In loop:
  - Read character from terminal
  - Write character to terminal
- Canonical mode => input processed line by line

```
[jmayo@asimov ~/codeExamples]$ cat chEcho.c
main(){
  int c;
  int n;
  n=0;
  while ( (c=getchar()) != 'q'){
    write(1,&c,1);
  }
}
```

```
[jmayo@asimov ~/codeExamples]$ ./chEcho
abcd
abcd
efghijk
efghijk
q
```

Terminal Device

- Terminal device controlled by terminal driver
- Each device has input queue and output queue
  - Implied link between queues when echo is enabled
  - Input queue may have finite size (MAX_INPUT)
  - Full queue behavior system dependent; UNIX usually bell
  - MAX_CANON is maximum chars in canonical input line


NEXT CHARACTER WRITTEN BY PROCESS
NEXT CHARACTER READ BY PROCESS
IF ECHO ENABLED
· NEXT CHARACTER TO TRANSMIT TO DEVICE
NEXT CHARACTER READ FROM DEVICE

Terminal Attributes

See man page for `termios`

```
struct termios {
    tcflag_t c_iflag;    /* input flags */
    tcflag_t c_oflag;    /* output flags */
    tcflag_t c_cflag;    /* control flags */
    tcflag_t c_iflag;    /* local flags */
    cc_t     c_cc[NCCS]; /* control characters */
}
```

c_iflag	c_oflag	c_cflag	c_lflag	cc_t
IGNBRK	OPOST	CBAUD	ISIG	VINTR
BRKINT	OLCUC	CBAUDEX	ICANON	VQUIT
IGNPAR	ONCLR	CSIZE	XCASE	VERASE
PARMRK	OCRNL	CSTOPB	ECHO	VKILL
INPCK	ONOCR	CREAD	ECHOE	VEOF
ISTRIP	ONLRET	PARENB	ECHOK	VMIN
INLCR	OFILL	PARODD	ECHONL	VEOL
IGNCR	OFDEL	HUPCL	ECHOCTL	VTIME
ICRNL	NLDLY	CLOCAL	ECHOPRT	VEOL2
IUCLC	CRDLY	LOBLK	ECHOKE	VSWTCH
IXON	TABDLY	CIBAUD	DEFECHO	VSTART
IXANY	BSDLY	CRTSCTS	FLUSHHO	VSTOP
IXOFF	VTDLY		NOFLSH	VSUSP
IMAXBEL	FFDLY		TOSTOP	VDSUSP
			PENDIN	VLNEXT
			IEXTEN	VWERASE
				VREPRINT
				VDISCARD
				VSTATUS

```
#include <termios.h>
#include <unistd.h>

int tcgetattr(int fd, struct termios *termios_p);
```

tcgetattr() gets the parameters associated with the object referred by fd and stores them in the termios structure referenced by termios_p. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

```
#include <termios.h>
#include <unistd.h>

int tcsetattr(int fd, int optional_actions, struct termios *termios_p);
```

tcsetattr() sets the parameters associated with the terminal (unless support is required from the underlying hardware that is not available) from the termios structure referred to by termios_p. optional_actions specifies when the changes take effect:

TCSANOW
: the change occurs immediately.

TCSADRAIN
: the change occurs after all output written to fd has been transmitted. This function should be used when changing parameters that affect output.

TCSAFLUSH
: the change occurs after all output written to the object referred by fd has been transmitted, and all input that has been received but not read will be discarded before the change is made.

Example

- Demonstrate modification of terminal attributes
  1. Read, print characters from `stdin` as before
  2. Modify so that uppercase input is automatically converted to lowercase

```
#include <stdio.h>
#include <termios.h>

void echoChar(){
  char ch;

  printf("Enter input, 'q' to quit\n");

  ch=getchar();
  while(ch != 'q'){
       printf("Got character %c code %d\n",ch,ch);
       ch=getchar();
  }

  //-- Read past \n after q
  ch=getchar();

}
```

```
main(){

  struct termios term;
  char           ch;

  if (isatty(0) == 0){
    perror("isatty");exit(1);}

  if (tcgetattr(0, &term) <0){
    perror("tcgetattr");exit(2);}

  //-- Set the bit that controls conversion of uppercase to lowercase
  term.c_iflag |= IUCLC;
  if (tcsetattr(0, TCSAFLUSH, &term) < 0){perror("tcsetattr");exit(3);}
  echoChar();

  //-- Clear the bit that controls conversion of uppercase to lowercase
  term.c_iflag &= ~IUCLC;
 if (tcsetattr(0, TCSAFLUSH, &term) < 0){perror("tcsetattr");exit(4);}
  echoChar();
}
```

Execution – type ⟨SHIFT⟩ABCDEFG at each prompt

```
 ./modtty
Enter input, 'q' to quit
abcdefg
Got character a code 97
Got character b code 98
Got character c code 99
Got character d code 100
Got character e code 101
Got character f code 102
Got character g code 103
Got character
 code 10
q
Enter input, 'q' to quit
ABCDEFG
Got character A code 65
Got character B code 66
Got character C code 67
Got character D code 68
Got character E code 69
Got character F code 70
```

```
Got character G code 71
Got character
 code 10
q
[jmayo@asimov ~]$
```

Example

- Read password from user
  - Make sure password is NOT echoed to the screen

Read Password[1]

```
/* readpass - read a password without echoing it */
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

int main(void) {
   struct termios ts, ots;
   char passbuf[1024];

   /* get and save current termios settings */
   tcgetattr(STDIN_FILENO, &ts);
   ots = ts;

   /* change and set new termios settings */
   ts.c_lflag &= ~ECHO;
   ts.c_lflag |= ECHONL;
   tcsetattr(STDIN_FILENO, TCSAFLUSH, &ts);
```

[1] Taken from *Linux Application Development*, Johnson and Troan

```
    /* paranoia: check that the settings took effect */
    tcgetattr(STDIN_FILENO, &ts);
    if (ts.c_lflag & ECHO) {
       fprintf(stderr, "Failed to turn off echo\n");
       tcsetattr(STDIN_FILENO, TCSANOW, &ots);
       exit(1);
    }

    /* get and print the password */
    printf("enter password: ");
    fflush(stdout);
    fgets(passbuf, 1024, stdin);
    printf("read password: %s", passbuf);
    /* there was a terminal \n in passbuf */

    /* restore old termios settings */
    tcsetattr(STDIN_FILENO, TCSANOW, &ots);

    exit(0);
}
```

```
[jmayo@asimov ~]$ ./readpass
enter password:
read password: AsDfG
[jmayo@asimov ~]$
```

```
#include <termios.h>
#include <unistd.h>

int tcflush(int fd, int queue_selector);
```

tcflush() discards data written to the object referred to by fd but not transmitted, or data received but not read, depending on the value of queue_selector:

TCIFLUSH
: flushes data received but not read.

TCOFLUSH
: flushes data written but not transmitted.

TCIOFLUSH
: flushes both data received but not read, and data written but not transmitted.

```
#include <termios.h>
#include <unistd.h>

int tcsendbreak(int fd, int duration);
```

tcsendbreak() transmits a continuous stream of zero-valued bits for a specific duration, if the terminal is using asynchronous serial data transmission. If duration is zero, it transmits zero-valued bits for at least 0.25 seconds, and not more that 0.5 seconds. If duration is not zero, it sends zero-valued bits for duration*N seconds, where N is at least 0.25, and not more than 0.5.

If the terminal is not using asynchronous serial data transmission, tcsendbreak() returns without taking any action.

```
#include <termios.h>
#include <unistd.h>

int tcdrain(int fd);

int tcflow(int fd, int action);
```

tcdrain() waits until all output written to the object referred to by fd has been transmitted.

```
#include <termios.h>
#include <unistd.h>

int tcflow(int fd, int action);
```

tcflow() suspends transmission or reception of data on the object referred to by fd, depending on the value of action:

TCOOFF suspends output.

TCOON restarts suspended output.

TCIOFF transmits a STOP character, which stops the terminal device from transmitting data to the system.

TCION transmits a START character, which starts the terminal device transmitting data to the system.

The default on open of a terminal file is that neither its input nor its output is suspended.

NAME

stty - change and print terminal line settings

SYNOPSIS

```
stty [-F DEVICE] [--file=DEVICE] [SETTING]...
stty [-F DEVICE] [--file=DEVICE] [-a|--all]
stty [-F DEVICE] [--file=DEVICE] [-g|--save]
```

DESCRIPTION

Print or change terminal characteristics.

-a, --all
print all current settings in human-readable form

-g, --save
print all current settings in a stty-readable form

-F, --file=DEVICE
open and use the specified DEVICE instead of stdin

--help display this help and exit

--version
       output version information and exit

Optional - before SETTING indicates negation. An * marks non-POSIX settings. The underlying system defines which settings are available.

```
[jmayo@asimov ~]$ stty -a
speed 38400 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon ixoff
-iuclc ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

Terminfo, curses

- `terminfo` is a database describing terminals by giving a set of capabilities which they have, by specifying how to perform screen operations, and by specifying padding requirements and initialization sequences
- `curses` is a set of functions to enable manipulation of the terminal display regardless of terminal type

Terminal Login

- `init` reads `/etc/inittab`
  - For every login terminal device entry: `fork/exec` specified `getty` (agetty,mingetty,mgetty)
- `getty` opens terminal device (`gettytab`)
- File descriptors 0,1,2 set to the device
- `getty` creates environment for `login` (`TERM=foo`)
- `getty` calls `exec()` of `login`
- `login`:
  - change owner of terminal device, change permissions
  - ...
  - calls `exec()` of login shell
- `init` is parent of login shell; will respawn getty on logout

```
#
# inittab       This file describes how the INIT process should set up
#               the system in a certain run-level.
#
# Author:       Miquel van Smoorenburg, <miquels@drinkel.nl.mugnet.org>
#               Modified for RHS Linux by Marc Ewing and Donnie Barnes
#

# Default runlevel. The runlevels used by RHS are:
#   0 - halt (Do NOT set initdefault to this)
#   1 - Single user mode
#   2 - Multiuser, without NFS (The same as 3, if you do not have networking)
#   3 - Full multiuser mode
#   4 - unused
#   5 - X11
#   6 - reboot (Do NOT set initdefault to this)
#
id:3:initdefault:

# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit

l0:0:wait:/etc/rc.d/rc 0
l1:1:wait:/etc/rc.d/rc 1
l2:2:wait:/etc/rc.d/rc 2
l3:3:wait:/etc/rc.d/rc 3
```

```
l4:4:wait:/etc/rc.d/rc 4
l5:5:wait:/etc/rc.d/rc 5
l6:6:wait:/etc/rc.d/rc 6

        ...

# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6

# Run xdm in runlevel 5
# xdm is now a separate service
x:5:respawn:/etc/X11/prefdm -nodaemon
```