Contents

Process Control

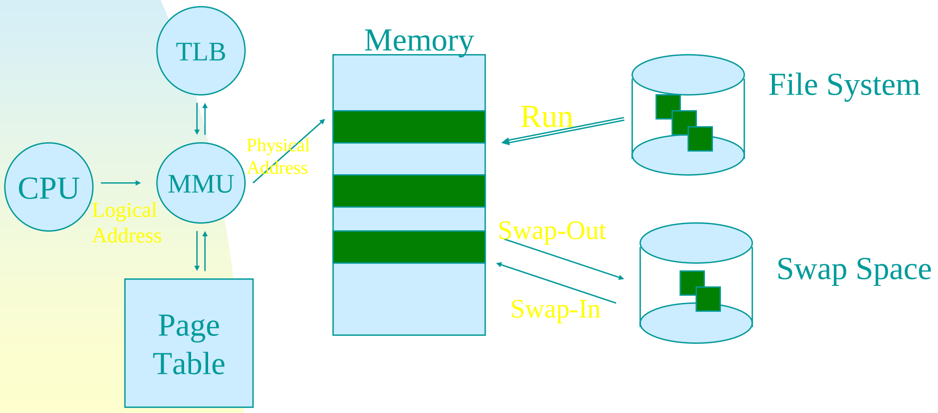
- Objective
  - Process Control: Process Creation and Termination, Program Execution, etc.
  - Process Properties and Accounting
    - E.g., ID's.
  - Related Functions
    - E.g., system()
- Process Identifiers
  - Process ID – a nonnegative unique integer
    - tmpnam

Process Control

- Special Processes
  - PID 0 – *Swapper* (I.e., the scheduler)
    - Kernel process
    - No program on disks correspond to this process
  - PID 1 – *init* responsible for bringing up a Unix system after the kernel has been bootstrapped. (/etc/rc* & init or /sbin/rc* & init)
    - User process with superuser privileges
  - PID 2 - pagedaemon responsible for paging
    - Kernel process

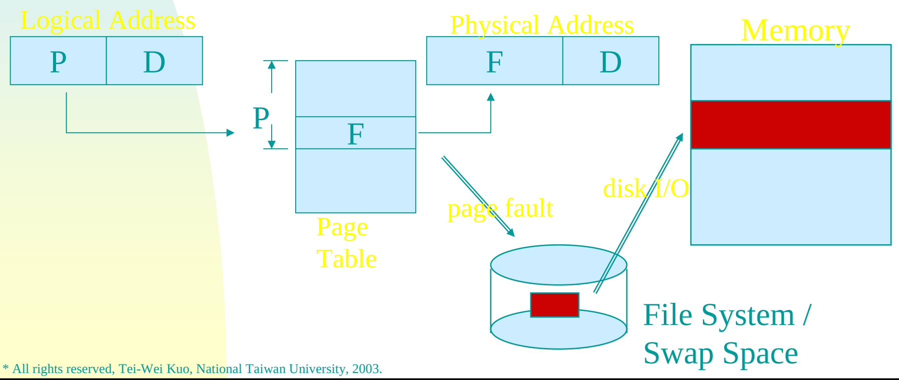
Memory Management

- Virtual Memory – Demand paging



Memory Management

- Demand Paging
  - Page fault -> disk I/O -> modify the page table -> rerun the instruction!



Process Control

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```

- None of them has an error return.

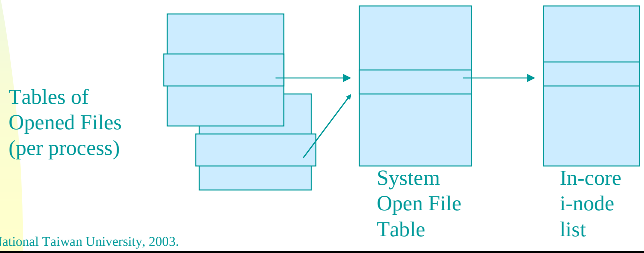
fork

#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);

- The only way beside the bootstrap process to create a new process.
- Call once but return twice
  - 0 for the child process (getppid)
  - Child pid for the parent (1:n)
- Copies of almost everything but no sharing of memory, except text
  - Copy-on-write (fork() – exec())

fork

- Program 8.1 – Page 189
  - fork(), race conditions, write vs standard I/O functions
- File sharing
  - Sharing of file offesets (including stdin, stdout, stderr)



fork

- Normal cases in fork:
  - The parent waits for the child to complete.
  - The parent and child each go their own way (e.g., network servers).
- Inherited properties:
  - Real/effective [ug]id, supplementary gid, process group ID, session ID, controlling terminal, set[ug]id flag, current working dir, root dir, file-mode creation mask, signal mask & dispositions, FD_CLOEXEC flags, environment, attached shared memory segments, resource limits
- Differences on properties:
  - Returned value from fork, process ID, parent pid, tms_[us]time, tms_c[us]time, file locks, pending alarms, pending signals

fork

- Reasons for fork to fail
  - Too many processes in the system
  - The total number of processes for the real uid exceeds the limit
    - CHILD_MAX
- Usages of fork
  - Duplicate a process to run different sections of code
    - Network servers
  - Want to run a different program
    - shells (spawn = fork+exec)

vfork

- Design Objective
  - An optimization on performance
    - Execute exec right after returns from fork.
- Mechanism – SVR4 & 4.3+BSD
  - Since 4BSD
    - <vfork.h> in some systems
  - No copying of the parent's address space into the child.
    - Sharing of address space

vfork

- vfork() is as the same as fork() except
  - The child runs in the address space of its parent.
  - The parent waits until the child calls exit or exec.
    - A possibility of deadlock
- Program 8.2 – Page 194
  - vfork, _exit vs exit (flushing/closing of stdout)

exit

- Five ways to terminate:
  - Normal termination
    - Return from main().
    - Call exit() – ANSI C
      - Incomplete in Unix – filedes, multiple processes & job control
    - Call _exit() – POSIX.1
  - Abnormal termination
    - Call abort()
      - Generate SIGABRT
    - Be terminated by a signal.

exit

- Termination
  - The same code in the kernel is finally executed.
    - Close all open descriptors, release memory, and the like.
  - Exit status vs termination status
    - Exit status (arg from exit, _exit, or return) → termination status
      - In abnormal case, the kernel generate it.
    - wait & waitpid

exit

- zombie
  - The process which has terminated, but its parent has not yet waited for it.
- Order of terminations
  - The parent before the child
    - Inherited by init
      - When a parent terminates, it is done by the kernel.
      - Clean up of the zombies by the init – wait whenever needed!
  - Otherwise
    - Keep some minimum info for the parent

wait & waitpid

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int op);
```

  - wait will block until one child terminates or an error could be returned.
    - waitpid could wait for a specific one and has an option not to be blocked. + job ctrl
- SIGCHILD from the kernel if a child terminates
  - Default action is ignoring.

wait & waitpid

- Three situations in calling wait/waitpid
  - Block
  - Return with the termination status of a child
  - Return with an error.
- Termination Status <sys/wait.h> – Figure 8.2
  - Exit status (WIFEXITED, WEXITSTATUS)
  - Signal # (WIFSIGNALED, WTERMSIG)
  - Core dump (WCOREDUMP)
  - Others (WIFSTOPPED, WSTOPSIG)

wait & waitpid

- Program 8.3 – Page 199
  - pr_exit, mapping of signal numbers <signal.h>

pid_t waitpid(pid_t pid, int *statloc, int op);

  - pid
    - pid == -1 → wait for any child
    - pid > 0 → wait for the child with pid
    - pid == 0 → wait for any child with the same group id
    - pid < -1 → wait for any child with the group ID = |pid|
  - pid of the child or an error is returned.

wait & waitpid

- Errors
  - No such child or wrong parent
- Option for waitpid
  - WNOHANG, WUNTRACED
  - WNOWAIT, WCONTINUED (SVR4)

- Program 8.4 – Page 200
  - Different exit status
- Program 8.5 – Page 202
  - Forking twice – inheritance by init

wait3 & wait4

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int op, struct rusage
    *rusage);
pid_t wait4(pid_t pid, int *statloc, int op,
    struct rusage *rusage);
```

- 4.3+BSD – Figure 8.4, Page 203
- User/system CPU time, # of page faults, # of signals received, the like.

Race Conditions

- Def: When multiple processes are trying to do something with shared data, the final outcome depends on the order in which the processes run.
- Example: Program 8.5 – Page 202
  - Who is the parent of the 2nd child?
- Program 8.6 – Page 205
  - Mixture of output by putc + setting of unbuffering for stdout

Race Conditions

- How to synchronize?
  - Waiting loops?
  - Inter-Process Communication facility, such as pipe (Program 14.3), fifo, semaphore, shared memory, etc.
- Program 8.7 – Page 206
  - WAIT_PARENT(), TELL_CHILD(), WAIT_CHILD(), TELL_PARENT()

```
while (getppid() != 1)
        sleep(1);
```

```
Child:
…
WAIT_PARENT (getppid());

Parent:
…
TELL_CHILD(pid);
```

exec

- Replace the text, data, heap, and stack segments of a process with a program!

```
#include <unistd.h>
int execl(const char *pathname, const char
    *arg0, … /* (char *) 0 */);
int execv(const char *pathname, char *const
    argv[]);
int execle(const char *pathname, const char
    *arg0, … /* (char *) 0, char *const envp[] */);
int execve(const char *pathname, char *const
    argv[], char *const envp[]);
```

  - *l*, *v*, and *e* stands for list, vector, and environment, respectively.

exec

```
#include <unistd.h>
int execlp(const char *filename, const char
    *arg0, … /* (char *) 0 */);
int execvp(const char *filename, , char *const
    argv[]);
```

  - With *p*, a filename is specified unless it contains ‘/’.
    - PATH=/bin:/usr/bin:.
    - /bin/sh is invoked with “filename” if the file is not a machine executable.
    - Example usage: login, ARG_MAX (4096)
  - Figure 8.5 – Page 209 (6 exec functions)

exec

- Inherited from the calling process:
  - pid, ppid, real [ug]id, supplementary gid, proc gid, session id, controlling terminal, time left until alarm clock, current working dir, root dir, file mode creation mask, file locks, proc signal mask, pending signals, resource limits, tms_[us]time, tms_cutime, tms_ustime
  - FD_CLOEXEC flag
- Requirements & Changes
  - Closing of open dir streams, effective user/group ID, etc.

exec

$$\begin{tikzcd}
\boxed{\text{execlp}} \arrow[d, "\text{build } argv"] & \boxed{\text{execl}} \arrow[d, "\text{build } argv"] & \boxed{\text{execle}} \arrow[d, "\text{build } argv"] \\
\boxed{\text{execvp}} \arrow[r, "\text{try each}", "\text{PATH prefix}"'] & \boxed{\text{execv}} \arrow[r, "\text{use}", "\text{environ}"'] & \boxed{\text{execve}}
\end{tikzcd}$$

- In many Unix implementations, execve is a system call.
- Program 8.8 – Page 211
  - Program 8.9 – Page 212
  - The prompt bet the printing of argv[0] and argv[1].

User/Group ID’s

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
```

- The process == superuser → set real/effective/saved-suid = *uid*
- Otherwise, euid=*uid* if *uid* == ruid or *uid* == saved-suid
- Or errno=EPERM (_POSIX_SAVED_IDS)

```
int setgid(gid_t gid);
```

- The same as setuid

User/Group ID’s

- Remark – Figure 8.7, Page 214
  - Only superuser process can change the real uid – normally done by the login program.
  - The euid is set by exec only if the setuid bit is set for the program file. euid can only be set as its saved-suid or ruid.
  - exec copies the euid to the saved-suid (after the setting of euid if setuid bit is on).

User/Group ID’s

- Example *tip* (BSD) or *cu* (SV) – Page 214
  - The setuid bit is on for *tip* (owner=uucp).
    - For file locking
  - *Tip* calls setuid(ruid) for file access & later creating of shells/processes
    - Correct uid
  - Switch the euid back to uucp
  - Remark: Not good for files with setuid = root!

User/Group ID’s

```
#include <sys/types.h>
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(uid_t rgid, uid_t egid);
```

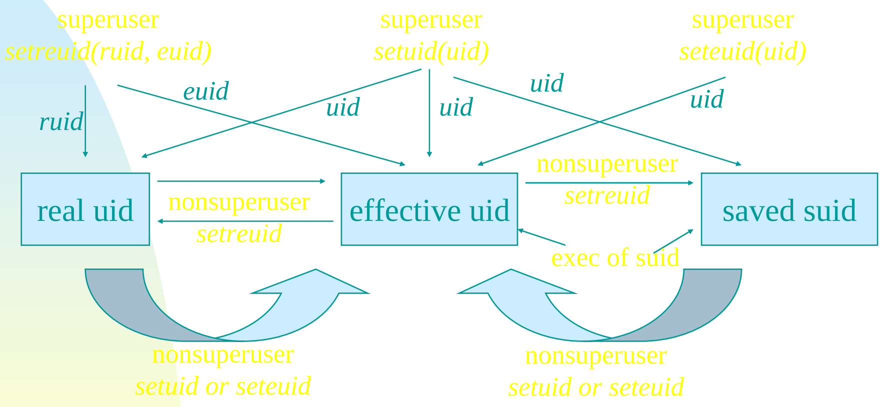
- Swapping of real and effective uids.
  - Good for even unprivileged users.
- BSD only or BSD-compatibility library

User/Group ID's

```
#include <sys/types.h>
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(uid_t gid);
```

- Non-superusers can only set euid=ruid or saved-setuid.
- A privileged user only sets euid = uid.
  - It is different from setuid(uid)

User/Group ID's



- The supplementary guid's are not affected by the setgid function.

Interpreter Files

- Def: a file begins with a line of the form: #! pathname [optional-argument]
  - E.g., “#! /bin/sh”
- Implementation:
  - Recognition is done within the kernel
  - Interpreter (normally an absolute pathname) vs the interpreter files
  - Line-limit of the first line, e.g., 32 chars.
- Program 8.10 – Page 218
  - Argument list, arg pathname of execl()

Interpreter Files

- Program 8.11 – Page 219
  - “awk –f myfile” lets awk to read an awk program from “myfile”.
  - Argument list: awkexample file1 FILE2 f3 (at /usr/local/bin/)

```
#! /bin/awk –f
BEGIN {
  for (i = 0; i < ARGC; i++)
  printf “ARGV[%d] = %s\n”, i, ARGV[i]
  exit
}
```

- /bin/awk –f /usr/local/bin/awkexample file1 FILE2 f3
  - Page 219

Interpreter Files

- Example: Removing of “-f”

```
$su
Passwd:
# mv /bin/awk /bin/awk.save
# cp /home/stevens/bin/echoarg /bin/awk
# suspend
[1] + Stopped  su
$arkexample file1 FILE2 f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILE2
argv[5]: f3
```

Interpreter Files

- Why interpreter files?
  - Pro:
    - Hid the fact that certain programs are scripts in other languages.
    - Easy to use (wrapping programs).
      - execlp → /bin/sh → fork, exec,wait
    - Choices of shells
  - Against:
    - Efficiency for users but at the cost of the kernel
      - Executable files? /bin/sh? /bin/awk?

system

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

- If cmdstring = null, return nonzero only if a command interpreter is available.
- Objective:
  - Convenient in usage
    - system("date > file");
    - Or write a program: call time, localtime, strftime, write, etc.
  - ANSI C definition → system-dependent
    - An interface to shells

system

- Implementation: fork-exec-waitpid
  - Program 8.12 – Page 223
    - The implementation of system()
    - The shell's –c tells to take next cmd-line argument as its command input – parameter passing, meta chars, path, etc.
    - Call _exit(127)
  - Returns –1 if fork fails or waitpid returns error other than EINTR (a caught signal).
  - Returns 127 if exec fails.
  - Return the termination status of the shell (in the format for waitpid)

system

- Program 8.13 – Page 224
  - Calling system()
- Advantage
  - system() does all the error handling/ signal handling
- A security hole
  - Call system from a setuid program
    - Programs 8.14 & 8.15 – Page 225
    - A set[ug]id program should change its uid's back to the normal after call fork.

Process Accounting

- Non-POSIX standards
  - SVR4 & 4.3+BSD supported
    - accton [filename]
      - /var/adm/pacct or /usr/adm/acct

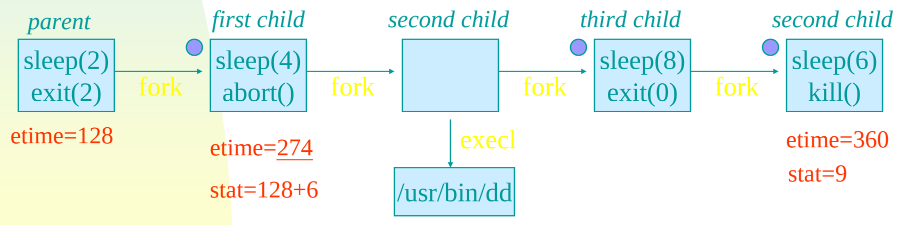
```
typedef u_short comp_t;
struct acct {
    char   ac_flag; /* Figure 8.9 – Page 227 */
    char   ac_stat; /* termination status (core flag + signal #) */
    uid_t  ac_uid; gid_t   ac_gid;  /* real [ug]id */
    dev_t ac_tty;  /* controlling terminal */
    time_t ac_btime; /* staring calendar time (seconds) */
    comp_t ac_utime; /* user CPU time (ticks) */
    comp_t ac_stime; /* system CPU time (ticks) */
    comp_t ac_etime; /* elapsed time (ticks) */
    comp_t ac_mem; /* average memory usage */
    comp_t ac_io; /* bytes transferred (by r/w) */
    comp_t ac_rw; /* blocks read or written */
    char      ac_comm[8]; /* command name: [8] for SVR4, [10] for 4.3 BSD */
 };
```

Process Accounting

- Accounting Information
  - Kept in the process table whenever a new process is created.
  - Each accounting record is written into the accounting file in the order of the termination order of processes.

Process Accounting

- A new record for each process
  - E.g., A execs B, then B execs C
    - ac_flag: AFORK is cleared. (cmd=C)
- Programs 8.16 & 8.17 – Page 228-230



Remark: 60 ticks/sec ● flag=AFORK

User Identification

#include <unistd.h>

char *getlogin(void);

- Fail if the process is not attached to a terminal – daemons
- A user could have a multiple login names – login's user name
  - getpwuid, getpwnam
    - Function ttyname – utmp
    - Environment var LOGNAME (set by the login process in 4.3+BSD) – user-space data

Process Times

#include <sys/times.h>

clock_t times(struct tms *buf);

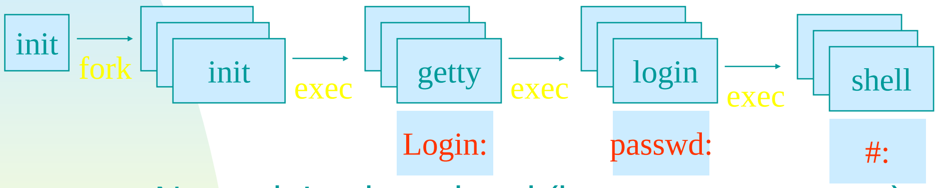
- The Returned value from some arbitrary point in the past.

```
struct tms {
  clock_t tms_utime; /* user CPU time */
  clock_t tms_stime; /* system CPU time */
  clock_t tms_cutime; /* user CPU time, terminated child */
  clock_t tms_cstime; /* system CPU time terminated child */
```

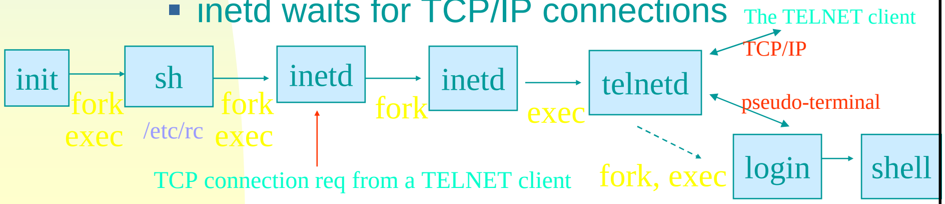
- Program 8.18 – Page 234
  - times
  - _SC_CLK_TCK

Remark: Logins – Chapter 9

- Terminal Logins – one for each terminal device

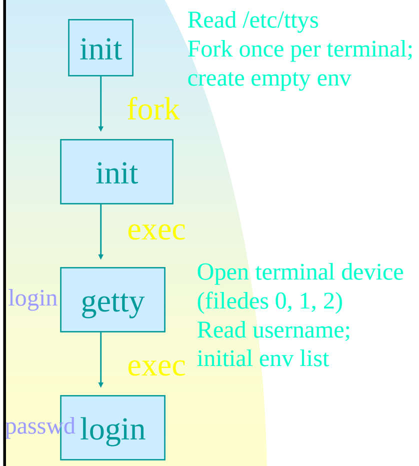


- Network Logins – inetd (internet superserver)
  - inetd waits for TCP/IP connections



* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

4.3+BSD Terminal Logins



- execle("/usr/bin/login", "login", "-p", usrname, (char *) 0, envp)
  - ruid = euid = 0
  - ppid = 1
  - TERM=foo
    - gettytab
  - Getpwnam; getpass; crypt
    - pw_passwd
  - Fail → exit; init forks; exec(getty)
  - Succeed → home dir (chdir); terminal device (chown); terminal access rights; setgid; initgroups; envp (HOME, PATH, etc), setuid, execl("/usr/bin", "-sh", (char *)0)

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

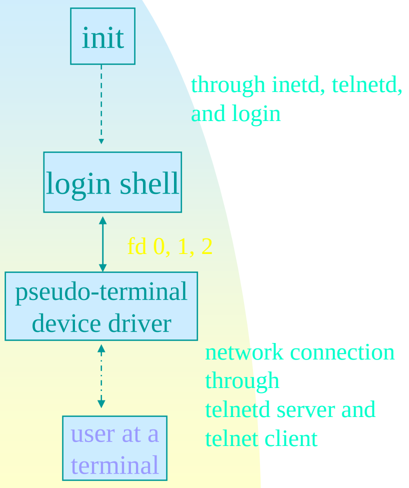
4.3+BSD Terminal Logins

- Login shell
  - Read the start-up files (.profile for Bourne shell and KornShell, .cshrc and .login for C shell)
- When a login shell terminates, init is notified, and the whole procedure restarts!
- SVR4: (1) getty (2) ttymon
  - Init → sac → ttymon → login → login shell

fork; exec fork; exec exec

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

Network Logins – 4.3+BSD

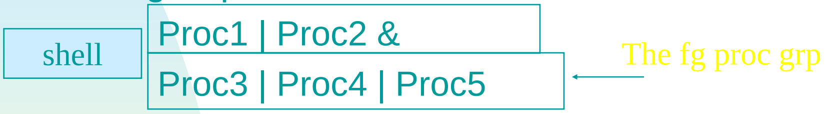

init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

- Regardless of terminal or network logins, the file descriptors 0, 1, 2 of a login shell is connected to a terminal device or a pseudo-terminal device.
- Login does more things
  - Checking of new mail, etc.
- SVR4
  - Parent(inetd) = service access controller (sac)

* All rights reserved, Tei-Wei Kuo, National Taiwan University, 2003.

Remark: Session, Controlling Terminal, Job Control – Ch 9

- Session – a collection of one or more process groups



- Controlling Terminal – (pseudo) terminal device we log in!
  - Foreground/background process grp (^C → who?)
- Job Control
  - Start multiple jobs (grp of processes) from a terminal and control which jobs can access the terminal and which jobs run in the background.

Process Groups

- Process Group
  - A collection of one or more processes
  - Unique process group ID

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void);
pid_t setpgid(pid_t pid, pid_t pgid);
```

    - PID (leader) = process group ID
    - pid = 0 → pid of the caller, gpid=0 → gpid=pid
    - Can only set the pgrp of itself and its children (without exec)
    - proc | proc2

Sessions

- Session
  - A collection of one or more process groups

```
#include <sys/types.h>
#include <unistd.h>
pid_t setsid(void);
```

  - A new session is created if the calling process is not a process group leader → the session leader, the process group leader
    - No controlling terminal
  - Otherwise; an error is returned!

Controlling Terminal

- Controlling Terminal – (pseudo) terminal device we log in!
- Controlling process – the session leader that establishes the connection to the controlling terminal
  - With a controlling terminal → one foreground pgrp & N background pgrps
  - open(/dev/tty)?
- Foreground/background process grp: terminal inputs, signal (e.g., ^C, ^Z) → who?

Job Control

- Job Control – control which jobs (groups of processes) can access the terminal and which jobs are to run in the background!
  - Shell, terminal driver, signals
  - _POSIX_JOB_CONTROL
  - SVR4, 4.3+BSD, POSIX.1
- ^Z → SIGSTP, ^C → SIGINT, ^\ → SIGQUIT ➔ foreground pgrp!
  - pr * | lpr &   (P250)
  - fg

Job Control

- SIGTTIN – sent by the terminal driver for a background job that try to reads from the terminal.

```
#include <sys/types.h>
#include <unistd.h>
pid_t tcgetpgrp(int filedes);
pid_t tcsetpgrp(int filedes, pid_t pgrpid);
```

- Shell could call tcsetpgrp to set the foreground process group ID to *pgrpid*, where filedes refers to the controlling terminal.