Process

Suntae Hwang
Kookmin University

Process Control

- creation of new processes
- executing programs
- process termination

Process Identifiers

- 모든 프로세스는 음이 아닌 정수로 된 ID를 갖는다.

- 관련된 함수
  - getuid, geteuid, getgid, getegid, getpid, getppid 등
  - man getuid
  - man getpid 를 실행해 볼 것

특별한 프로세스

- Process ID 0 : scheduler process
  - *swapper*로 알려져 있다. 디스크 상에 올려져 있는 것이 아니라 kernel의 일부분이다. System process로 알려져 있다.
- Process ID 1 : **init** process
  - 부팅이 끝나면 kernel이 호출한다. (보통 /etc/init 이나 /sbin/init 의 file이다.) init는 각 시스템별 초기화 화일(/etc/rc* files)을 읽어서 시스템을 원하는 상태로 전환 한다.(예: multiuser).
  - init은 결코 죽지 않으며 superuser privilege로 실행 되지만 보통의 user process이다. (unlike swapper)
- Process ID *2* : *pagedaemon*
  - virtual memory system의 paging을 지원한다. It is a kernel process like the swapper.

fork Function

- 앞에서 언급한 3개의 특수 프로세스를 제외한 모든 프로세스를 생성하는 유일한 방법은 fork 함수를 이용하는 것이다.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
 Returns: 0 in child, process ID of child in parent,
 -1 on error
```

fork Function(cont)

- 새로 생성된 process를 child process라 한다.
- 한번 호출되면 두번 return한다.

- 한 프로세스가 여러개의 child를 가질 수 있으므로 child process의 ID를 return한다. 그래서 자신의 child process의 ID를 알기 위한 함수는 없다.

- process는 오직 한 개의 parent process를 갖는다. 따라서 fork는 child에게 0을 반환 한다. child는 getppid를 통해서 언제든지 parent process ID를 알 수 있다.

- child와 parent 둘다 fork 이후에 나오는 명령을 실행한다.

fork Function(cont)

- child는 parent의 copy이다. (data space, heap, stack) - 이 copy는 child를 위한 것이므로 parent는 이를 access할 수 없다. 즉 child와 parent는 data space, heap, stack등을 share하지 않는다. (except read-only text segment)

- **COW(copy-on-write)**

  위의 region들(data space, heap, stack)이 share 되고 kernel에 의해 read-only로 바뀌어 보호된다. parent와 child 중 어느 하나가 이 region을 수정하려고 하면 kernel은 그 부분의 memory만을 copy하여 준다. 통상 page 단위 이다.

proc/fork1

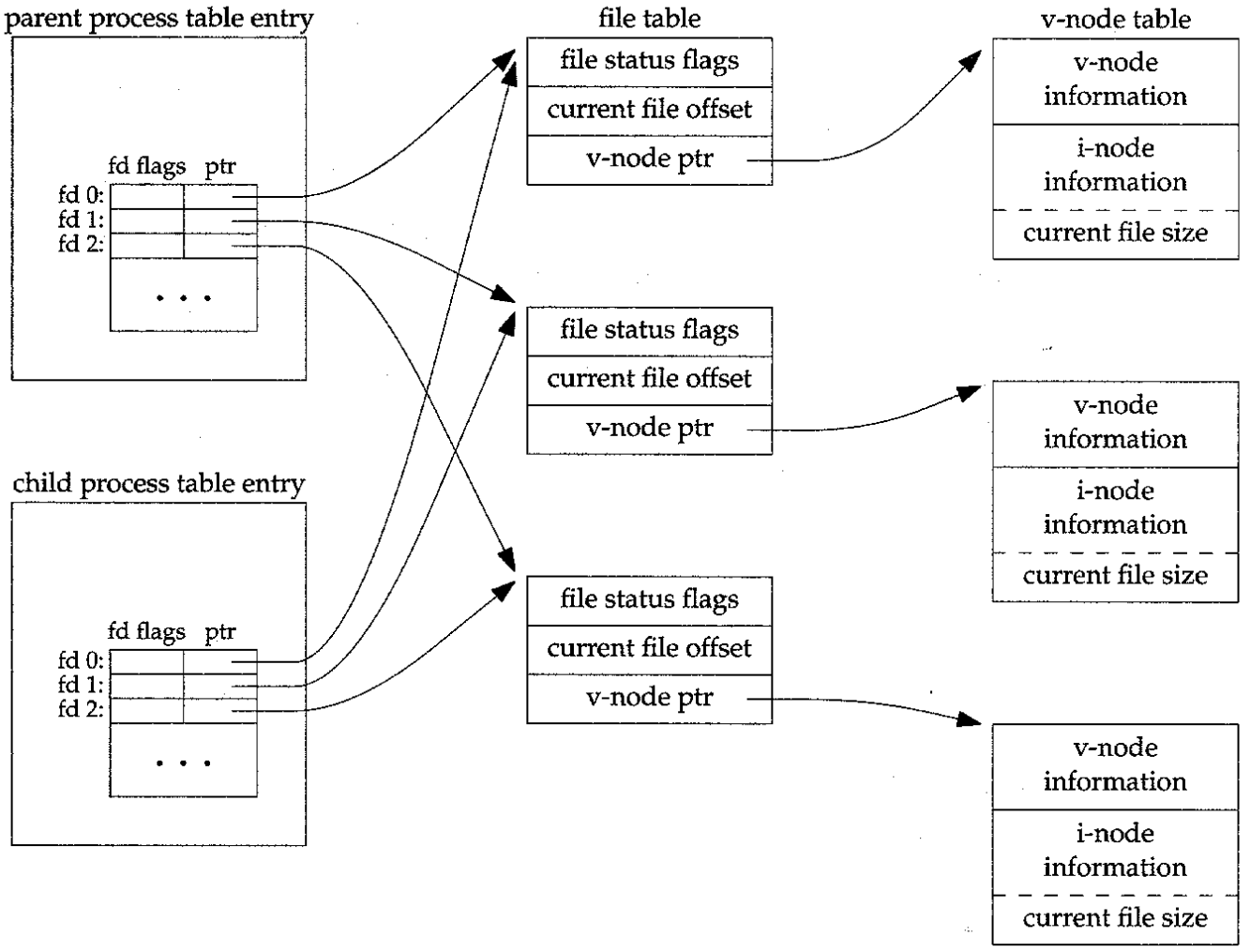
```
#include    <sys/types.h>
#include    “ourhdr.h”
int         glob = 6;  /* external variable in initialized data */
char buf[] = “a write to stdout\n”;
int main(void)
{ int       var;           /* automatic variable on the stack */
  pid_t     pid;
  var = 88;
  if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
     err_sys("write error");
  printf("before fork\n"); /* we don't flush stdout */
  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);
}
```

```
cs:~/advprog/apue-sun/proc$ fork1
a write to stdout
before fork
pid = 17069, glob = 7, var = 89
pid = 17068, glob = 6, var = 88
cs:~/advprog/apue-sun/proc$ fork1>fork1.out
cs:~/advprog/apue-sun/proc$ cat fork1.out
a write to stdout
before fork
pid = 17071, glob = 7, var = 89
before fork
pid = 17070, glob = 6, var = 88
cs:~/advprog/apue-sun/proc$
```

proc/fork1

- child와 parent 중 어느 것이 먼저 실행 되는 지는 정해져 있지 않다.
- stdout은 terminal에 연결 된 경우는 lined buffered이고 file로 연결된 경우에는 fully buffered된 것이다.
- fork1.out에서 “before fork\n”은 buffer에 남아 있는데 (fully buffered 이므로 아직 flush out 되지 않음) 이것이 fork에 의해 child의 address space에 copy가 됨. 두번째 printf는 이 buffer에 추가되었다가 exit()에 의해 마지막으로 flush out 됨

File Sharing



File Sharing(cont)

- parent에서 redirect된 stdout은 child에서도 redirect된다는 것이다.
  - 사실 parent에서 open된 모든 descriptor들은 child에 그대로 복제된다.
  - parent와 child가 같은 file offset을 공유한다는 것은 중요하다

File Sharing(cont)

- parent와 child가 synchronization 없이 같은 descriptor에 쓰는 것은 출력이 서로 섞이게 한다.
  - 이것은 정상이 아니다.

- 다음과 같이 fork 이후에 descriptor를 다루는 정상적인 방법 두 가지가 있다.
  - parent는 child가 끝나기를 기다린다.
  - parent와 child는 각각 제갈 길을 간다. Fork 이후 각 process는 필요 없는 descriptor를 close하여 상대방의 open된 descriptor에 간섭하지 않는다. Network server에서 자주 쓰인다.

fork의 두가지 용법

- process가 그 자신을 복제하고 싶을 때.
  - process와 child는 같은 code의 다른 부분을 실행.
  - 예: network server, parent는 client의 service request를 기다리고 child는 이를 처리한다.

- process가 다른 program을 실행하고 싶을 때.
  - 예: shell

vfork Function

- 새 프로세스가 다른 프로그램을 exec 하기 위한 것일 때 사용.

- parent의 address space를 copy하지 않고 새로운 process를 생성,
  - 그러나 exec나 exit이 실행 되기 전까지 child는 parent의 address space에서 실행 된다.

- vfork는 child가 먼저 실행 됨을 보장하며 child가 exec나 exit을 호출하면 parent가 실행 된다.

proc/vfork1.c

```
#include   <sys/types.h>
#include   "ourhdr.h"
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");     /* we don't flush stdio */
    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);
       }
```

```
cs:~/advprog/apue-sun/proc$ vfork1
before vfork
pid = 17195, glob = 7, var = 89
cs:~/advprog/apue-sun/proc$
```

- 만약에 _exit() 대신 exit()을 사용하면?

exit Function

- Normal termination
  - Executing a return from the main function.(equivalent to calling exit)
  - Calling the exit function.atext에의 등록된 function들 수행, close all standard I/O streams.
  - Calling the _exit function, handles the UNIX-specific details

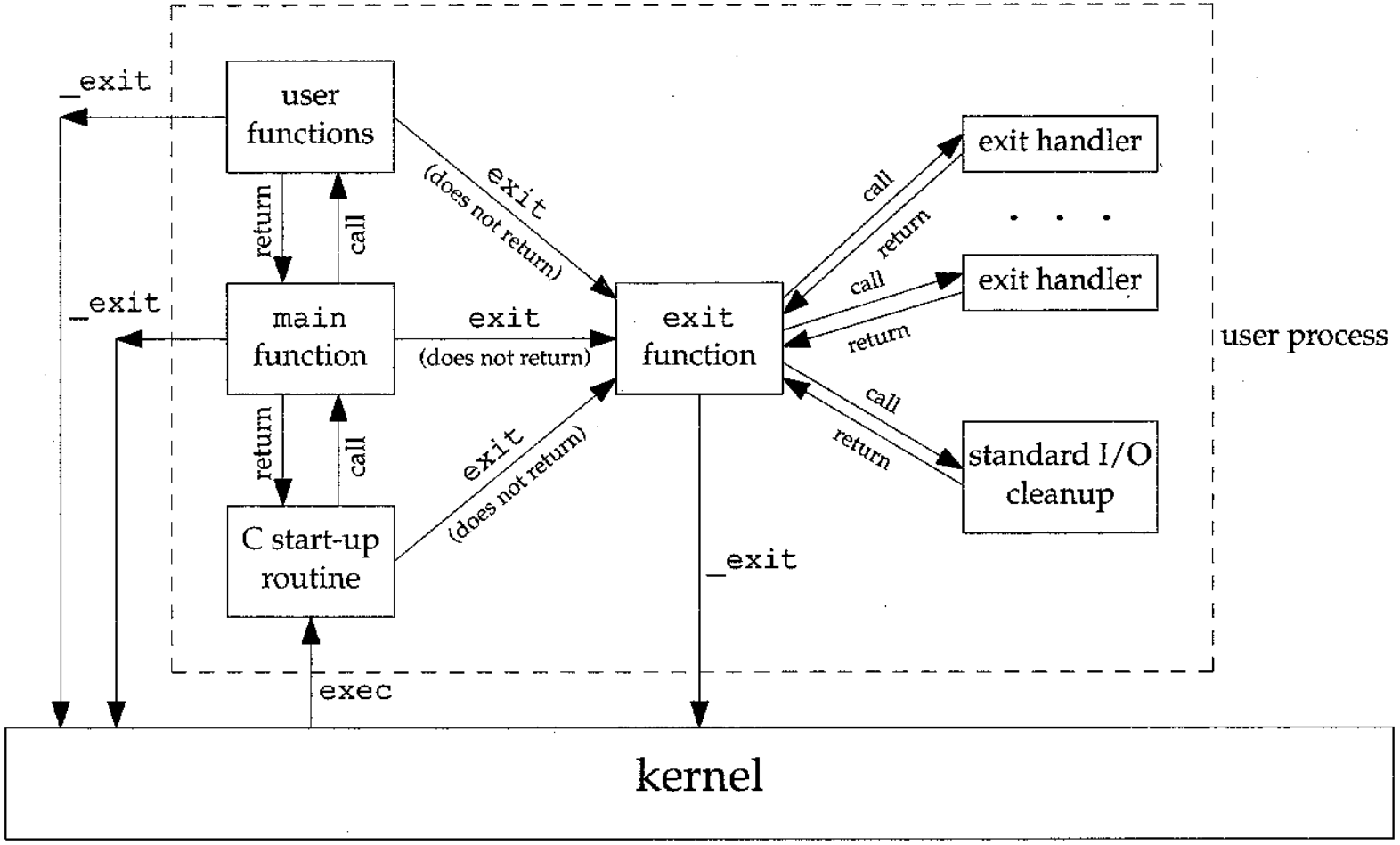
- Abnormal termination
  - Calling abort()
  - When the process receives certain signal which can be generated by the process itself, by some other process, or by the kernel.

exit Function(cont)

- process가 어떻게 끝났 건, kernel안의 동일한 code가 실행 된다.
  - 즉 close all the descriptors for the process, release the memory that it was using, ...

- exit status vs termination status
  - exit status : exit이나 _exit의 argument이거나 main으로부터 반환된 값
  - child가 정상적으로 terminate 되면 parent는 그의 exit status를 알 수 있다.

Exit


_exit
user functions
exit (does not return)
exit handler
call
return
. . .
exit handler
_exit
main function
exit (does not return)
exit function
user process
return
call
call
return
standard I/O cleanup
return
call
exit (does not return)
C start-up routine
_exit
exec
kernel

exit Function(cont)

- 만약에 parent가 child보다 먼저 끝나면?
  - init가 그 child process를 상속 받는다. 즉 어떤 process가 끝날 때 마다 모든 active process를 확인 하여 child process가 아직 존재하면 그 것의 parent process ID를 1로 바꾼다.

- child가 parent 보다 먼저 끝나면?
  - parent가 child가 끝났는지 check하려고 하는데 그전에 이미 child가 사라져 버리면?
  - kernel이 모든 terminating process에 대한 어느 정도의 정보를 가지고 있다가 parent가 wait, waitpid를 실행할 때 활용한다.
  - 이 정보는 최소한 process ID, termination status, CPU time taken by the process등을 포함한다.

Zombie

- 자신이 끝났을 때 parent가 아직 기다리고 있지 않으면 그 process를 zombie 라고 한다.

- init 의 child는 zombie가 되지 않는다. 왜냐하면 init은 그의 child가 끝날 때 마다 wait function을 호출하여 그 termination status를 fetch하기 때문이다.

- init의 child는 자신이 생성 했거나 상속 받은 것이다.

wait and waitpid Function

- The process that calls wait or waitpid can
  - block(if all of its children are still running) or
  - return immediately with the termination status of a child(if the child is a zombie)
  - return immediately with an error(if it doesn't have any child processes)

- The differences between wait and waitpid
  - wait can block the caller until a child process terminates, while waitpid has an option that prevent it from blocking
  - waitpid doesn't wait for the first child to terminate - it has a number of options that control which process it waits for.

lib.svr4/prexit.c

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
void
pr_exit(int status)
{
  if (WIFEXITED(status))
     printf("normal termination, exit status = %d\n",
            WEXITSTATUS(status));
  else if (WIFSIGNALED(status))
     printf("abnormal termination, signal number = %d%s\n",
            WTERMSIG(status),
#ifdef WCOREDUMP
            WCOREDUMP(status) ? " (core file generated)" : "");
#else
            "");
#endif
  else if (WIFSTOPPED(status))
     printf("child stopped, signal number = %d\n",
            WSTOPSIG(status));
}
```

proc/wait1.c

```
#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);
}
```

```
cs:~/advprog/apue-sun/proc$ wait1
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
cs:~/advprog/apue-sun/proc$
```

Example

- child가 끝나는 것을 기다리지도 않고 child가 zombie가 되는 것도 원치 않을 때

proc/fork2.c

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
int main(void)
{ pid_t     pid;
  if ( (pid = fork()) < 0)
     err_sys("fork error");
  else if (pid == 0) {                    /* first child */
     if ( (pid = fork()) < 0)
            err_sys("fork error");
     else if (pid > 0)
            exit(0); /* parent from second fork == first child */
     /* We're the second child; our parent becomes init as soon
        as our real parent calls exit() in the statement above.
        Here's where we'd continue executing, knowing that when
        we're done, init will reap our status. */
     sleep(2);
     printf("second child, parent pid = %d\n", getppid());
     exit(0);
  }
```

```
  if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
     err_sys("waitpid error");
 /* We're the parent (the original process); we continue executing,
     knowing that we're not the parent of the second child. */
  exit(0);
}
```

cs:~/advprog/apue-sun/proc$ fork2

cs:~/advprog/apue-sun/proc$ second child, parent pid = 1

- 위의 프로그램이 실행 되는 상황을 그림으로 표현 해보시오

Race Condition

- 여러 개의 프로세스가 공유하는 데이터를 가지고 어떤 일을 하는데 그 결과가 프로세스들이 실행된 순서에 따라서 바뀌면 race condition이 발생한 것이다.

- 앞의 fork2.c에서 두번째 child가 첫번째 child보다 먼저 수행되면 두번째 자식의 parent는 init으로 inherited 되지 않고 그대로 첫번째 child가 된다.

- 이 때 sleep(2)의 호출은 두번째 child가 첫번째 child 보다 나중에 실행된다는 것을 보장해 줄 수 있는 방법이 아니다.

Race Condition(cont)

- 따라서 parent가 끝나기를 기다리려면 다음과 같은 polling 하는 code를 삽입한다

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

- 이런 polling은 CPU의 낭비이다.

proc/tellwait1.c

```
#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);
}
```

```
sizzle:~/advprog/apue-sun/proc$ tellwait1
output from child
output from parent
sizzle:~/advprog/apue-sun/proc$ tellwait1
oouuttppuutt  ffrroomm  cphairlendt
```

proc/tellwait2.c

```
#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);
}
static void charatatime(char *str)
 . . .
```

```
sizzle:~/advprog/apue-sun/proc$ tellwait2
output from parent
output from child
```

Process-id

- Non-negative number
- Is unique at any one time
- Special processes
  - Process 0: scheduler or swapper
  - Process 1: UNIX의 모든 user process의 조상
    /etc/init program의 실행 상태
  - Process 2: pagedaemon
- 관련 system call
  - getpid() : 자신의 Process-id를 얻는다
  - getppid() : 자신의 parent의 Process-id를 얻는다.

Process group and process group-ids

- UNIX process는 group에 속하게 된다.
- 전형적으로 shell의 command line에서 pipe로 연결된 process들은 한 group에 속한다.
  예: $ who | tee w.out | grep jenny
- Process에 signal이 보내질 수 있는데 이 signal이 group 전체에 보내질 수도 있다.
- System call getpgrp() : process group-id 값을 얻는다.
- Process group-id 값은 fork, exec를 통해 inherit된다.
- Leader of process group : process-id와 process group-id가 같은 process

Changing process group

- Job control allows a shell
  - to start multiple process groups
  - And to control which of the process groups should run in the foreground (and therefore have access to the terminal) and which should run in the background.
- system call setpgid

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

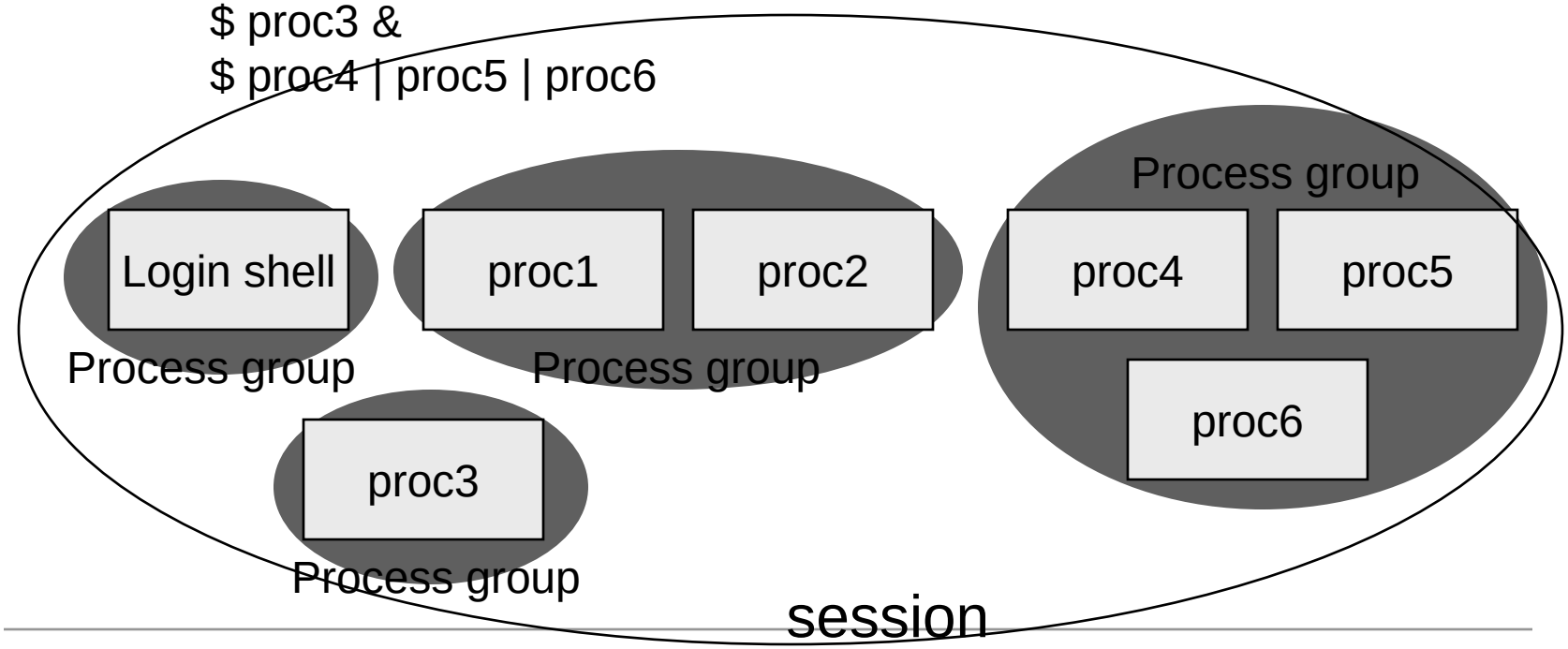
process-id가 pid인 process의 process group-id를 pgid로 설정한다.

  - If pid is 0 : 호출 process의 process-id를 이용
  - If pid == pgid : process는 process group leader가 된다.
  - If pgid is 0 : pid 를 process group-id로 이용
  - on error : `-1` is returned

Sessions and session-ids

- 각 process group은 session에 속한다.
- A session is really about a process's connection to a **controlling terminal**.
- 전형적인 session의 예

```
$ proc1 | proc 2 &
$ proc3 &
$ proc4 | proc5 | proc6
```



Sessions and session-ids (cont)

- System call getsid

  pid_t getsid(pid_t pid);

  - Session-id를 얻는다
  - pid가 0이면 호출 process의 session-id를 return

- System call setsid

  pid_t setsid(void);

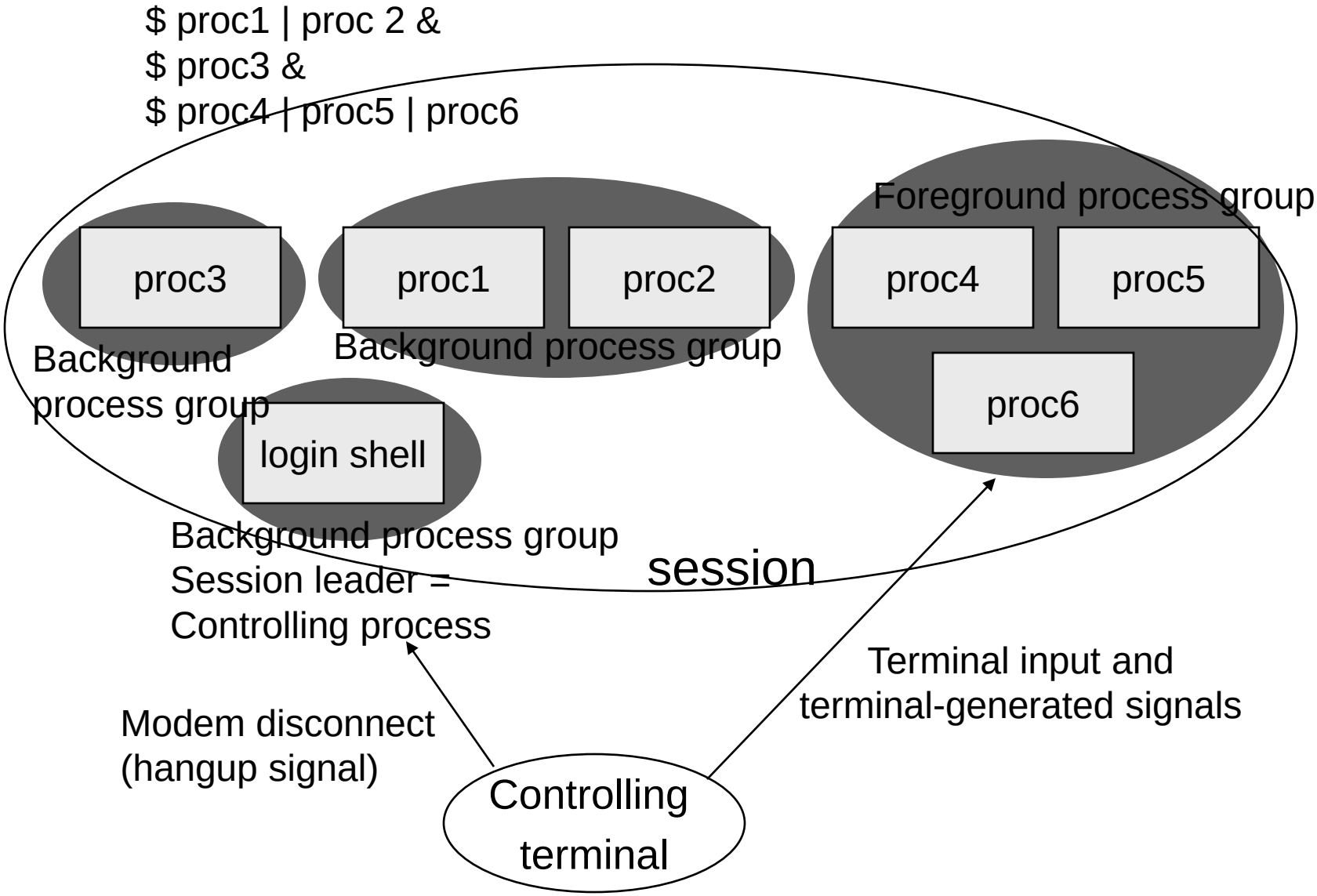
  calling process가 process group leader가 아니면 그 process는
  - 새로운 session의 leader가 된다.
  - 새로운 process group의 leader가 된다.
  - Controlling terminal을 갖지 않는다.

  calling process가 process group leader이면
  - Error : -1을 return 한다
  - 이를 방지하기 위해서 fork한 뒤에 parent는 종료하고 child에서 실행한다.

Controlling terminal

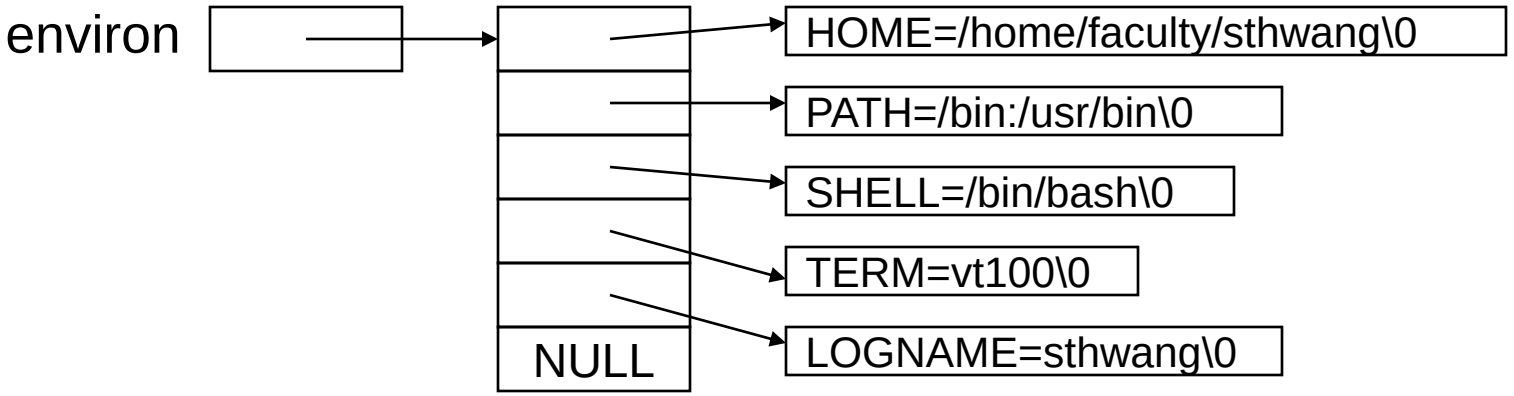
- There are few other characteristics of session and process group
  - A session can have a single *controlling terminal* on which we log in.
  - *Controlling process*: the session leader that establishes the connection to the controlling terminal
  - Process groups within a session
    - A single foreground process group
    - One or more background process group
  - If a session has a controlling terminal, then it has a single foreground process group
    - All other process groups in the session are background process group
  - Whenever we type our terminal's interrupt key (DEL or ^C) or quit key (^\) this causes signal (either interrupt or quit) 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(the session leader).

Controlling terminal (cont)



The environment

- Simply a collection of null-terminated strings
- Each environment string has the form
  name=something
- Program에서는 null-terminated array of character pointers로 표현



The environment (cont)

- 자신의 환경을 출력하고 exit하는 프로그램

```
/* showmyenv.c */
main(int argc, char **argv, char **envp)
{
  while (*envp)
        printf("%s\n", *envp++);
}
```

- 새로운 환경을 명세하기 위해서

```
execle(path, arg0, arg1, ..argn, (char *)0, envp);
execve(path, argv, envp);
```

- main 함수의 parameter대신에 global 변수로 access

```
extern char **environ;
```

- Scan or change environment pointed to by environ

```
eg: printf("PATH=%s\n",getenv("PATH"));
    putenv("TERM=ansi");
```

User- and group-ids

- Real user-id and real group-id
  - 해당 프로세스를 호출한 사용자와 그 사용자가 현재 속한 그룹
- Effective user-id and effective group-id
  - 어떤 사용자가 한 파일에 접근 가능한지의 여부를 결정
  - 해당 프로세스나 그의 조상 중 하나가 set-user-id나 set-group-id permission bit를 1로 함으로써 effective user- and group-ids를 변경할 수 있다.
    - 예: 프로그램 파일의 set-user-id bit가 1이면, 그 프로그램이 exec로 호출될 때, 그 프로세스의 effective user-id는 그 프로세스를 시작 시킨 user가 아니라 프로그램 파일의 owner의 user-id가 된다.
- Non-privileged user (not superuser)가 호출한 프로세스는 자신의 effective user- and group-ids를 real user- and group-ids로 재설정할 수 있는 권한 밖에는 없다.

User- and group-ids (cont)

- Three kinds of User- and group ids

Real user- and group-ids	Who we really are
Effective user- and group-ids Supplementary group-ids	Used for file access permission checks
Saved set-user- and set-group-ids	Saved by exec functions

- Different ways to change the three user-ids

id	exec		setuid(*uid*)	
	set-user-id bit off	set-user-id bit on	superuser	Non-privileged user
Real user-id	unchanged	unchanged	Set to *uid*	unchanged
Effective user-id	unchanged	Set from user-id of program file	Set to *uid*	Set to *uid* *uid* is either ruid or saved set-user-id
Saved set-user-id	Copied from effective user-id	Copied from effective user-id	Set to *uid*	unchanged

Miscellaneous

- The current working directory

  int chdir(const char *path)

- The current root directory

  int chroot(const char *path)

- File size limits: ulimit

  long ulimit(int cmd, [long newlimit]);

  – 프로세스 별로 write system call을 이용하여 만들 수 있는 파일의 크기에 제한이 있다.

  – cmd를 UL_GETFSIZE로 지정하여 현재 파일의 크기 제한을 알 수 있다.

  – Superuser는 파일 크기의 제한을 바꿀 수 있다.

```
if (ulimit(UL_SETFSIZE, newlimit) < 0)
    perror("ulimit failed");
```

Miscellaneous (cont)

- Process priority: nice
  - 시스템이 특정 프로세스에게 할당할 CPU time의 비율을 정할 때 부분적으로 nice 값에 기반
  - Nice 값은 0 부터 시스템이 정하는 최대값까지
  - 큰 값일 수록 낮은 우선 순위
  - Superuser만이 음수의 nice 값을 세팅할 수 있다.