UNIX PROCESSES

MAIN FUNCTION

- **PROTOTYPE:**

  int main(int argc, char *argv[ ]);

Argc – is the number of command line arguments

argv [ ] – is an array of pointers to the arguments

- A **C program** is started by a **kernel**
- A special **start up routine** is called before the **main function** is called
- This start up routine takes values from the kernel and sets things up so that the main function is called

Process termination

- Normal termination
  - * return from main
  - * calling exit
  - * calling _exit
- Abnormal termination
  - * calling abort
  - * terminated by a signal

exit and _exit functions

- _exit returns to kernel immediately
- exit performs certain cleanup processing and then returns to kernel
- PROTOTYPE

```
#include <stdlib.h>
void _exit (int status)
void exit (int status)
```

- The exit status is undefined if

1. Either of these function is called without an exit status
2. Main does a return without a return value
3. Main “falls of the end”

At exit function

- With ANSI C a process can register up to 32 functions that are called by exit ---called exit handlers
- Exit handlers are registered by calling the atexit function

```
#include <stdlib.h>
Int atexit (void (*fun) void));
```

- **Atexit function calls these functions in reverse order of their registration**
- **Each function is called as many times as it was registered**

```
#include   "ourhdr.h"
static void my_exit1(void), my_exit2(void);
 int main(void)
 {
   if (atexit(my_exit2) != 0)
        err_sys("can't register my_exit2");
   if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
   if (atexit(my_exit1) != 0)
        err_sys("can't register my_exit1");
   printf("main is done\n");
   return(0);
}
```

```
static void
my_exit1(void)
{
    printf("first exit handler\n");
}
static void
my_exit2(void)
{
    printf("second exit handler\n");
}
```

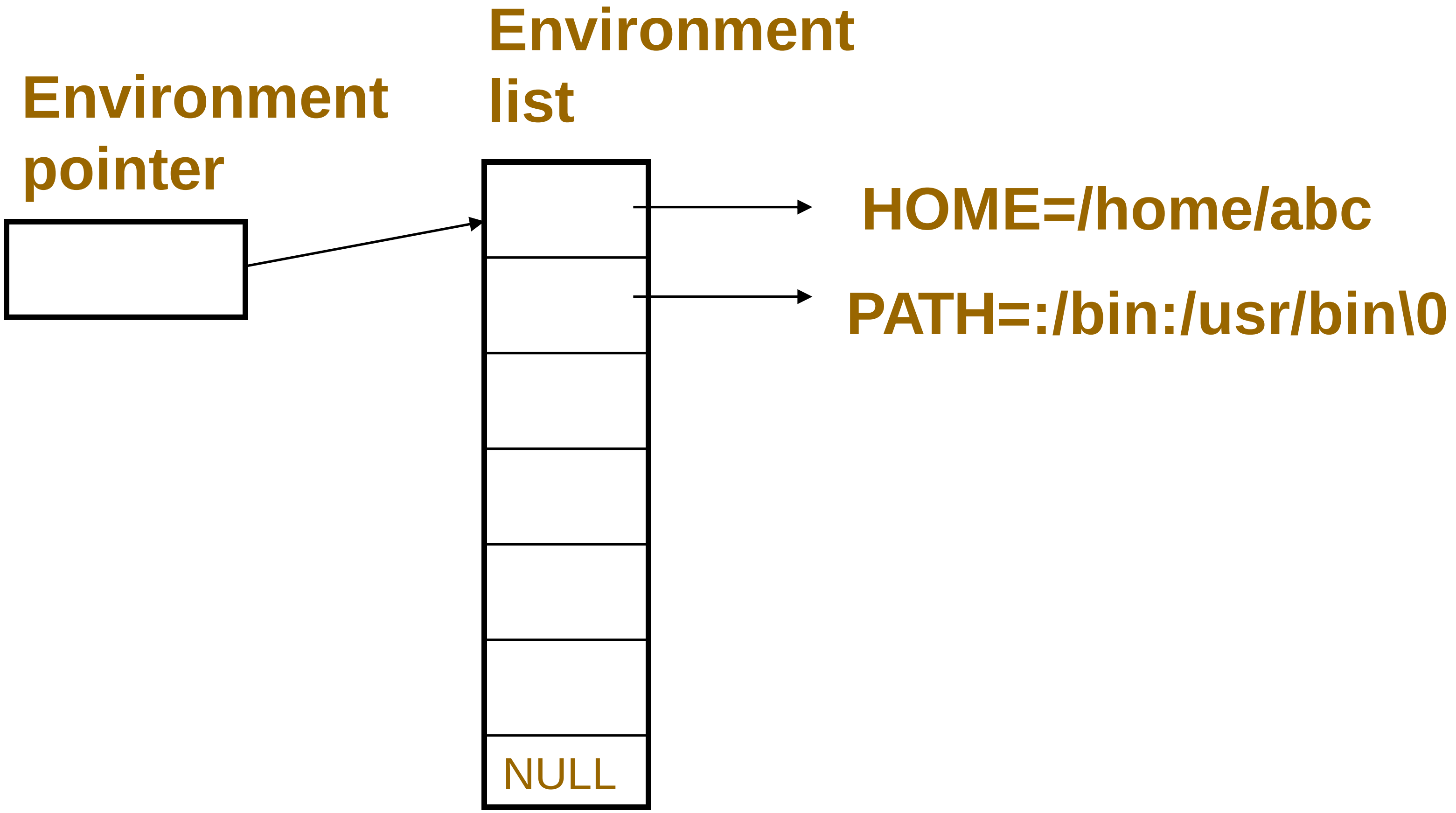
Command-line arguments

- /* program to echo command line arguments*/

```
int main (int argc, char* argv[ ])
 {
   for(int i=0;i<argc ;i++)
    {
       printf("argv[%d]:%s \n",I,argv[i]);
    }
 }
```

Environment list

- Environment list – is an array of character pointers ,where each pointer contains the address of a null terminated C string
- The address of array of pointers is contained in global variable environ
- extern char **environ;
- each string is of the form name=value

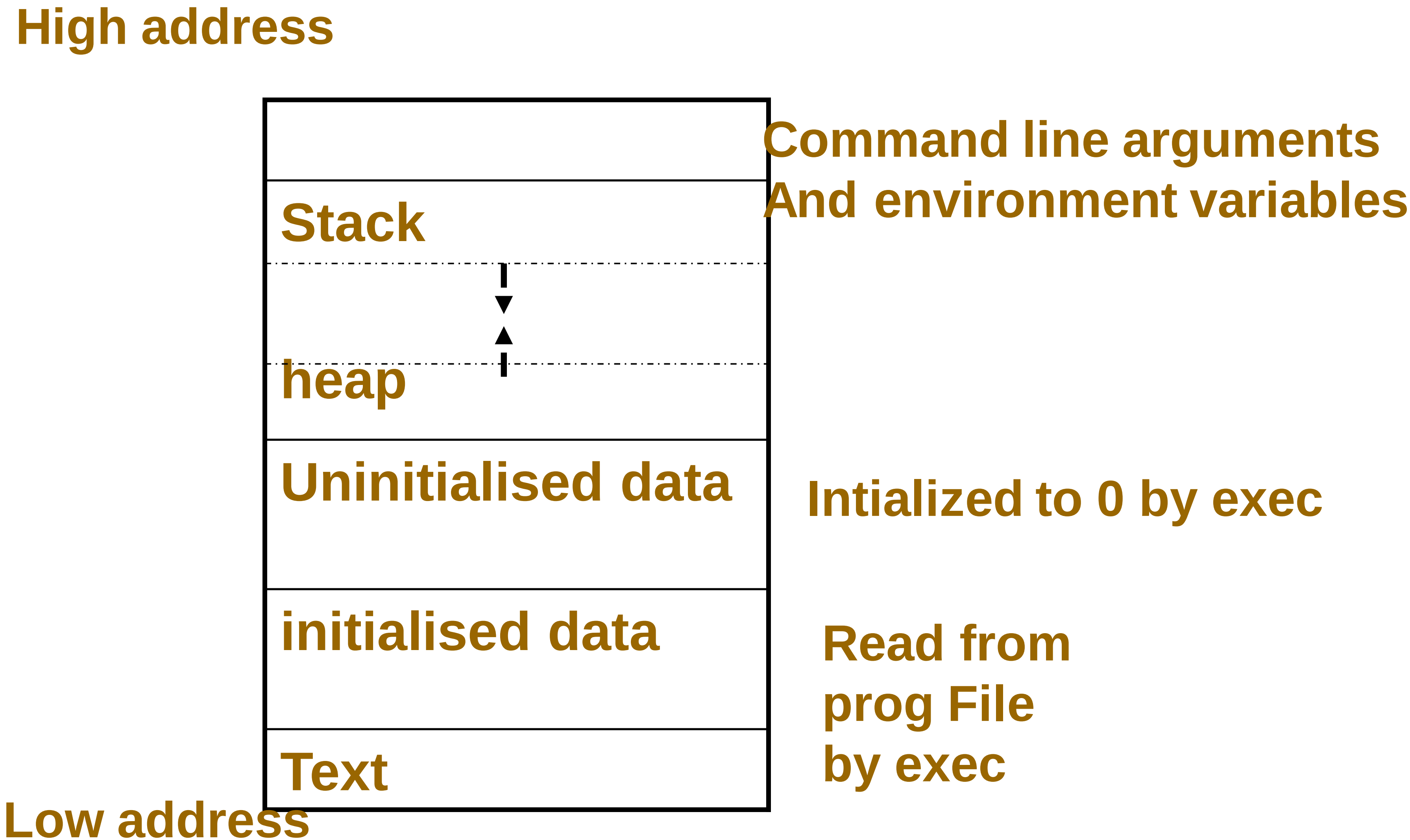

Environment pointer
Environment list
HOME=/home/abc
PATH=:/bin:/usr/bin\0
NULL

Memory layout of a C program

- **Text segment** – sharable copy
- **Initialized data segment** – variables

  specifically initialized in the program
- **Uninitialized data segment** – “bss” segment

  data is initialized to arithematic 0 or null
- **Stack** – return address and information about caller’s environment
- **Heap –** dynamic memory allocation takes place on the heap


High address
Command line arguments
And environment variables
Stack
heap
Uninitialised data
Intialized to 0 by exec
initialised data
Read from
prog File
by exec
Text
Low address

Shared libraries

- **Shared libraries remove the common library routines from the executable file , instead maintaining a single copy of the library routine some where in memory that all processes reference**
- **Advantage: reduces size of executable file**

  easy to replace with a newer version
- **Disadvantage: some- runtime overhead**

Memory allocation

- malloc : allocates specified number of bytes of memory
- calloc : allocates specified number of objects of specified size
- realloc : changes size of previous allocated area

```
#include <stdlib.h>
void *malloc (size_t size);
void *calloc (size_t nobj, size_t size);
void *realloc (void *ptr, size_t newsize);
```

realloc may increase or decrease the size of previously allocated area .If it decreases the size no problem occurs But if the size increases then………….

1. **Either there is enough space then the memory is reallocated and the same pointer is returned**
2. **If there is no space then it allocates new area copies the contents of old area to new area frees the old area and returns pointer to the new area**

Alloca function

- It is same as malloc but instead of allocating memory from heap, the memory allocated from the stack frame of the current function

Environment variables

- Environment strings are of the form name=value
- ANSI C defined functions

```
#include <stdlib.h>
char *getenv (const char *name);
int putenv (const char *str);
int setenv (const char *name, const char
                    *value ,int rewrite);
void unsetenv (const char *name);
```

- **Getenv : fetches a specific value from the environment**
- **Putenv : takes a string of the form name=value , if it already exists then old value is removed**
- **Setenv : sets name to value. If name already exists then a) if rewrite is non zero, then old definition is removed**

  b) if rewrite is zero old definition is retained
- **Unsetenv : removes any definition of name**

- **Removing an environment variable is simple just find the pointer and move all subsequent pointers down one**
- **But while modifying**

  *** if size of new value<=size of new value**

  just copy new string over the old string

  *** if new value >oldvalue use malloc obtain room for new string, copy the new string to this area and replace the old pointer in environment list for name with pointer to this malloced area**

- **While adding a new name call malloc allocate room for name=value string and copy the string to this area**
- **If it’s the first time a new name is added ,**

  use malloc to obtain area for new list of pointers. Copy the old list of pointers to the malloced area and add the new pointer to its end
- **If its not the first time a new name was**

  added ,then just reallocate area for new pointer since the list is already on the

  heap

Set jump and long jump

- To transfer control from one function to another we make use of **setjmp** and **longjmp** functions

```
#include <stdio.h>
int setjmp (jmp_buf env);
void longjmp (jmp_buf env, int val);
```

- **env is of type jmp_buf ,this data type is form of array that is capable of holding all information required to restore the status of the stack to the state when we call longjmp**

- **Val allows us to have more than one longjmp for one setjmp**

```
#include  <setjmp.h>
#include  "ourhdr.h"

static void        f1(int, int, int);
static void        f2(void);

static jmp_buf  jmpbuffer;
int main(void)
{
    int count;
    register int     val;
    volatile int     sum;
```

```
count = 2; val = 3; sum = 4;
  if (setjmp(jmpbuffer) != 0) {
  printf("after longjmp: count = %d,
  val = %d, sum = %d\n", count, val, sum);
      exit(0);
  }
  count = 97; val = 98; sum = 99;
  /* changed after setjmp, before longjmp */
  f1(count, val, sum);
                        /* never returns */
}
```

```
static void
f1(int i, int j, int k)
{
   printf("in f1(): count = %d, val = %d,
 sum = %d\n", i, j, k);
   f2();
}
static void f2(void)
{
   longjmp(jmpbuffer, 1);
}
```

- **The state of automatic,register and volatile variables after longjmp**
- **If compiled with optimization**

getrlimit and setrlimit

```
#include <sys/time.h>
#include <sys/resource.h>
int getrlimit (int resource ,struct
                                        rlimit *rlptr);
int setrlimit (int resource ,const struct
                                        rlimit *rlptr);
```

Struct rlimit

```
{
  rlim_t rlim_cur;            /*soft limit*/
  rlim_t rlim_max;            /*hard limit */
}
```

1. **Soft link can be changed by any process to a value <= to its hard limit**
2. **Any process can lower its hard limit to a value greater than or equal to its soft limit**
3. **Only super user can raise hard limit**

- **RLIMIT_CORE – max size in bytes of a core file**
- **RLIMIT_CPU – max amount of CPU time in seconds**
- **RLIMIT_DATA – max size in bytes of data segment**
- **RLIMIT_FSIZE – max size in bytes of a file that can be created**
- **RLIMIT_MEMLOCK – locked in-memory address space**

- RLIMIT_NOFILE – max number of open files per process
- RLIMIT_NPROC – max number of child process per real user ID
- RLIMIT_OFILE – same as RLIMIT_NOFILE
- RLIMIT_RSS – max resident set size in bytes
- RLIMIT_STACK – max size in bytes of the stack
- RLIMIT_VMEM – max size in bytes of the mapped address space

```
#include   <sys/types.h>
#include   <sys/time.h>
#include   <sys/resource.h>
#include   "ourhdr.h"
#define    doit(name) pr_limits(#name, name)
static voidpr_limits(char *, int);
int main(void)
{
   doit(RLIMIT_CORE);
   doit(RLIMIT_CPU);
   doit(RLIMIT_DATA);
   doit(RLIMIT_FSIZE);
```

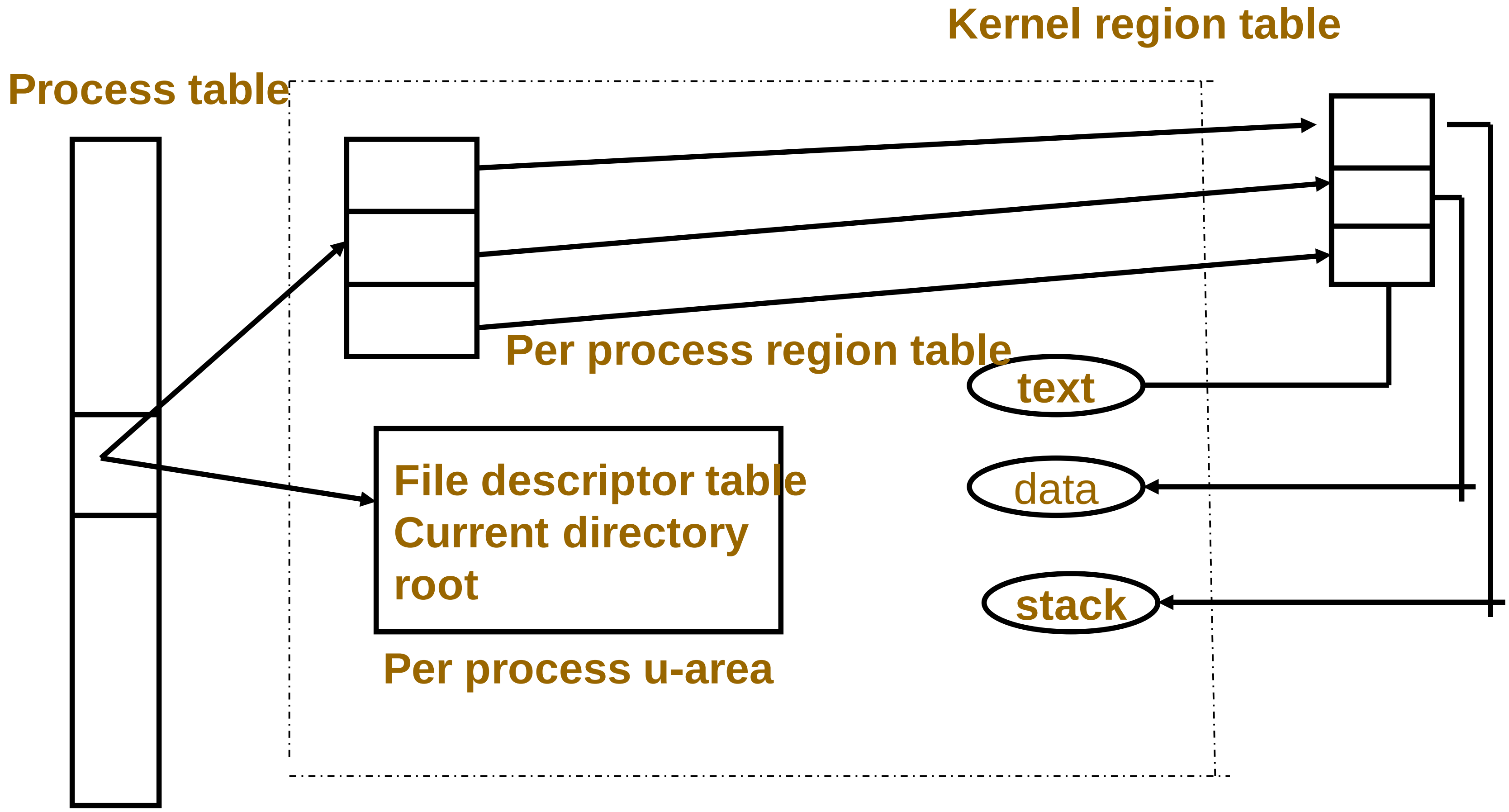
```
#ifdef      RLIMIT_MEMLOCK
   doit (RLIMIT_MEMLOCK);
#endif
#ifdef      RLIMIT_NOFILE /* SVR4 name */
   doit (RLIMIT_NOFILE);
#endif
#ifdef      RLIMIT_OFILE  /* 44BSD name */
   doit (RLIMIT_OFILE);
#endif
```

```
#ifdef      RLIMIT_NPROC
    doit (RLIMIT_NPROC);
#endif
#ifdef      RLIMIT_RSS
    doit(RLIMIT_RSS);
#endif
    doit(RLIMIT_STACK);
#ifdef      RLIMIT_VMEM
    doit(RLIMIT_VMEM);
#endif
    exit(0);
}
```

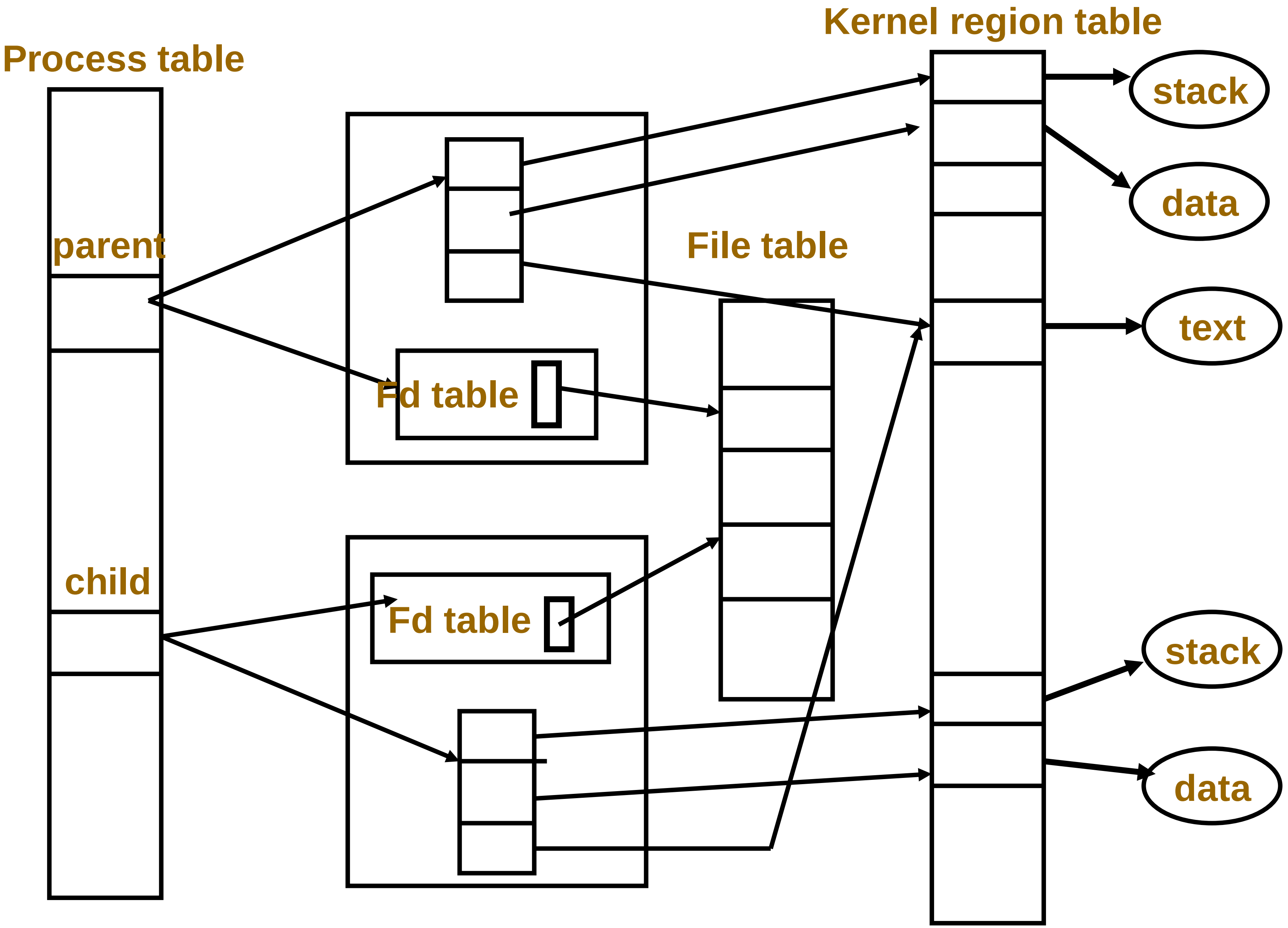
```
static void
pr_limits(char *name, int resource)
{
    struct rlimit   limit;
    if (getrlimit(resource, &limit) < 0)
        err_sys("getrlimit error for %s", name);
    printf("%-14s  ", name);
    if (limit.rlim_cur == RLIM_INFINITY)
        printf("(infinite)  ");
```

```
else
        printf("%10ld  ", limit.rlim_cur);
    if (limit.rlim_max == RLIM_INFINITY)
        printf("(infinite)\n");
    else
        printf("%10ld\n", limit.rlim_max);
}
```

Kernel support for processes



- A process consists of
- **A text segment** – program text of a process in machine executable instruction code format
- **A data segment** – static and global variables in machine executable format
- **A stack segment** – function arguments, automatic variables and return addresses of all active functions of a process at any time
- **U-area** is an extension of Process table entry and contains process-specific data



Kernel region table
Process table
stack
data
parent
File table
text
Fd table
Fd table
child
stack
data

Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- Current working directory

- Root directory
- Signal handling
- Signal mask and dispositions
- Umask
- Nice value
- The difference between the parent & child
- The process ID
- Parent process ID
- File locks
- Alarms clock time
- Pending signals

Process identifiers

- Every process has a unique process ID, a non negative integer
- Special processes : process ID 0 scheduler process also known as swapper
  process ID 1 init process
  init process never dies ,it’s a normal user process run with super user privilege
  process ID 2 pagedaemon

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

Fork function

- The only way a new process is created by UNIX kernel is when an existing process calls the fork function

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

- The new process created by fork is called child process
- The function is called once but returns twice
- The return value in the child is 0
- The return value in parent is the process ID of the new child
- The child is a copy of parent
- Child gets a copy of parents text, data , heap and stack
- Instead of completely copying we can use COW copy on write technique

```
#include   <sys/types.h>
#include   "ourhdr.h"
int        glob = 6;
/* external variable in initialized data */
char buf[ ] = "a write to stdout\n";
int main(void)
{
    int        var;
/* automatic variable on the stack */
    pid_t      pid;
```

```
var = 88;
  if (write(STDOUT_FILENO, buf,
  sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");

       printf("before fork\n");
  if ( (pid = fork()) < 0)
        err_sys("fork error");
  else if (pid == 0)
  {                                     /* child */
        glob++;         /* modify variables */
        var++;
  }
```

```
else
        sleep(2);
        /* parent */

    printf("pid = %d, glob = %d, var = %d\n",
    getpid(), glob, var);
    exit(0);
}
```

File sharing

- **Fork creates a duplicate copy of the file descriptors opened by parent**
- **File Table is shared Offset changes are reflected both in parent and child**
- **There are two ways of handling descriptors after fork**

1. **The parent waits for the child to complete**
2. **After fork the parent closes all descriptors that it doesn’t need and the child does the same thing**

Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- Current working directory

- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Resource limits

The difference between the parent and child

- The return value of fork
- The process ID
- Parent process ID
- The values of tms_utime , tms_stime , tms_cutime , tms_ustime is 0 for child
- file locks set by parent are not inherited by child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to empty set

- **The functions of fork**

1. **A process can duplicate itself so that parent and child can each execute different sections of code**

2. **A process can execute a different program**

vfork

- It is same as fork
- It is intended to create a new process when the purpose of new process is to exec a new program
- The child runs in the same address space as parent until it calls either exec or exit
- vfork guarantees that the child runs first ,

  until the child calls exec or exit

```
int glob = 6;
   /* external variable in initialized data */
int main(void)
{
   int  var;
       /* automatic variable on the stack */
   pid_t     pid;
   var = 88;
   printf("before vfork\n");
    if ( (pid = vfork()) < 0)
        err_sys("vfork error");
```

```
else if (pid == 0) {          /* child */
        glob++;
    /* modify parent's variables */
        var++;
        _exit(0);             /* child terminates */
    }
    /* parent */
    printf("pid = %d, glob = %d, var = %d\n",
    getpid(), glob, var);
    exit(0);
}
```

exit functions

- Normal termination

1. Return from main
2. Calling exit – includes calling exit handlers
3. Calling _exit – it is called by exit function

- Abnormal termination

1. Calling abort – SIGABRT
2. When process receives certain signals

- **Exit status** is used to notify parent how a child terminated
- When a parent terminates before the child, the child is **inherited by init** process
- If the child terminates before the parent then the information about the is obtained by parent when it executes **wait or waitpid**
- The information consists of the **process ID,** the **termination status** and amount of **CPU time** taken by process

- **A process that has terminated , but whose parents has not yet waited for it, is called a zombie**
- **When a process inherited by init terminates it doesn’t become a zombie**
- **Init executes one of the wait functions to fetch the termination status**

Wait and waitpid functions

- When a child id terminated the parent is notified by the kernel by sending a SIGCHLD signal
- The termination of a child is an asynchronous event
- The parent can ignore or can provide a function that is called when the signal occurs

- **The process that calls wait or waitpid can**

1. **Block**
2. **Return immediately with termination status of the child**
3. **Return immediately with an error**

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

- Statloc is a pointer to integer
- If statloc is not a null pointer ,the termination status of the terminated process is stored in the location pointed to by the argument
- The integer status returned by the two functions give information about exit status, signal number and about generation of core file

- **Macros which provide information about how a process terminated**

WIFEXITED	**TRUE – if child terminated normally** **WEXITSTATUS – is used to fetch the lower 8 bits of argument child passed to exit or _exit**

WIFSIGNALED	**TRUE – if child terminated abnormally** **WTERMSIG – is used to fetch the signal number that caused termination** **WCOREDUMP – is true is core file was generated**
WIFSTOPPED	**TRUE – for a child that is currently stopped** **WSTOPSIG -- is used to fetch the signal number that caused child to stop**

waitpid

- The interpretation of pid in waitpid depends on its value

1. Pid == -1 – waits for any child
2. Pid > 0 – waits for child whose process ID equals pid
3. Pid == 0 – waits for child whose process group ID equals that of calling process
4. Pid < -1 – waits for child whose process group ID equals to absolute value of pid

- **Waitpid** helps us wait for a particular process
- It is **nonblocking** version of wait
- It supports **job control**

WNOHANG	Waitpid will not block if the child specified is not available
WUNTRACED	supports job control. Returns the status of stopped child

```
Void pr_exits(int status)
{
  if WIFEXITED(status)
    printf(“%d”,WEXITEDSTATUS(status));
Else if WIFSIGNALED(status)
   printf(“%d”,WTERMSIG(status),
```

```
#ifdef WCOREDUMP
  WCOREDUMP(status)? “core file
   generated”:””””);
#else   “”);
#endif
Else if(WIFSTOPPED(status))
Printf(“%d”,WSTOPSIG(status));
}
```

```
#include   <sys/types.h>
#include   <sys/wait.h>
#include   "ourhdr.h"
Int main(void)
{
    pid_t      pid;
    int        status;
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);
```

```
if (wait(&status) != pid)
                                   /* wait for child */
      err_sys("wait error");
      pr_exit(status);
                             /* and print its status */
   if ( (pid = fork()) < 0)
      err_sys("fork error");
   else if (pid == 0)               /* child */
      abort();
                    /* generates SIGABRT */
```

```
if (wait(&status) != pid)
                                    /* wait for child */
        err_sys("wait error");
    pr_exit(status);
                            /* and print its status */
    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)               /* child */
        status /= 0;
            /* divide by 0 generates SIGFPE */
```

```
if (wait(&status) != pid)
                                         /* wait for child */
      err_sys("wait error");
      pr_exit(status);
                              /* and print its status */

  exit(0);
}
```

wait3 and wait4 functions

- These functions are same as waitpid but provide additional information about the resources used by the terminated process
- Resource Info includes
  - User & System CPU time,No. of page faults,
  - No. of signals etc..

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

Struc rusage

- struct timeval ru_utime; /* user time used */
- struct timeval ru_stime; /* system time used */
- long ru_maxrss; /* maximum resident set size */ long ru_idrss; /* integral resident set size */
- long ru_minflt; /* page faults not requiring physical I/O */
- long ru_majflt; /* page faults requiring physical I/O */
- long ru_nswap; /* swaps */
- long ru_inblock; /* block input operations */
- long ru_oublock; /* block output operations */
- long ru_msgsnd; /* messages sent */
- long ru_msgrcv; /* messages received */
- long ru_nsignals; /* signals received */
- long ru_nvcsw; /* voluntary context switches */
- long ru_nivcsw; /* involuntary context switches

```
pid=fork();//parent
If(pid==0)//first child
{
  pid1=fork();//second
 if(pid1>0)//first
   exit(0);
```

```
sleep(2);//2nd
printf("%d",getppid());
}
//parent
waitpid(pid,NULL,0);
exit(0);
```

Race condition

- **Race condition occurs when multiple process are trying to do something with shared data and final out come depends on the order in which the processes run**
- **If a child wants parent to finish**
- **While(getppid()!=1)**
- **sleep(1)**
- **TELL_WAIT,WAIT_PARENT,WAIT_CHILD, TELL_CHILD,TELL_PARENT**

Program with race condition

```
#include   <sys/types.h>
#include   "ourhdr.h"
static void charatatime(char *);
int main(void)
{
   pid_t      pid;
   if ( (pid = fork()) < 0)
       err_sys("fork error");
```

```
else if (pid == 0)
    {
        charatatime("output from child\n");
    }
  else
    {
        charatatime("output from parent\n");
    }
    exit(0);
}
```

```
static void
charatatime(char *str)
{
    char        *ptr;
    int         c;
    setbuf(stdout, NULL);
                              /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

```
/*altered program*/
#include   <sys/types.h>
#include   "ourhdr.h"
static void charatatime(char *);
Int main(void)
{
   pid_t     pid;
   TELL_WAIT();
   if ( (pid = fork()) < 0)
        err_sys("fork error");
```

```
else if (pid == 0)
  {
   WAIT_PARENT();          /* parent goes first */
        charatatime("output from child\n");
  }
  else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
   }
   exit(0);
}
```

```c
static void charatatime(char *str)
{
    char        *ptr;
    int         c;
    setbuf(stdout, NULL);
                                /* set unbuffered */
    for (ptr = str; c = *ptr++; )
        putc(c, stdout);
}
```

exec functions

- **Exec replaces the calling process by a new program**
- **The new program has same process ID as the calling process**
- **No new process is created , exec just replaces the current process by a new program**
- **Starts from main**

- File name:if slash in it treat as path name
- Otherwise search in dir looking at PATH variable.
- If not executable invoke as shell script
- Env can be passed
- Last is 0. type cast with char * as diff mem requirement for int and char*
- l->list,v->vector,e->env

- Six Different exec functions

```
#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 [ ]);
int execlp (const char *filename, const
                 char *arg0 ,… /*(char *) 0*/);
int execvp (const char *filename ,char
                                    *const argv[ ] );
```

New program inherits following properties from calling process

Pid, ppid, ruid, rgid, sgid, pgid, sid,

ct, alarm clock, current working dir, root dir,

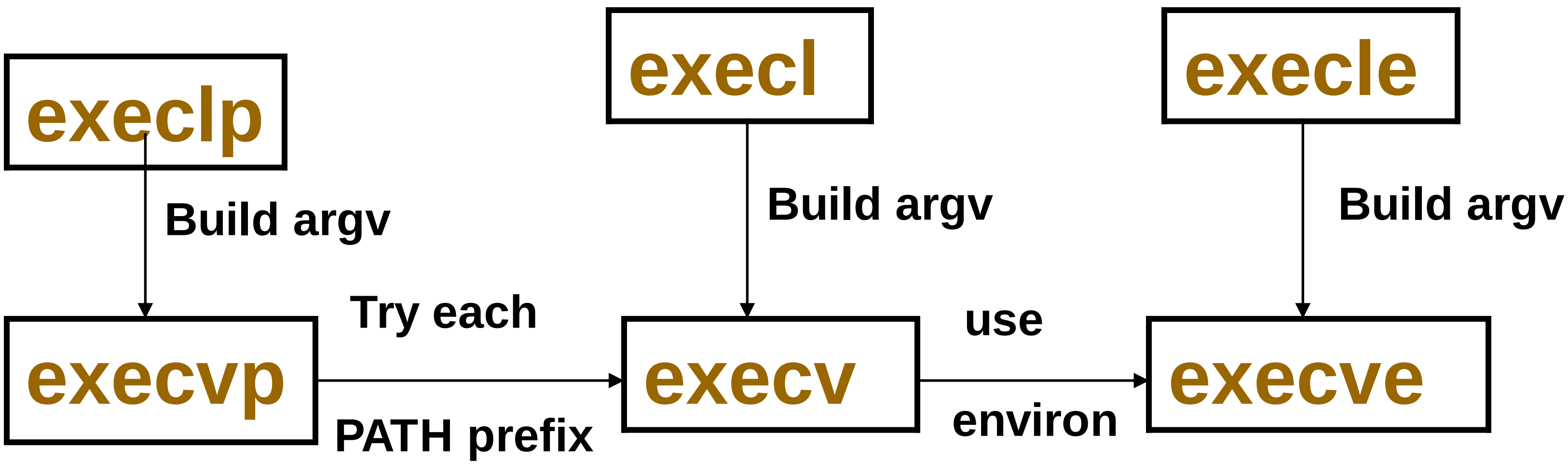
file mode creation mask, file locks,

process signal mask, pending signals,

resource limits, tms times

Close on Exec:fd will kept open if flag not set.

- **Relation between exec functions**



```
#include  <sys/types.h>
#include  <sys/wait.h>
#include  "ourhdr.h"
char *env_init[ ] =
{ "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
   pid_t     pid;
   if ( (pid = fork()) < 0)
        err_sys("fork error");
else if (pid == 0) {
/* specify pathname, specify environment */
```

```
if ( execle ("/home/stevens/bin/echoall",
  "echoall", "myarg1", "MY ARG2",
(char *)  0,   env_init) < 0)
               err_sys("execle error");
  }
  if (waitpid(pid, NULL, 0) < 0)
      err_sys("wait error");


  if ( (pid = fork()) < 0)
      err_sys("fork error");
```

```
else if (pid == 0) {
/* specify filename, inherit environment */
        if (execlp("echoall",
                "echoall", "only 1 arg",
                                (char *) 0) < 0)
                err_sys("execlp error");
    }
    exit(0);
}
```

Changing user IDs and group IDs

- **Prototype**

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

- Rules

1. **If the process has superuser privilege, the setuid function sets – real user ID, effective user ID , saved set-user-ID to uid**
2. **If the process doesnot have superuser privilege, but uid equals either real user ID or saved set-user-ID, setuid sets only effective user ID to uid**
3. **If neither of the two conditions is true, errno is set to EPERM and an error is returned**

ID	exec	exec
	Set-user-ID bit off	Set-user-Id bit on
Real user ID	unchanged	unchanged
Effective user ID	unchanged	Set from user ID of program file
Saved set user ID	copied from effective user ID	copied from effective user ID

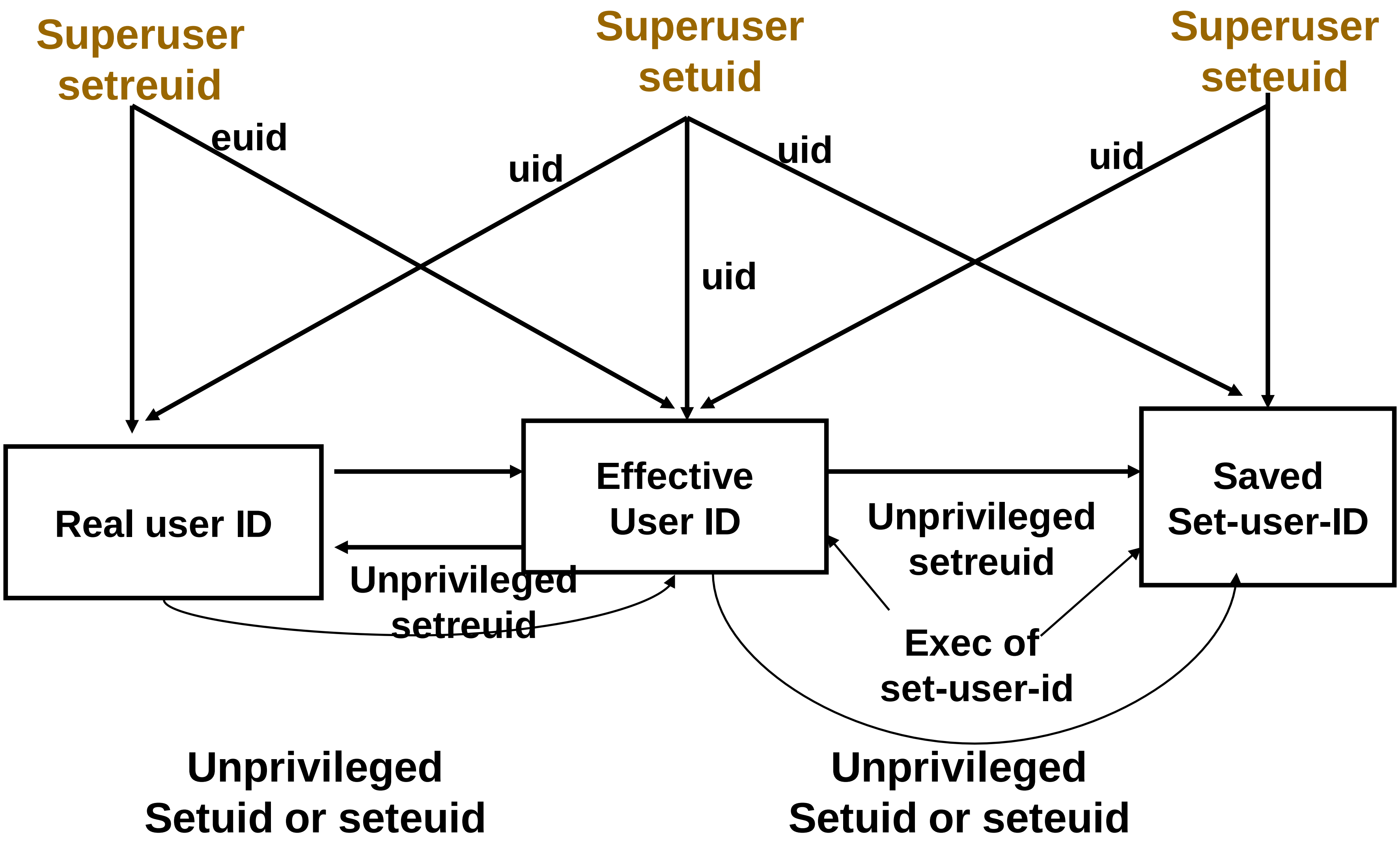
ID		Super user	Un privileged user
Real user ID		Set to uid	unchanged
Effective user ID		Set to uid	Set to uid
Saved set-user ID		Set to uid	unchanged

setreuid and setregid

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

seteuid and setegid

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

int seteuid (uid_t euid);
int setegid (gid_t egid);
```



Superuser
setreuid
Superuser
setuid
Superuser
seteuid
euid
uid
uid
uid
uid
Real user ID
Effective
User ID
Saved
Set-user-ID
Unprivileged
setreuid
Unprivileged
setreuid
Exec of
set-user-id
Unprivileged
Setuid or seteuid
Unprivileged
Setuid or seteuid

Interpreter files

- Files which begin with a line of the form

  #! pathname [optional argument]

most common example :

  #! /bin/bash

- The actual file execed by kernel is the one specified in the pathname

- Execl(“/home/stevems/bin/testinterp”, “testinterp”,”myarg1”,”Myarg2”,(char*)0);
- $cat /home/stevems/bin/testinterp

#! /home/stevems/bin/echoarg foo

Arg0-> /home/stevems/bin/echoarg

Arg1->foo

Arg2-> /home/stevems/bin/testinterp

Arg3-> myarg1,arg4->Myarg2

/*example of interpreter file*/

```
#!/bin/awk -f

BEGIN
 {
   for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
   exit
}
```

- **Uses of interpreter files**

1. They hide the fact that certain programs are scripts in some other language
2. They provide an efficiency gain
3. They help us write shell scripts using shells other than /bin/sh

system function

- It helps us execute a command string within a program
- System is implemented by calling fork, exec and waidpid

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

- **Return values of system function**

- **-1 – if either fork fails or waitpid returns an error other than EINTR**
- **127 -- If exec fails [as if shell has executed exit]**
- **termination status of shell -- if all three functions succeed**

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

int system(const char *cmdstring)
   /* version without signal handling */
{
   pid_t     pid;
   int       status;
```

```
if (cmdstring == NULL)
        return(1);
/* always a command processor with Unix */
   if ( (pid = fork()) < 0)
{
        status = -1;
   /* probably out of processes */

   } else if (pid == 0)
  {                                              /* child */
   execl("/bin/sh", "sh", "-c", cmdstring,
                                                        (char *) 0);
        _exit(127);                          /* execl error */
   }
```

```
else {                                        /* parent */
        while (waitpid(pid, &status, 0) < 0)
                if (errno != EINTR) {
                        status = -1;
 /* error other than EINTR from waitpid() */
                        break;
                }
  }
  return(status);
}
```

```c
/* calling system function*/
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
int main(void)
{
    int             status;

    if ( (status = system("date")) < 0)
        err_sys("system() error");
    pr_exit(status);
```

```
if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
    pr_exit(status);

    if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
    pr_exit(status);

    exit(0);
}
```

Process accounting

- **Process accounting : when enabled kernel writes an accounting record each time a process terminates**

- **Accounting records : 32 bytes of binary data**

```
Struct acct
 {
   char ac_flag;
   char ac_stat;
   uid_t ac_uid;
   gid_t ac_gid;
   dev_t ac_ttty;
   time_t ac_btime;
   comp_t ac_utime;
   comp_t ac_stime;
   comp_t ac_etime;
   comp_t ac_mem;
   comp_t ac_io;
   comp_t ac_rw;
   char ac_comm;
}
```

```
/*prog: to generate accounting data */
#include   <sys/types.h>
#include   <sys/acct.h>
#include   "ourhdr.h"
#define    ACCTFILE        "/var/adm/pacct"
static unsigned long
        compt2ulong(comp_t);
int main(void)
{
   struct acct          acdata;
   FILE                 *fp;
```

```
if ( (fp = fopen(ACCTFILE, "r")) == NULL)
        err_sys("can't open %s", ACCTFILE);
   while
   (fread(&acdata, sizeof(acdata), 1, fp) == 1)
 { printf("%-*.*s  e = %6ld, chars = %7ld, "
                 "stat = %3u: %c %c %c %c\n",
                     sizeof(acdata.ac_comm),
                     sizeof(acdata.ac_comm),
                     acdata.ac_comm,
                 compt2ulong(acdata.ac_etime),
                     compt2ulong(acdata.ac_io),
             (unsigned char)    acdata.ac_stat,
```

```
#ifdef      ACORE
    /* SVR4 doesn't define ACORE */
    acdata.ac_flag & ACORE ? 'D' : ' ',
#else
                                                    ' ',
#endif
#ifdef      AXSIG
              /* SVR4 doesn't define AXSIG */
             acdata.ac_flag & AXSIG ? 'X' : ' ',
#else
                                                    ' ',
#endif
```

```
  acdata.ac_flag & AFORK ? 'F' : ' ',
    acdata.ac_flag & ASU   ? 'S' : ' ');
    }
if (ferror(fp))
        err_sys("read error");
    exit(0);
}
```

```
static unsigned long
compt2ulong(comp_t comptime)
/* convert comp_t to unsigned long */
{
    unsigned long       val;
    int                 exp;
    val = comptime & 017777;
                            /* 13-bit fraction */
    exp = (comptime >> 13) & 7;
                        /* 3-bit exponent (0-7) */
    while (exp-- > 0)
        val *= 8;
    return(val);
}
```

User identification

- **To obtain the login name**

```
#include <unistd.h>
 char *getlogin (void);
```

Process times

```
#include <sys/times.h>
clock_t times (struct tms *buf);
```

- Struct tms {

```
 clock_t tms_utime;
clock_t tms_stime;
clock_t tms_cutime;
clock_t tms_cstime;
}
```

```
#include  <sys/times.h>
#include  "ourhdr.h"
static void
    pr_times (clock_t, struct tms *, struct tms *);
static void        do_cmd(char *);
int main (int argc, char *argv[ ])
{            int       i;
   for (i = 1; i < argc; i++)
        do_cmd(argv[i]);
            /* once for each command-line arg */
   exit(0);
}
```

```
static void
do_cmd (char *cmd)
/* execute and time the "cmd" */
{
    struct tms      tmsstart, tmsend;
    clock_t         start, end;
    int             status;
    fprintf(stderr, "\ncommand: %s\n", cmd);
    if ( (start = times(&tmsstart)) == -1)
            /* starting values */
        err_sys("times error");
```

```
if ( (status = system(cmd)) < 0)
                    /* execute command */
        err_sys("system() error");
    if ( (end = times(&tmsend)) == -1)
                                    /* ending values */
        err_sys("times error");
    pr_times(end-start, &tmsstart, &tmsend);
    pr_exit(status);
}
```

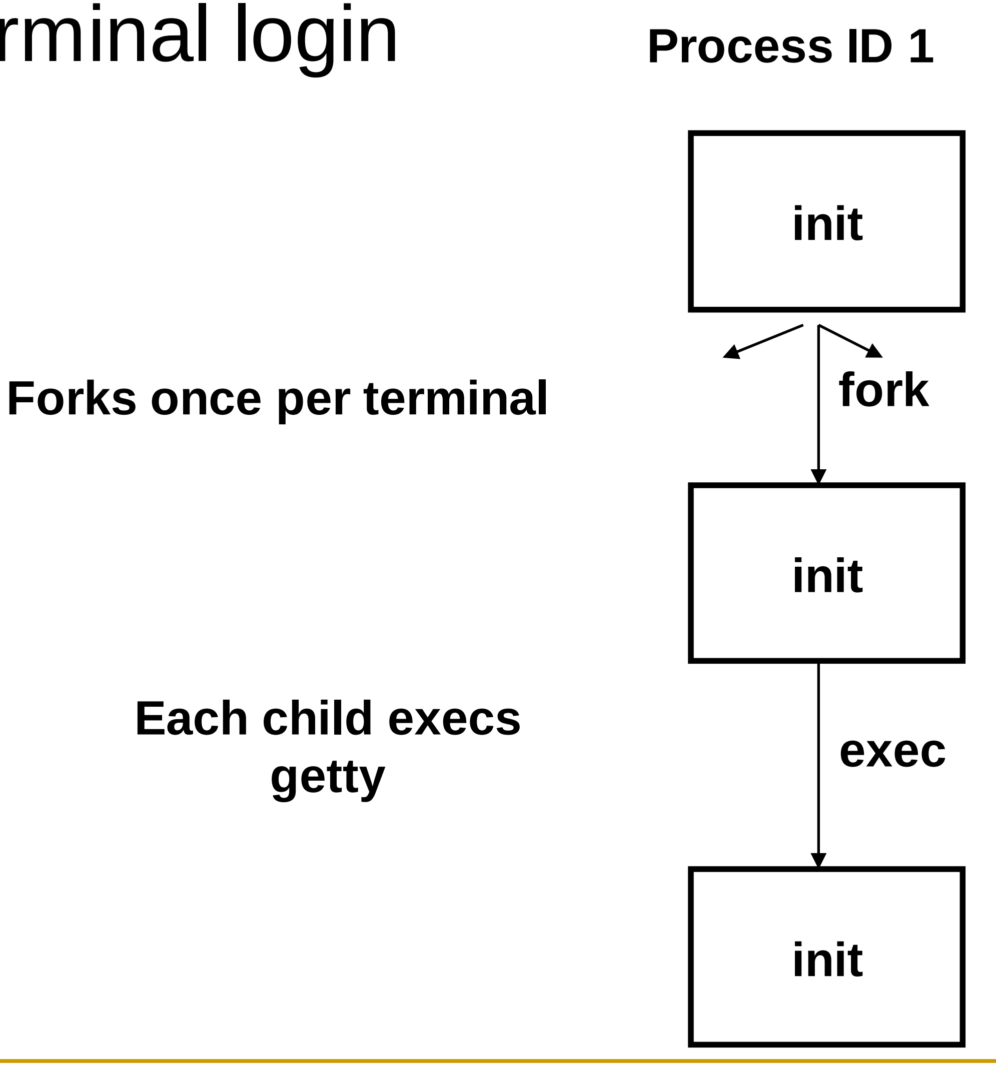
```
static void
  pr_times (clock_t real, struct tms *tmsstart,
                                    struct  tms *tmsend)
{  static long clktck = 0;
   if (clktck == 0)
    /* fetch clock ticks per second first time */
   if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
        err_sys("sysconf error");
   fprintf (stderr, "  real:  %7.2f\n",
   real / (double) clktck);
```

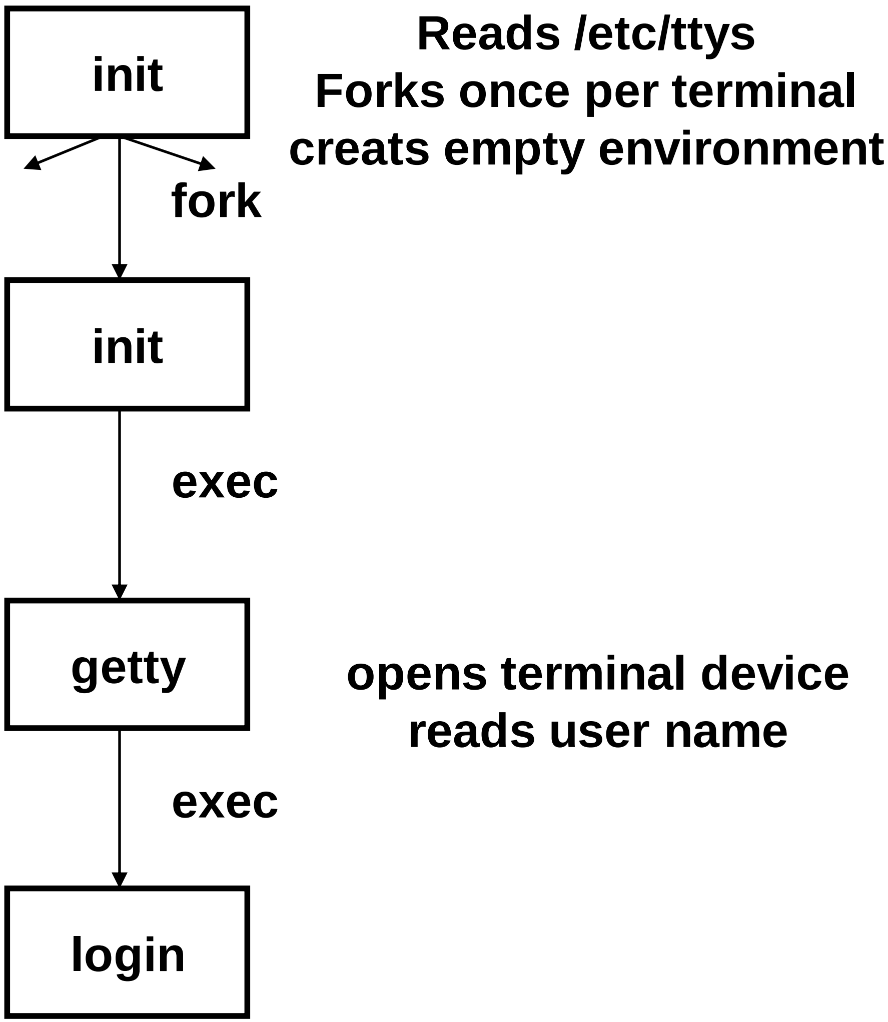
```
fprintf (stderr, "  user:  %7.2f\n",
    (tmsend->tms_utime - tmsstart>
    tms_utime) /  (double) clktck);

fprintf(stderr, "  sys:   %7.2f\n",
(tmsend->tms_stime - tmsstart-
    >tms_stime) / (double) clktck);
```

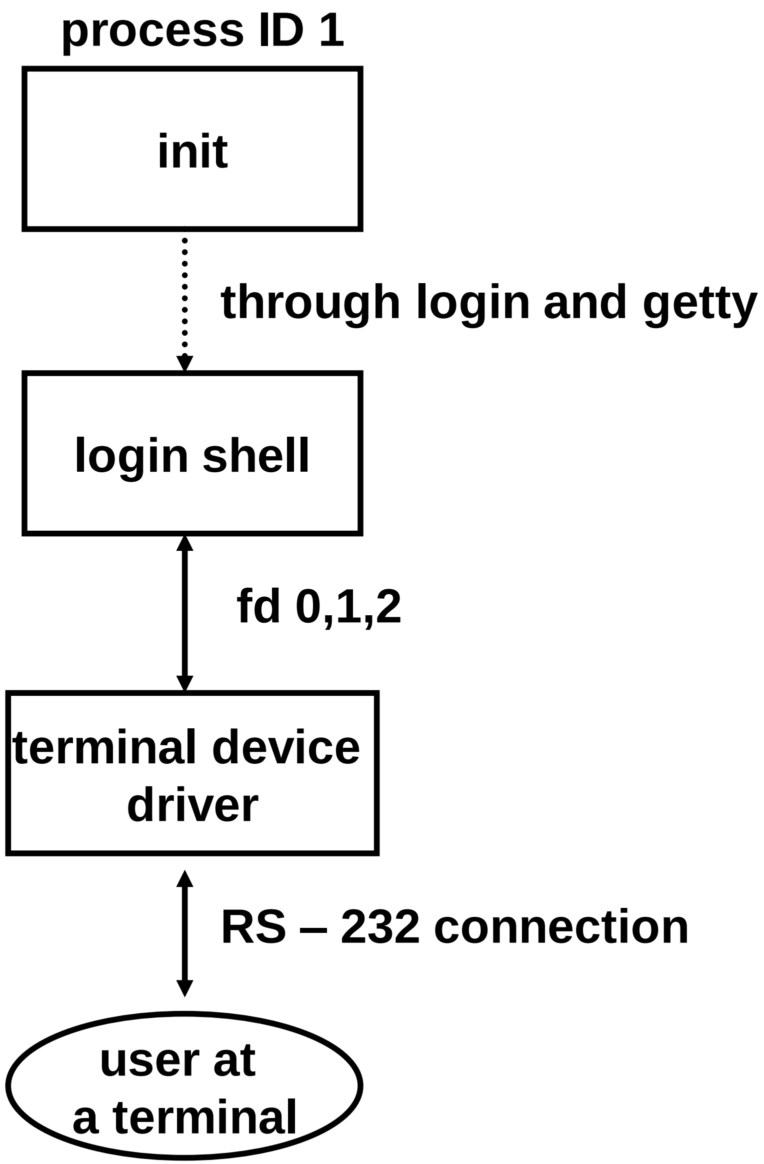
```
fprintf(stderr, "  child user:  %7.2f\n",
   (tmsend->tms_cutime - tmsstart->
   tms_cutime) / (double) clktck);

fprintf (stderr, "  child sys:   %7.2f\n",
   (tmsend->tms_cstime - tmsstart->
   tms_cstime) / (double) clktck);
}
```

Terminal logins

- 4.3+BSD terminal login



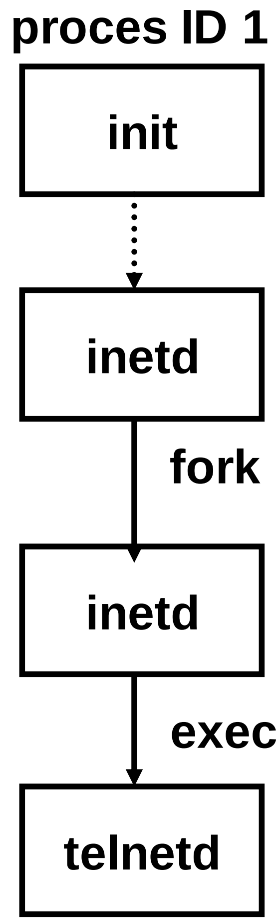

init
Reads /etc/ttys
Forks once per terminal
creats empty environment
fork
init
exec
getty
opens terminal device
reads user name
exec
login

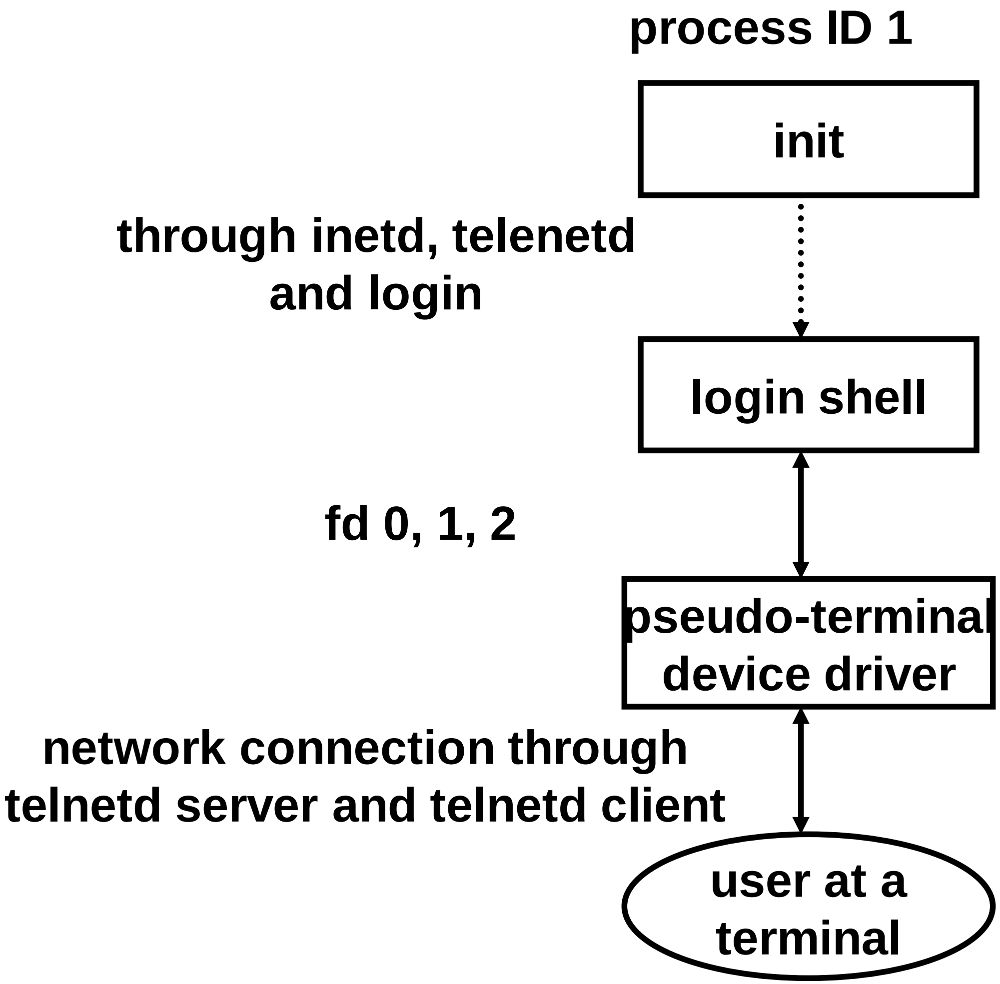

process ID 1
init
through login and getty
login shell
fd 0,1,2
terminal device driver
RS – 232 connection
user at a terminal

network login

fork/exec of/bin/sh
which executes shell script /etc/rc

when connection request arives
from telnet user




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

Process groups

- **A process group is a collection of one or more processes.**
- **Each process group has a unique process group ID.**
- **Process group IDs are similar to process IDs---they are positive integers and they can be stored in a pid_t data type.**
- **The function getpgrp returns the process group ID of the calling process.**

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

- **Each process group can have a process leader. The leader is identified by having its process group ID equal its process ID.**

- It is possible for a process group leader to create a process group, create processes in the group, and then terminate.
- The process group still exists, as long as there is at least one process in the group, regardless whether the group leader terminates or not
- process group lifetime — the period of time that begins when the group is created and ends when the last process in the group leaves the group

- **A process joins an existing process group, or creates a new process group by calling setpgid.**

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

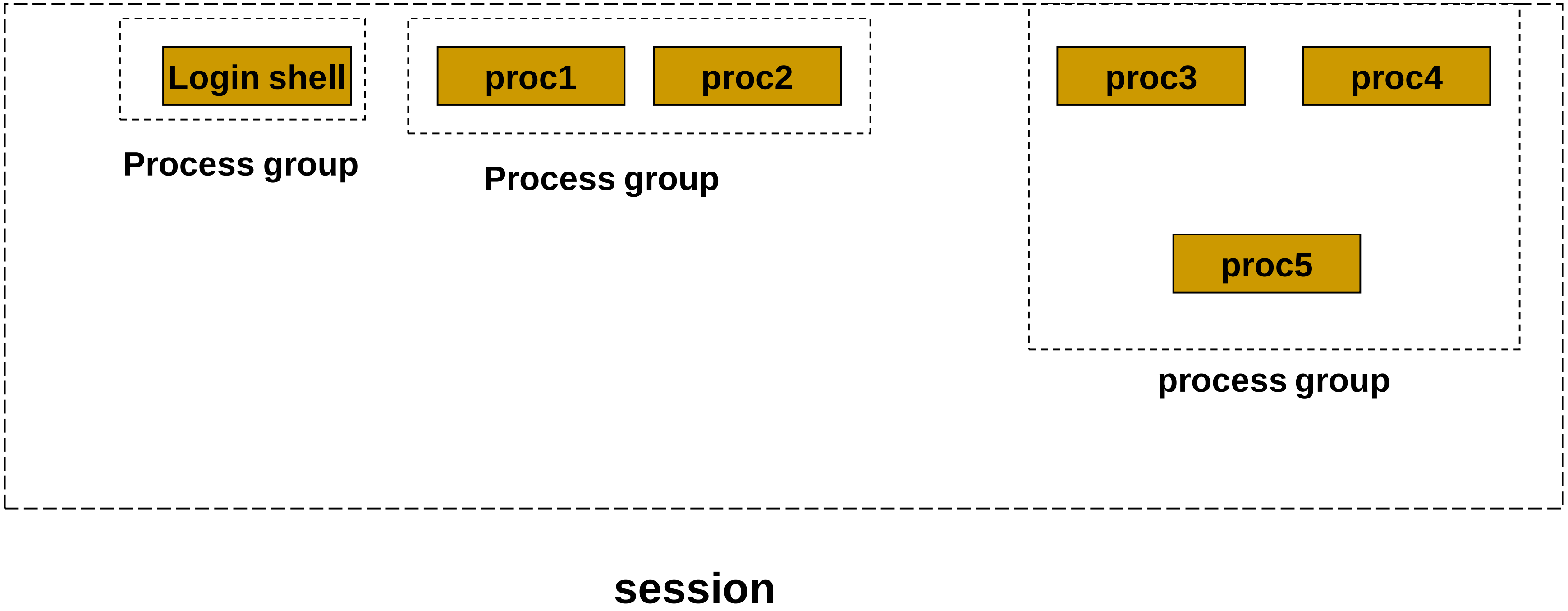
- **This sets the process group ID to pgid of the process pid. If the two arguments are equal, the process specified by pid becomes a process group leader.**

- **A process can set the process group ID of only itself or one of its children. If *pid* is 0, the process ID of the caller is used. Also if *pgid* is 0, the process ID specified by *pid* is used as the process group ID.**
- **In most job-control shells this function is called after a fork to have the parent set the process group ID of the child, and to have the child set its own process group ID.**

SESSIONS

- A **Session** is a **collection** of **one or more groups**.
- The processes in a process group are usually grouped together into a process group by a **shell pipeline**.
- A process establishes a new session by calling the ***setsid*** function.

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



Arrangement of processes into process groups and sessions

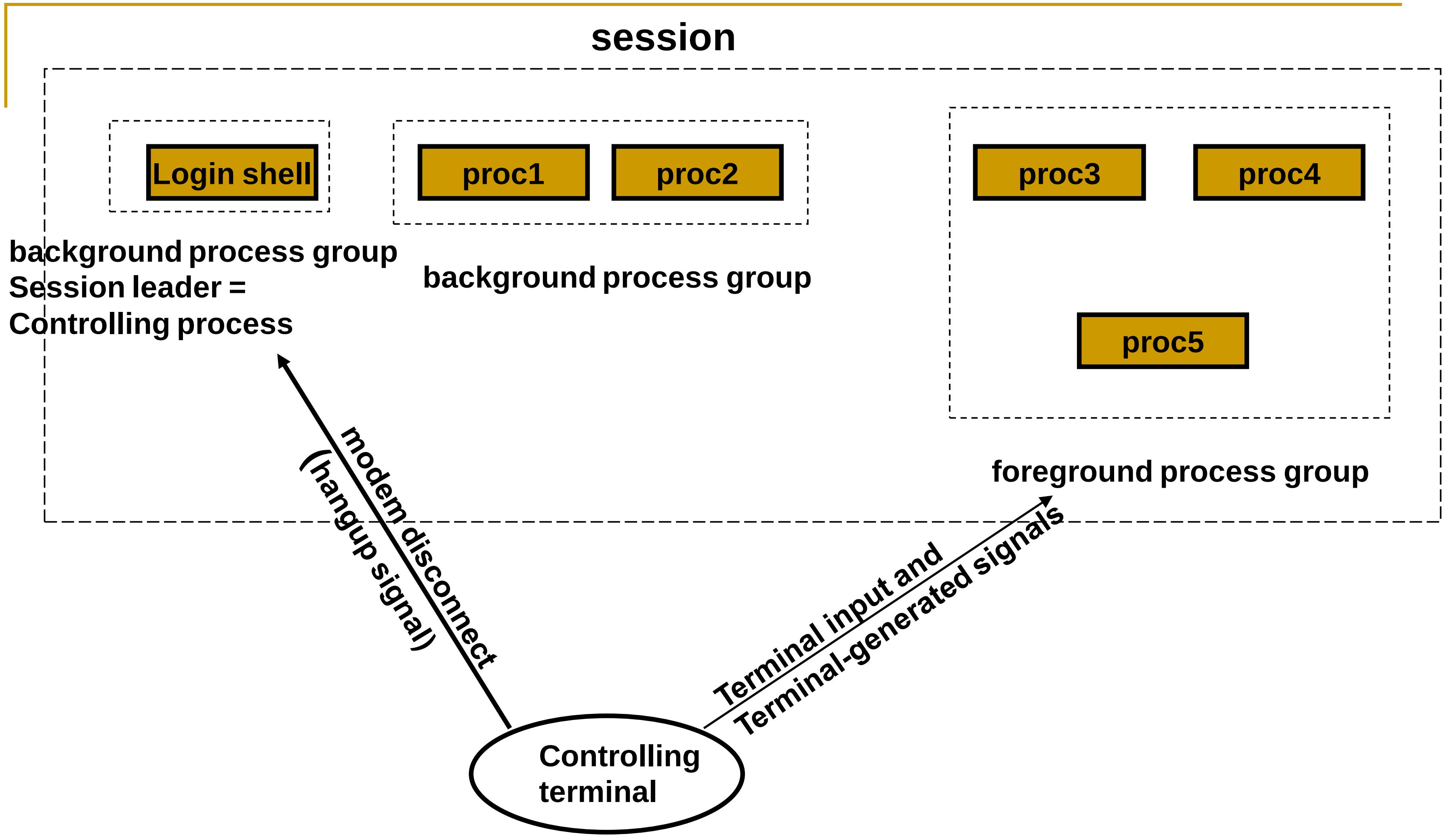
- If the calling process is not a process group leader, this function creates a new session. Three things happen:

1. The process becomes the session leader of this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal.

Controlling terminal

- **characteristics of sessions and process groups**
  - **A session can have a single controlling terminal.**
  - **The session leader that establishes the connection to the controlling terminal is called the controlling process.**
  - **The process groups within a session can be divided into a single foreground process group and one or more background process groups.**

- If a session has a controlling terminal, then it has a single foreground process group, and all other process groups in the session are background process groups.
- Whenever we type our terminal’s interrupt key or quit key this causes either the interrupt signal or the quit signal to be sent to all processes in the foreground process group.
- If a modem disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process



Process groups and sessions showing controlling terminal

tcgetpgrp and tcsetpgrp Functions

- **We need to tell the kernel which process group is the foreground process group, so that the terminal device driver knows where to send the terminal input and the terminal- generated signals**

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

- **The function *tcgetpgrp* returns the process group ID of the foreground process group associated with the terminal open on filedes.**
- **If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group ID to pgrpid..**

Job Control

- Why do we need job control?
- To allow us to start multiple jobs from a single terminal and control which jobs can access the terminal and which jobs are to be run in the background.
- It requires 3 forms of support:
  - A shell that supports job control.
  - The terminal driver in the kernel must support job control.
  - Support for certain job-control signals

- **A job is just a collection of processes, often a pipeline of processes.**
- **When we start a background job, the shell assigns it a job identifier and prints one or more process IDs.**
- $ make all > Make.out &

  [1] 1475

  $ pr *.c | lpr &

  [2] 1490

  $ just press RETURN

  [2] + Done pr *.c | lpr &

  [1] + Done make all > Make.out &

- **The reason why we have to press RETURN is to have the shell print its prompt. The shell doesn’t print the changed status of background jobs at any random time -- only right before it prints its prompt, to let us enter a new command line.**
- **Entering the suspend key (Ctrl + Z) causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group.**

The terminal driver really looks for 3 special characters, which generate signals to the foreground process group:

- **The interrupt character generates SIGINT**
- **The quit character generates SIGQUIT**
- **The suspend character generates SIGTSTP**

PROGRAM:

$cat temp.foo &	start in background, but It’ll read from standard input
[1] 1681	
$	we press RETURN
[1] + Stopped (tty input)	cat > temp.foo &
$ fg %1	bring job number 1 to foreground
cat > temp.foo	the shell tells us which job is now in the foreground
hello, world	enter one line
^D	type our end-of-file
$ cat temp.foo	check that the one line put into the file
hello, world	

- **What happens if a background job outputs to the controlling terminal?**

- **This option we can allow or disallow. Normally we use the stty(1) command to change this option.**

$ cat temp.foo &	execute in background
[1] 1719	
$ hello, world	the output from the background appears after the prompt we press return
[1] + Done	cat temp.foo &
$ stty tostop	disable ability of background jobs to output to controlling terminal
[1] 1721	
$	we press return and find the job is stopped
[1] + Stopped(tty output)	cat temp.foo &

Shell Execution Of Programs

- Bourne shell doesn't support job control
- ps –xj gives the following output

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	168	163	163	163	ps

- **Both the shell and the ps command are in the same session and foreground process group(163). The parent of the ps command is the shell.**
- **A process doesn't have a terminal process control group.**
- **A process belongs to a process group, and the process group belongs to a session. The session may or may not have a controlling terminal.**

- The foreground process group ID is an attribute of the terminal, not the process.
- If ps finds that the session does not have a controlling terminal, it prints -1.

If we execute the command in the background,

Ps –xj &

The only value that changes is the process ID.

ps –xj | cat1

PPID	PID	PGID	SID	TPGID	COMMAND
1	163	163	163	163	-sh
163	200	163	163	163	cat1
200	201	163	163	163	ps

- The last process in the pipeline is the child of the shell, and the first process in the pipeline is a child of the last process.

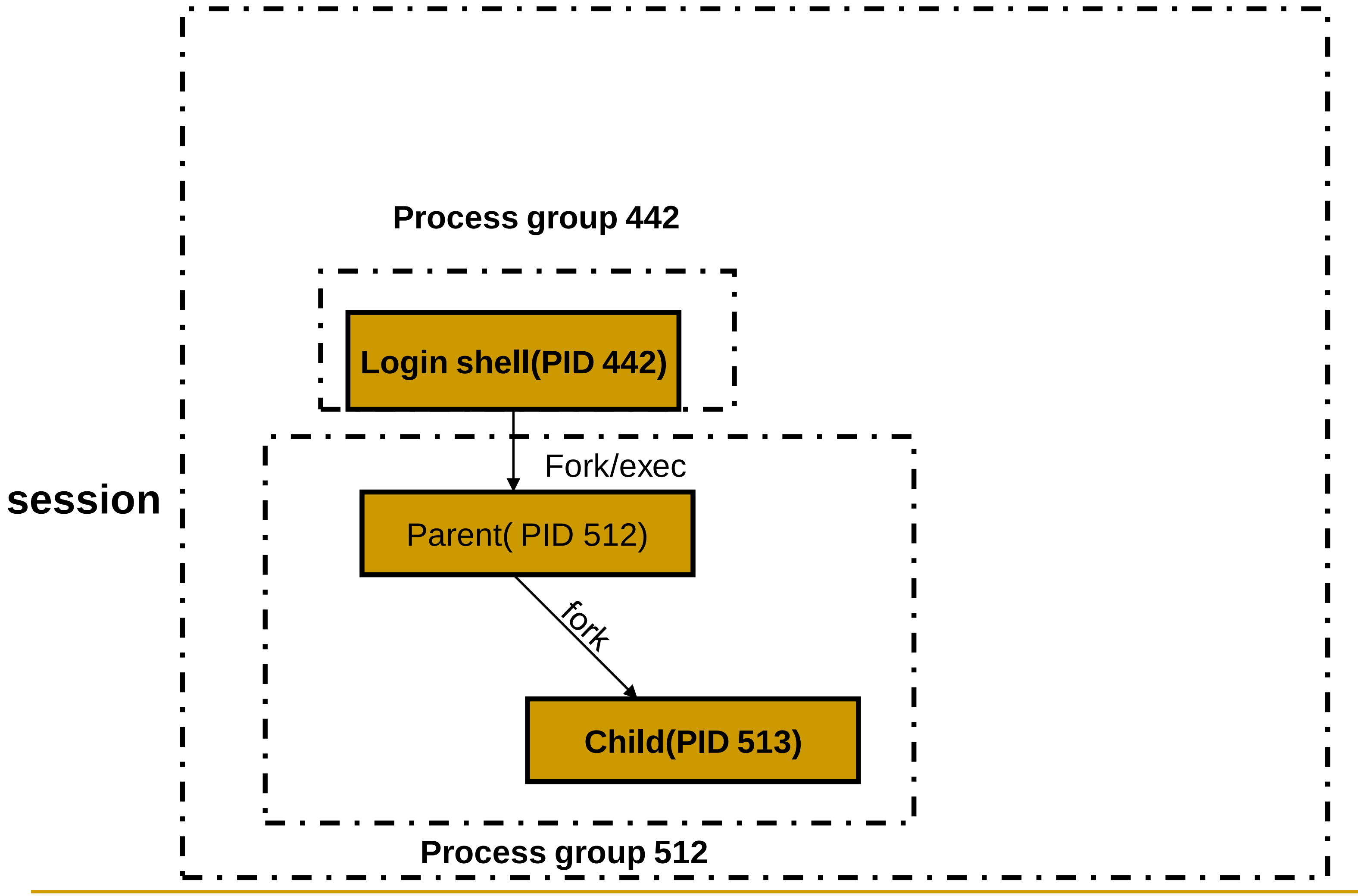
- If we execute the pipeline in the background

  ps –xj | cat1 &

- Only the process IDs change.
- Since the shell doesn't handle job control, the process group ID of the background processes remains 163, as does the terminal process group ID.

Orphaned process groups

- We know that a process whose parent terminates is called an orphan and is inherited by the init process.
- Sometimes the entire process groups can be orphaned.



session
Process group 442
Login shell(PID 442)
Fork/exec
Parent(PID 512)
fork
Child(PID 513)
Process group 512

- **This is a job-control shell. The shell places the foreground process in its own process group(512 in the example) and the shell stays in its own process group(442). The child inherits the process group of its parent(512). After the fork,**

- **The parent sleeps for 5 seconds. This is the (imperfect) way of letting the child execute before the parent terminates**
- **The child establishes a signal handler for the hang-up signal (SIGHUP). This is so we can see if SIGHUP is sent to the child.**

- **The child itself the stop signal(SIGTSTP) with the kill function.**

- **When the parent terminates, the child is orphaned, so the child's parent process ID becomes 1, the init process ID.**

- **At this point the child is now a member of an *orphaned process group*.**

- **Since the process group is orphaned when the parent terminates, it is required that every process in the newly orphaned process group that is stopped be sent the hang-up signal (SIGHUP) followed by the continue signal.**
- **This causes the child to be continued, after processing the hang-up signal. The default action for the hang-up signal is to terminate the process, which is why we have to provide a signal handler to catch the signal**

Creating an orphaned process group

```
#include   <sys/types.h>
#include   <errno.h>
#include   <fcntl.h>
#include   <signal.h>
#include   "ourhdr.h"

static void        sig_hup(int);
static void        pr_ids(char *);
```

```
int main(void)
{
   char      c;
   pid_t     pid;
    pr_ids("parent");
   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid > 0)
    {                                    /* parent */
     sleep(5);
      exit(0);
      }
```

```
else {                                  /* child */
        pr_ids("child");
        signal(SIGHUP, sig_hup);
        /* establish signal handler */
    kill (getpid(), SIGTSTP);
        pr_ids("child");
      /* this prints only if we're continued */
        if (read(0, &c, 1) != 1)
        printf ("read error from control
                terminal,errno = %d\n", errno);
        exit(0);
    }
}
```

```
static void    sig_hup (int signo)
 {
  printf("SIGHUP received, pid = %d\n",
                                          getpid());
  return;
 }
static void  pr_ids (char *name)
 {
  printf("%s: pid = %d, ppid = %d, pgrp =
 d\n", name, getpid(), getppid(), getpgrp());
  fflush(stdout);
 }
```

```
/* OUTPUT
$ a.out
Parent: pid = 512, ppid=442, pgrp = 512
Child: parent = 513, ppid = 512, pgrp = 512
$ SIGHUP received, pid = 513
Child: pid = 513 , ppid = 1, pgrp = 512
Read error from control terminal, errno = 5
*/
```

- **The parent process ID of the child has become 1.**
- **After calling pr_ids in the child, the program tries to read from standard input. When the background process group tries to read from its controlling terminal, SIGTTIN is generated from the background process group.**
- **The child becomes the background process group when the parent terminates, since the parent was executed as a foreground job by the shell**

- **The parent process ID of the child has become 1.**
- **After calling pr_ids in the child, the program tries to read from standard input. When the background process group tries to read from its controlling terminal, SIGTTIN is generated from the background process group.**
- **The child becomes the background process group when the parent terminates, since the parent was executed as a foreground job by the shell**

Questions

- Explain briefly the memory layout of a C program (10)
- What is fork and vfork ? Explain with an example for each (8)
- What is a zombie process ? Write a program in C/C++ to avoid zombie process by forking twice (6)
- What is job control ? Summarize the job control features with the help of a figure (10)

- Explain the different exec functions. Explain how their functioning differ from each other . Write a program that execs an interpreter file (10)
- What is job control ? What support is required for job control ? Explain with an example (10)
- Explain how accounting is done in UNIX system. Write a program to generate accounting data and give its process structure (10)

- What is a controlling terminal ? Explain its characteristics and relation to session and process groups (10)
- With an example explain the use of setjmp and longjmp functions (10)
- What is race condition ? Write a program to demonstrate the race condition (10)
- With a neat block diagram, explain how a C program is started and the various ways it can terminate. Give the prototypes for exit and _exit functions and explain their difference (10)