Topic 4

Unix Process Programming

Objectives

- Understand the concept of process;
- Understand what constitutes the memory image of a process;
- Understand various attributes of a process;
- Understand the parent/child relationship between processes;
- Be able to create multiple processes;
- Be able to execute a new program in a process;
- Be able to synchronise multiple processes with `wait`, `waitpid` and `exit` system calls;
- Understand how a process is terminated and claimed by its parent process;
- Be able to obtain the termination status of a child process;
- Understand and be able to obtain and change a process's environment;
- Understand the concept of process group, session and control terminal, and be able to use the relevant system calls;
- Understand and be able to create daemon processes.

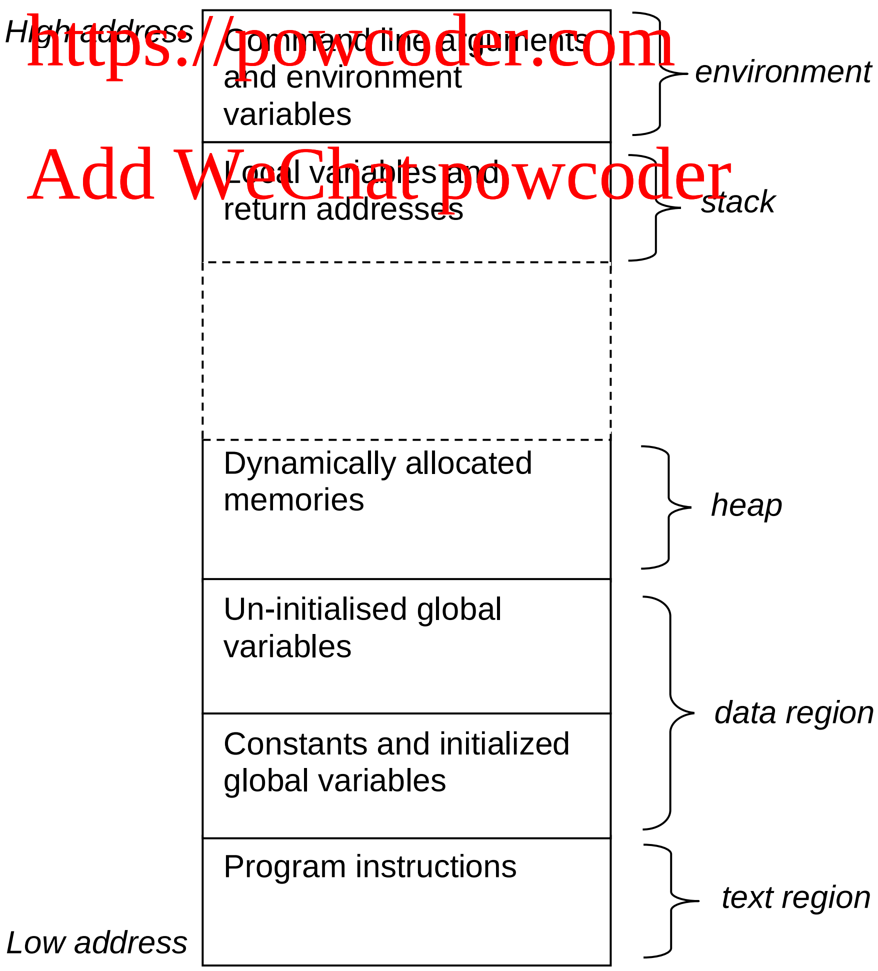
Readings

- Stevens & Rago: Ch 7.1-7.9, Ch 8, and Ch 13
- Manual pages for `fork`, `execve`, `waitpid`, `exit` etc

1. Process

(a) A process is the execution of a program. In order to execute a program, the operating system must create a process to house a data structure in the memory. In addition to the PCB (Process Control Block), this data structure also includes the memory space for keeping the program instructions (text), the global data, the heap, the stack, and the process environment. We will refer to this data structure as the *memory image* of the process.

The following diagram depicts part of a process' memory image that is kept in the user space, which is accessible by the programmer.



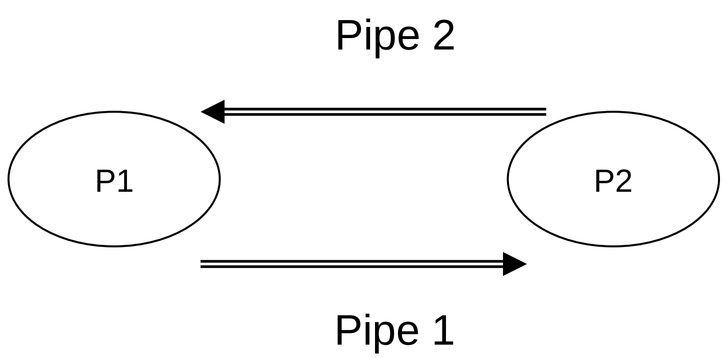
(b) Many processes may execute concurrently. Several processes may come from the same program. On a single processor system, all processes time-share the same CPU.

(c) In addition to its program instructions, a process is also defined by such things as:

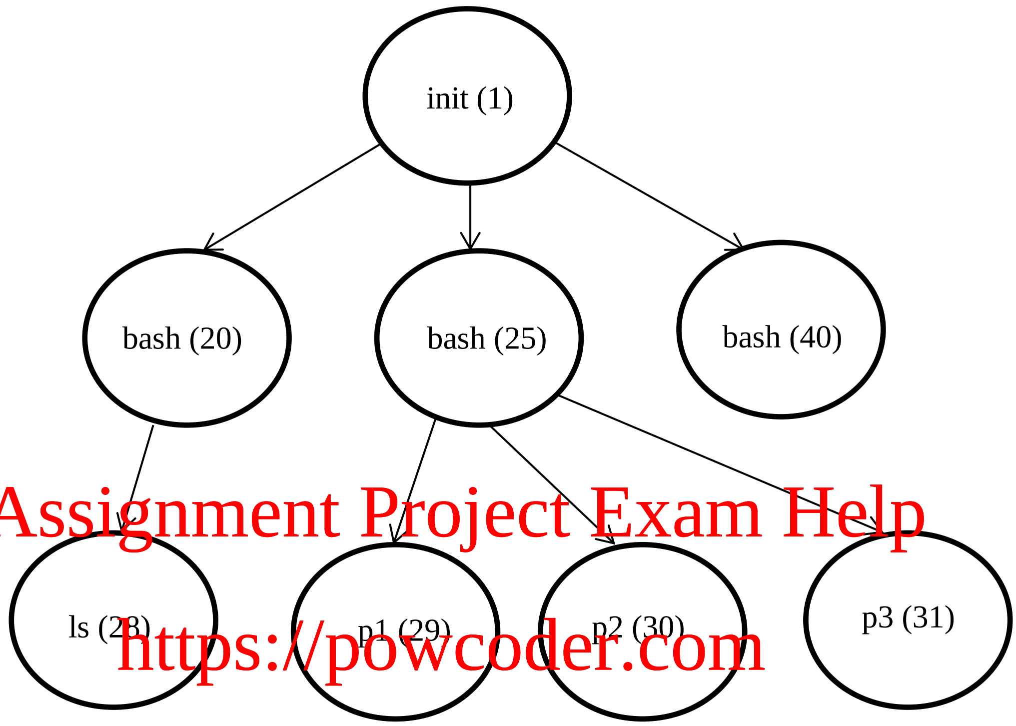
- values of the global variables
- contents of the heap
- contents of its stack and environment
- values of the CPU registers
- the process state in the PCB
- I/O status (also kept in PCB)
- open files (in separate data structures)

Unlike the program instructions, the above data keep changing during the lifetime of the process.

(d) A process has its own address space – it does not share its address space with another process. However, two processes may communicate with each other through a variety of IPCs (Interprocess Communication Mechanisms), e.g., pipes, FIFOS, files, sockets, shared memories, and message queues.



(e) A process (the parent process) can create a new process (its child process). Each process is identified by a unique number known as the process ID (PID).



(f) Four basic system calls for process control:

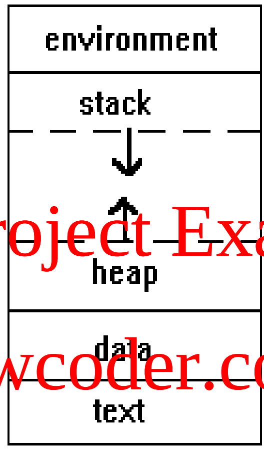
1) `fork`: creates a child process by duplicating itself
2) `exec`: loads a program into the process memory space (to replace the existing program) and start the execution of the loaded program
3) `wait`: `waitpid`: waits for a child process to terminate and obtains the termination status of the child process
4) `exit`: terminates the process with an exit status.

2. The `fork` call

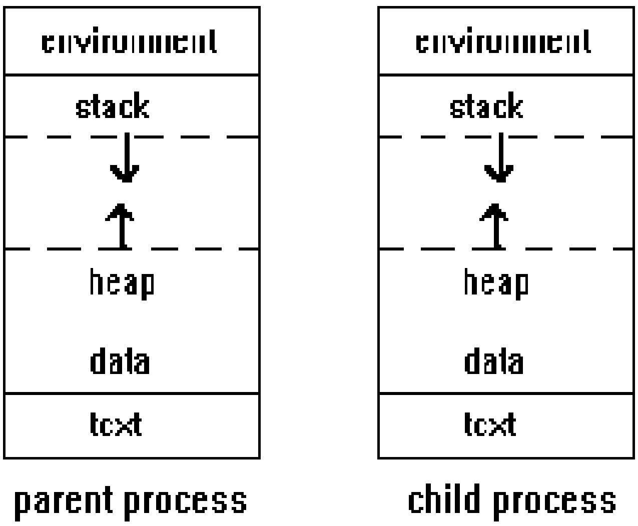
```
#include <unistd.h>

pid_t  pid;
pid = fork();
```

(a) `fork` creates a new process by duplicating itself.

before fork:



after fork:



Immediately after the `fork` call, the contents of the text, data, heap, stack and environment in the new process are exactly same as those of the parent process, except the return value from the `fork` call.

(b) `pid`

- = -1 there is an error
- = 0 the process is the child process
- > 0 the process is the parent process. The return value is the PID of the child process

(c) After the `fork` call, both the parent and the child processes execute *concurrently*. In a single processor system, they time-share the same processor. However, the execution order of the two processes depends on the process scheduler in the Unix kernel. To the end users, it is unpredictable!


```
printf ("fork ...\n");
pid = fork();
printf("pid = %d\n", pid);
```

Process A (pid = 200)

before forking

after forking

```
printf ("fork ...\n");
pid = fork();
printf("pid = %d\n", pid);
```

Process A (pid = 200)

```
printf ("fork ...\n");
pid = fork();
printf("pid = %d\n", pid);
```

Process B (pid = 300)

```
fork ...
pid = 300
pid = 0
```

or

```
fork ...
pid = 0
pid = 300
```

Example: by examining the return value from the `fork` call, you can determine whether the current process is the original parent process or the newly created child process.

```
/* spawn.c  --- demonstrate fork  */

#include <stdio.h>
#include <unistd.h>

int main()
{
   pid_t  pid;

   printf("Just one process so far, call fork…\n");
   pid = fork();

   if (pid==0)
        printf("I am the child process\n");
   else if (pid > 0)
        printf("I am parent process, child PID =
%d\n", pid);
   else
      printf("fork returned error code. No child\n");

   exit(0);
}
```

3. The exec family of function calls

The `exec` is a family of function calls used by a process to load and execute a new program in the *existing* process. When a process calls one of these `exec` functions, the process' existing text (i.e., the program instructions) and data (the global data, the stack and the heap) are completely replaced by those of the new program. The execution of the new program starts at the `main` function of the new program.

Note, however, most other properties of the process will remain unchanged (e.g., PID, and sometimes, the process environment), because no new process is created.


```
#include <unistd.h>

int execl(char *pathname, char *arg0, …, (char *)0);
int execv(char *pathname, char *argv[]);
int execlp(char *filename, char *arg0, …, (char *)0);
int execvp(char *filename, char *arg[];
int execle(char *pathname, char *arg0, …, (char*)0,
          char *envp[]);
int execve(char *pathname, char *arg[], char *envp[]);
```


These six function calls differ in:

(a) **where to locate the program file**

- `execl` and `execv` use the given pathname to locate the program file
- `execlp` and `execvp` search the directories listed in the current `PATH` environment variable to locate the program file.

- `execle` and `execve` use the `PATH` environment variable given in the parameter `envp` to locate the program file.

```
/* Example: execute program:  ls -lt  */

#include <unistd.h>

int main()
{
   execl("/bin/ls",  "ls",  "-lt",  (char *)0);
}
```

in the above example, `execl (…)` can be replaced by the following:

```
execlp ("ls",  "ls",  "-lt", (char *)0);
```


(b) **How are the parameters provided?**

- with `execl`, `execlp` and `execle`, the parameters are provided as a list of individual strings
- with `execv`, `execvp` and `execve`, the parameters are provided as an array of strings

Example:

```
char * argv[] = {"ls", "-lt", (char *)0};

execv ("/bin/ls", argv);
execvp("ls", argv);
```

(c) **Whether to retain the old environment in the process?**

A process inherits its environment from its parent process. None of the `exec` calls, except `execle` and `execve`, changes the existing environment of the current process.

One can replace the existing environment of the process with a new environment using either `execle` or `execve` call. To do this, you need to construct an array and store the new environment variables and their values in that array. The array is passed as the last argument to the `execle` or `execve` function.

The environment of a process is a sequence of strings such as:

```
PATH=/usr/bin:/bin:.
HOME=/home/staff/hong
SHELL=/bin/csh
TERM=xterm
EDITOR=vi
USER=hong
```


The environment is usually set up in the shell initialization files such as `.login` and `.cshrc` (for csh shell) or `.bash_profile`, `.bashrc` and `.profile` (for bash shell) in a user's home directory.

The behaviours of many Unix programs depend on some of these environment variables. One of the common methods to configure an application program is by setting up certain pre-defined environment variables in the shell.

When the shell creates a new process, these environment variables are passed to the new process from the shell process (which is the parent of the new process).

- `fork`: copy the environment of the parent (ie, the shell) to the child process;
- `execl`, `execv`, `execlp`, `execvp`: does not change the environment of the current process;

- `execle`, `execve`: replaces the environment of the current process with the environment stored in parameter `envp` of the `exec` call**.**

A successful `exec` call will load the new program into the calling process's memory space and completely replace the original program (as well as global data, the stack and the heap). Hence the `exec` function call will *not* return unless the call fails. When an `exec` call fails, it returns –1.

```
int main()
{
    printf(“executing  ls -l\n”);
    execl(“/bin/ls”, “ls”, “-lt”, (char *)0);

    /* if the following statement is executed,
    then the execl call must have failed  */

    printf(“execl failed to run ls -l\n”);
    exit(1);
}
```

4. The `exit` function

```
#include <stdlib.h>

void exit(int exitstatus);
```

This function is defined in ANSI C. On a Unix system, it calls the system call `_exit`, which is defined in POSIX.1, to:

(1) perform clean-up operations (such as closing all open file descriptors, and release memory used by the calling process).

(2) terminate the calling process and notify its parent process with an exit status.

Note: the difference between `return` and `exit`:

(1) in the `main` function, both can be used to terminate current process.

(2) in other functions, `return` will cause control flow to return to the calling function. While `exit` will terminate calling process.

5. `wait` and `waitpid`

The system calls `wait` and `waitpid` are used by the calling process to synchronise with its child processes

`wait`:

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

pid_t wait(int *statloc);
     Return: child process id if OK
        or -1 on error
```

Note:

(1) the calling process blocks until one of its child processes terminates. When the child process terminates, `wait` returns its *pid* and passes back the termination status of the child process.

termination status	termination signal	exit status

(2) The termination status is stored in address given by the parameter `statloc`. It is a 32-bit integer divided into two parts. The higher 16 bits contain the termination signal number if the child process was killed by a signal. Otherwise, the lower 16 bits contain the exit status if the child process was terminated by calling the `exit` function.

If we are not interested in the termination status of the child process, we may pass NULL pointer

```
((int *)0)
```

to the wait function.

(3) If the calling process has no child process or if each one of its child processes has already been claimed by a previous `wait` call, the current `wait` call will return immediately with return value `-1`.

(4) If the calling process has a child process that had already exited but not yet claimed by the wait call (such process is called a “zombie” process), the current `wait` call will return immediately with the pid of the zombie process and its termination status.

(5) If the parent process is terminated before its child process, the child process will be “adopted” by the `init` process.

When the child process terminates eventually, it will be claimed by the `init` process.

`waitpid`


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

pid_t waitpid(pid_t pid, int *statloc, int option);
     Return: child process id if OK
        or -1 on error
```

With the `wait` call, it is not possible to wait for a specific child process. This was one of the reasons why `waitpid` call was added later. With the `waitpid` call, one can wait for a specific child process. Another difference between `wait` and `waitpid` is that `wait` will always block if no child process is dead and at least one child process is still running. With `waitpid`, one may choose non-blocking wait by using the appropriate option:

`pid==-1`: wait for any child process

`pid>0`: wait for the child whose process id is pid

`pid==0`: wait for any child whose process group id equals to the group id of the calling process

`pid<-1`: wait for any child whose process group id equals to `|pid|`.

`options==0` blocking
`options==WNOHANG` non blocking

Macros for Obtaining Termination Status

Since the termination status from the `wait` or `waitpid` function consists of two parts – the termination signal (if it is killed by a signal – we will discuss signals in Topic 6) or the exit status (if the process terminates voluntarily by calling the `exit` function), it is necessary to break the two parts in order to obtain the exit status or the termination signal. Unix systems have defined a number C Macros for this purpose. For example, to see if the process was terminated by calling the `exit` function, we use


```
WIFEXITED(termination_status)
```

If the above macro returns true, the process was terminated voluntarily. Similarly to see if the process was terminated by a signal, use

```
WIFSIGNALED(termination_status)
```

If the process terminates voluntarily, we can find out the exit status of the process:

`WEXITSTATUS(`*`termination_status`*`)`

If the process was killed by a signal, we can find out the signal number used to kill the process:

`WTERMSIG(`*`termination_status`*`)`

For full details of these macros, you should consult the manual pages for `wait` and `waitpid` functions. For example, use command:

```
man -S 2 wait
```

6. Process Synchronization

A parent process and its child process can synchronise with each other with the `wait` (and `waitpid`) and `exit` calls, as shown in the following example:

```
/* runls -- run “ls” in a child process */

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

int main()
{
   pid_t pid = fork();

   /* if parent, wait until child finishes */
   if (pid > 0)
   {
      wait ( (int *)0);
      printf("ls completed\n");
      exit(0);
   }

   /* if child, exec ls */
   if (pid == 0)
   {
      execl ("/bin/ls",  "ls",  "-l",  (char *)0);
      perror("execl failed");
      exit(1);
   }

   /*  pid  <  0,  fork failed  */
    perror("fork failed");
   exit(1);
}
```

7. Process Environment

(a) Environment Variables

on the command line, command env will display a list of environment variables (and their values)

```
% env

PATH = /bin:/usr/bin: /usr/ucb: /lsdsn/bin:
USER = hong
SHELL = /bin/csh
TERM = vt100
```

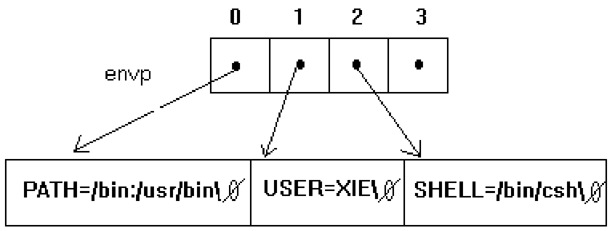
An environment variable can be created, set or changed by `setenv` command (under csh shell), or using `export` command (under bash shell).

The environment of a process is inherited from its parent process during `fork` and `exec` (`execl`, `execv`, `execlp`, `execvp`) calls.

(b) Accessing Environment

Method 1:

```
int main (int argc, char *argv[], char *envp[])
{
   int i = 0;
   while (envp[i] != NULL)
   {
      printf("%s\n", envp[i] );
      i++;
   }
}
```


0
1
2
3
envp
PATH=/bin:/usr/bin\0
USER=XIE\0
SHELL=/bin/csh\0

Method 2: (POSIX.1 compliant)

```
extern char **environ;

int main()
{
   int i=0;

   while (environ[i] != NULL )
   {
      printf("%s\n", environ[i]);
      i++;
   }
}
```

(c)   Obtain an environment variable


```
char *name, *value;
value = getenv(name);
```


Example:


```
char *s;

if ((s=getenv("USER")) != NULL )
      printf("The user is %s\n", s);
```

8. Process Groups and Sessions

Apart from parent-child relationship, processes are also organized into sessions and process groups. Each process belongs to one process group, and each process group belongs to one session.

A session is a collection of process groups, usually associated with the same control terminal. The process groups that are created under a terminal login session (either terminal login using RS232 or network terminal login using TELNET or SSH protocol) are placed in the same session, unless some of those processes choose to create new sessions.

A process group is a collection of processes. Usually a process inherits its process group from its parent process (hence also its session), unless it chooses to create a new process group, or it is placed in a different process group by its parent process. On shells that support job control, all processes in the same job are placed in the same process group.

Example:

```
% ls -lt  | grep john | more
```

processes ls, grep and more are placed in the same process group under bash shell (or other shells that support job control).

For example, assuming that there are currently two login sessions on a computer, one from the local terminal (ttya) and the other from remote computer via ssh (ttyb).

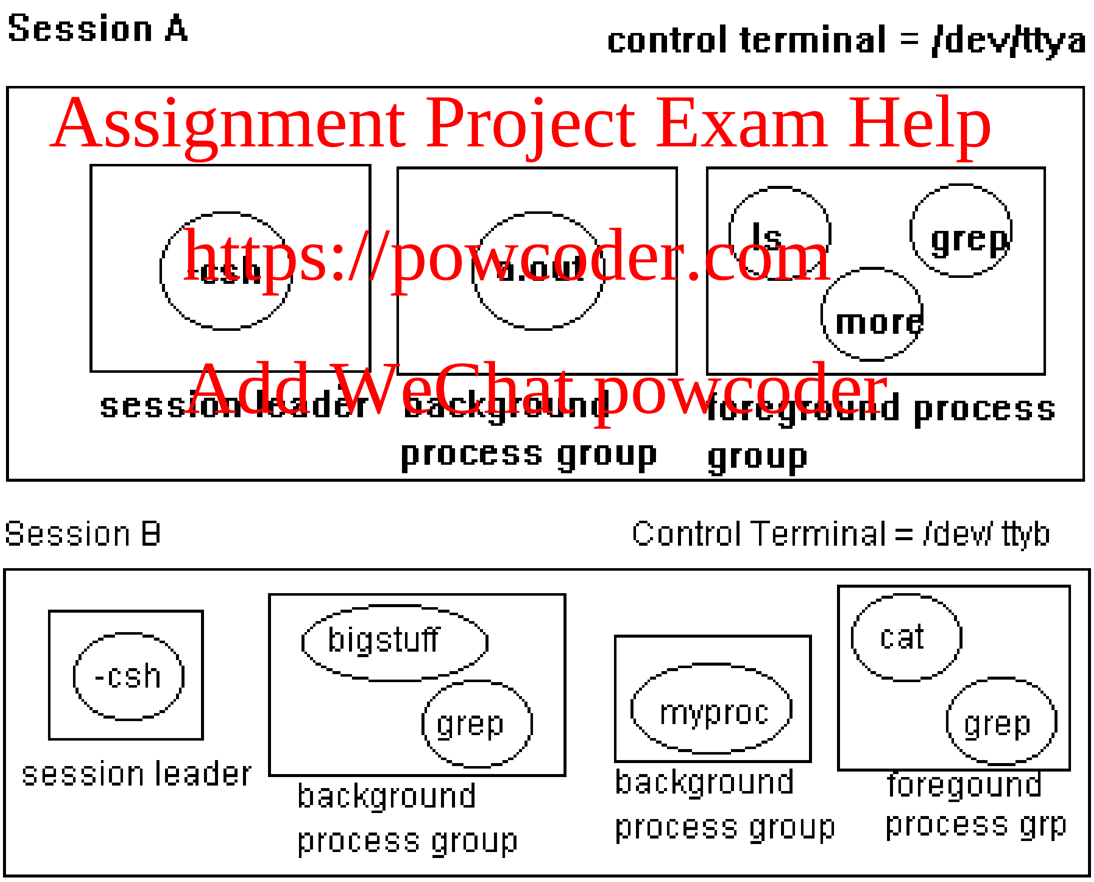
On terminal `ttya`

```
ttya% a.out  &
ttya% ls -lt | grep john | more
```

On terminal `ttyb`:

```
ttyb% bigstuff | grep xyz > foo &
ttyb% myproc &
ttyb% cat readme | grep apple
```

We would see two sessions:



9. Process Group

Each process group is identified by a unique process group ID. A process group may have a group leader. The leader's process group ID equals to its own process ID.

A process can join an existing process group or create a new process group by calling `setpgid`:

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

int setpgid(pid_t pid, pid_t pgid);
           Returns  0 if OK
              or  -1 on error
```


This call places the process with process ID = `pid` into the process group with group ID = `pgid`. Here a process can only put itself (with `pid=0`) or one of its children (with `pid>0`) into another process group. If `pgid=0`, then `pid` is used as the process group ID.

Example: the function call

```
setpgid(0,0);
```

would create a new process group whose group ID equals to the pid of the calling process. The calling process becomes the group leader of the new process group.

10. Daemon Process

In the Unix system, a daemon process is one that usually does not have a controlling terminal and continues to run unless it is explicitly terminated. A daemon process is often used to run a server program such as web server and SSH server.

To turn a normal process into a daemon, the following issues need to be considered:

(a) **Not to be terminated at logout.**
The process is not terminated when the session in which the process was created is closed. Assume that you logged in and then started a process in the background. When you logged out the parent process, usually the shell, would terminate all its child processes, including that background process, automatically. Therefore, we need to make sure that when you log out, the daemon process will not be terminated automatically by its parent.

(b) **Not to have a controlling terminal.**
A normal process has a control terminal. Its behaviour can be affected by the terminal activities such as certain keystrokes. For example, typing `CTRL-C` or `CTRL-\` will usually terminate all foreground processes. In order to prevent keyboard activities from interfering with the daemon process, the daemon should be moved out of the session in which it was created and moved to a session that has no controlling terminal. In this way, the keyboard activities will have no effects on the daemon processes.

(c) **Setting up appropriate current directory.**
When a process is created it remembers the current directory of the shell (or parent process) immediately before it is created. This current directory may be any directory and therefore may not make much sense to the daemon process. Hence we should change the current directory of the daemon to a directory that is more appropriate for the service it provides. For example, if the daemon is a web server, it is more appropriate to set its current directory to the document root directory rather than a random directory from which the server program was invoked.

(d) **Allow file systems to be un-mounted.**
Once the Unix is booted, it mounts several file systems (or disk partitions). Each Unix file system forms a tree of file and directory hierarchy. All file systems combine together to form a single tree of files and directories. You can find out the file systems that are currently mounted using command


```
df -a
```

or command

```
mount
```

Note not all file systems are created on hard disk partitions. For example, a file system may be created in a CD rom and others may even be in the RAM.

Of all the mounted file systems, one is special - the root file system. Unlike other file systems, the root file system is where the system is booted from and it cannot be un-mounted. Other file systems can be mounted or un-mounted by the system administrator. For example, the system administrator may need to reformat a hard disk partition. In order to do so he would need to un-mount the file system that resides on that disk partition first. However if there is a process that has set its current directory to one of the directories on that file system, that file system cannot be un-mounted. Since a daemon is supposed to run forever, it is advisable that we do not set its current directory to a file system which may need to be un-mounted in the future, so as not to affect normal system administration.

The following function is designed to turn a normal process into a daemon. Once it is called, it will create a child process (which would be almost identical to its parent), and terminate the parent process. In this way, the shell will no longer wait for the process to terminate (as it is already terminated) and the shell can continue to accept the next command.

At this point, the child process is still under the same session, hence it still has a control terminal. The next function call, `setsid()`, creates a new session and puts itself into the new session. The new session does not have a control terminal.

The system call, `chdir("/")`, changes the current directory of the process to the root directory. The root directory is always on the root file system which is never unmounted.

The system call, `umask(0)`, clears the `umask` value of the process. This is so that the files can be created with the exact permissions specified in the program. We will discuss `umask` in Topic 5 on file systems.

```
int daemon_init(void)
{
   pid_t  pid;

   if ( (pid = fork() ) < 0) return (-1);
   else if (pid != 0) {
       printf("server pid=%d\n", pid);
       exit(0);
   }

   /* child continues */
   setsid();      /* become session leader */
   chdir("/");    /* change current directory */
   umask(0);      /* clear umask */

   return (0);
}
```

Since a daemon is not associated with any terminal, what would happen if the process wants to send any warning or diagnostic information to its user? Note that it cannot use function `printf` to send messages to the screen, as the `stdout` and `stderr` are no longer associated with the terminal screen. In Topic 5, we will explain more on standard input and standard output.

One common solution is to create a log file. The daemon simply sends the messages to the log file. The user can check the log file to find out what the daemon is up to. However, where to store the log file and what are to be logged are highly application dependant, therefore log file is not handled in the above function.

The following code demonstrates how to create a daemon process.

```
/* name:     useless.c */

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

```
#define  logfile "/tmp/useless.log"

// include daemon_init here

int main()
{
   FILE *log;
   pid_t pid;

   // create a log file
   log = fopen(logfile, "w+");
   if (!log) {
      fprintf(stderr,
       "cannot create log file %s\n", logfile);
      exit(1);
   }

   // turn the process into a daemon
   daemon_init();

   // log daemon pid
   pid = getpid();
   fprintf(log, "My pid is %d\n", pid);
   fflush(log);

   // pretend to do something
   while (1) {
      sleep(100);
      fprintf(log,
        "Who says that I am useless?\n")?
      fflush(log);
   }
}
```