CHUNGNAM NATIONAL UNIVERSITY

유닉스 프로그래밍

강의 9. 프로세스 II

교재 8.4

Nov. 2, 2010

http://eslab.cnu.ac.kr

좀비의 교훈

- 프로세스가 종료하는 경우 커널이 시스템에서 즉시 제거하지 않는다
- 종료된 프로세스는 자신의 부모가 청소할 때까지 종료된 상태로 남아있게 된다
- 부모가 정상적으로 자식을 청소하지 않으면, 커널은 init 프로세스(PID = 1) 로 하여금 대신 청소하도록 해 준다

- 프로세스를 명시적으로(explicit)하게 청소하는 함수
  - **wait** 함수를 이용해서 자식을 청소할 수 있다

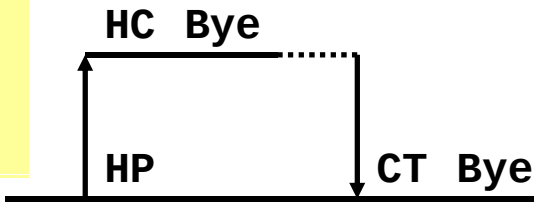
wait: 자식과 동기하기

- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the `pid` of the child process that terminated
  - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated
  - then, kernel removes child process
  - This is the explicit reaping of child process by the parent

wait: 자식과 동기하기

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Wait() Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

Returns true if normal exit

Returns exit status if normal exit

Waitpid(): 특정 pid를 청소한다

- **waitpid(pid, &status, options)**
  - wait for specific process id
  - Options : WNOHANG(check once), WUNTRACED(wait for the child)

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```

Wait/Waitpid Example Outputs

Using **wait** (**fork10**)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using **waitpid** (**fork11**)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

프로세스를 sleep 시키기

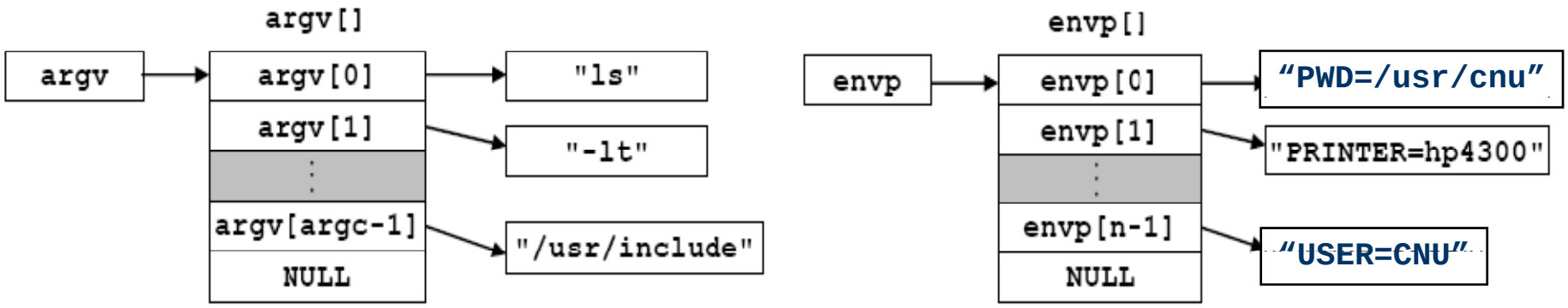
- `unsigned int sleep(unsigned int secs)`
  - **suspends a process for some period of time**
  - **return 0 when normal wake and remaining secs otherwise**
- `int pause(void)`
  - **puts calling function to sleep until a signal is received**

exec: 새 프로그램을 실행하기(1)

- `int execl(char *path, char *arg0, char *arg1, …, 0)`
  - **loads and runs executable at `path` with args `arg0`, `arg1`, …**
    - `path` is the complete path of an executable
    - `arg0` becomes the name of the process
      - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
    - "real" arguments to the executable start with `arg1`, etc.
    - list of args is terminated by a `(char *)0` argument
  - **returns -1 if error, otherwise doesn't return!**

새 프로그램을 실행하기(2)

- int execve (char *path, char *argv[], char *envp[])
  - argv, envp: null terminated pointer arrays
  - environ: global variable pointing to program's environment strings



새 프로그램을 실행하기(3)

- Example : running /bin/ls

```
main() {
    if (fork() == 0) {
        execl("/bin/ls", "ls", "/", 0);
    }
    wait(NULL);
    printf("completed\n");
    exit();
}
```

```
main() {
    char *args[] = {"ls", "/", NULL};
    if (fork() == 0) {
        execv("/bin/ls", args);
    }
    wait(NULL);
}
```

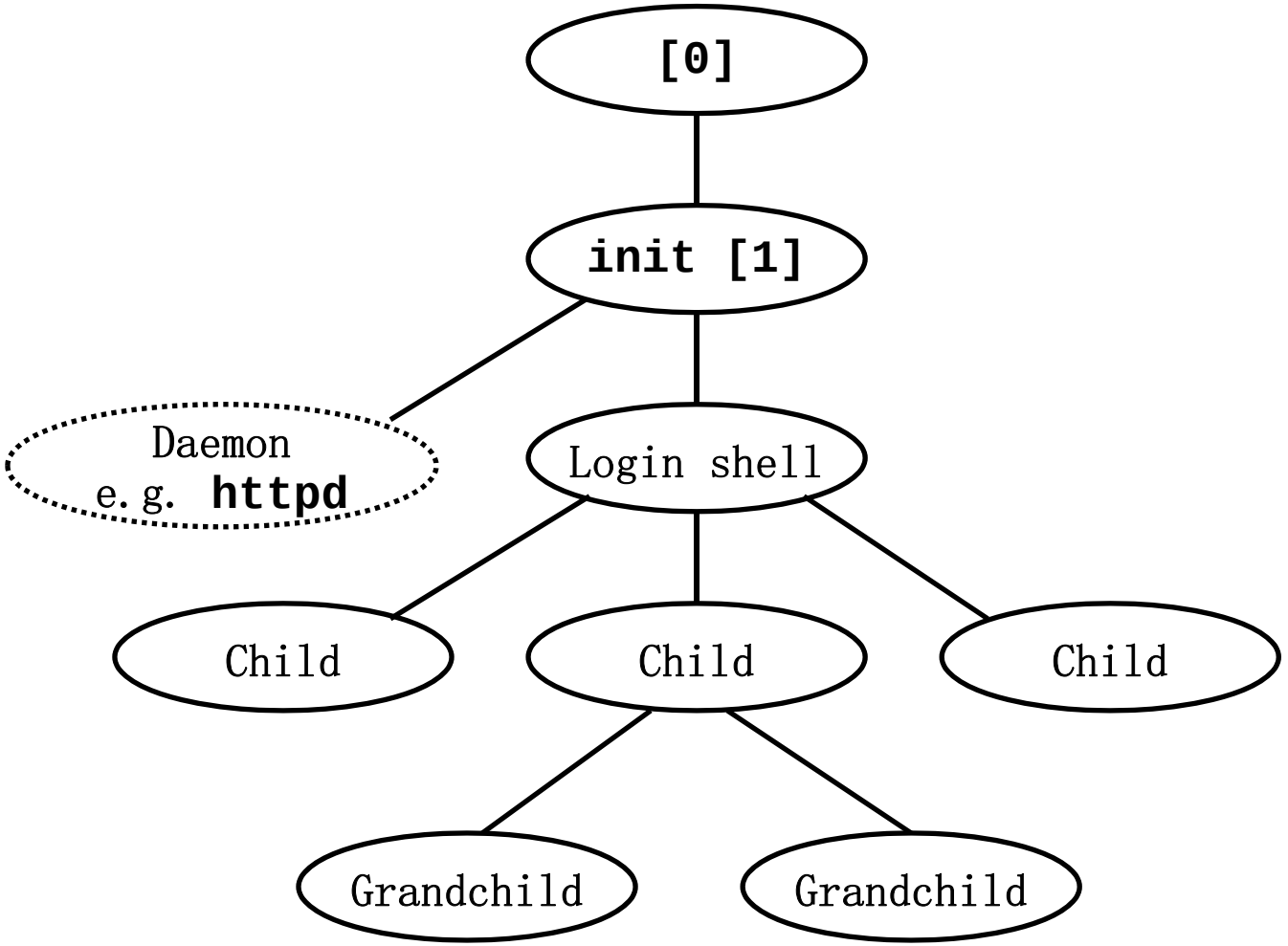
멀티 태스킹의 세계

- 시스템은 다수의 프로세스를 동시에 실행한다
  - **Process: executing program**
    - State consists of memory image + register values + program counter
  - **Continually switches from one process to another**
    - Suspend process when it needs I/O resource or timer event occurs
    - Resume process when I/O available or given scheduling priority
  - **Appears to user(s) as if all processes executing simultaneously**
    - Even though most systems can only execute one process at a time
    - Except possibly with lower performance than if running alone

멀티 태스킹에 대한 프로그래머의 관점

- 기본 함수
  - **`fork()` spawns new process**
    - Called once, returns twice
  - **`exit()` terminates own process**
    - Called once, never returns
    - Puts it into “zombie” status
  - **`wait()` and `waitpid()` wait for and reap terminated children**
  - **`execl()` and `execve()` run a new program in an existing process**
    - Called once, (normally) never returns
- 프로그램의 어려움
  - **Understanding the nonstandard semantics of the functions**
  - **Avoiding improper use of system resources**
    - E.g. “Fork bombs” can disable a system.

Unix 프로세스 체계


[0]
init [1]
Daemon
e.g. httpd
Login shell
Child
Child
Child
Grandchild
Grandchild

작업제어(1)

- 프로세스 상태 : ps –[aefglu]
  - -a all (단말기에서 제어하는)
  - -e everything (단말기에서 제어하지 않는 것도)
  - -f full listing, -g group
  - -l long, -u user
  - 예: $ps –ef

```
UID        PID  PPID  C STIME TTY          TIME CMD
604      24019 21917  0 17:28 pts/20   00:00:00 emacs codon2amino.pl
595      31328 30722  0 20:49 pts/10   00:00:00 vim wwiz
540      31945 31037  0 21:32 pts/12   00:00:00 vim wwiz.pl
501       1007 31192  0 22:27 pts/8    00:00:00 script
501       1008  1007  0 22:27 pts/8    00:00:00 script
506       1283  1151  0 22:35 pts/16   00:00:00 vim homewk
506       1677   490  0 22:49 pts/5    00:00:00 emacs codon2amino.pl
560       1866  1824  0 22:58 pts/1    00:00:00 vim wwiz
519       1892 32153  2 23:00 pts/13   00:00:00 emacs wwiz.pl
501       1894  1009  0 23:00 pts/14   00:00:00 ps -af
```

작업제어(2)

- ps 헤더(header)
  - F: flags(현재는 별 의미 없음)
  - S: state(O: running, S: sleeping, R: runnable, Z: zombie, T: stopped)
  - UID: user ID, PID: process ID, PPID: parent ID
  - C: central processor utilization
  - STIME: starting time(24시간 지나면 월일로)
  - TTY: terminal 번호, TIME: 실행된 분과 초, CMD: command
- sleep seconds
  - 명시된 초동안 잠든 후 프로세스 종료
  - 예: $ (sleep 10; echo done) &

The ps 명령어

Unix> ps aux -w --forest (output edited to fit slide)

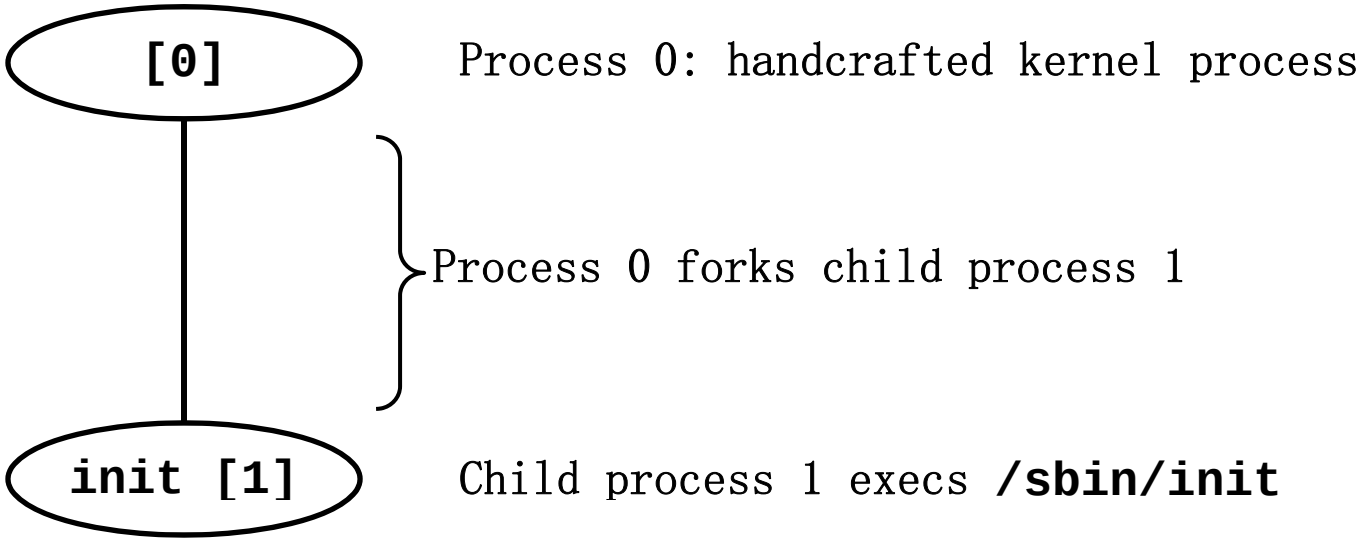
```
USER       PID TTY      STAT COMMAND
root         1 ?        S    init [3]
root         2 ?        SW   [keventd]
root         3 ?        SWN  [ksoftirqd_CPU0]
root         4 ?        SW   [kswapd]
root         5 ?        SW   [bdflush]
root         6 ?        SW   [kupdated]
root         9 ?        SW<  [mdrecoveryd]
root        12 ?        SW   [scsi_eh_0]
root       397 ?        S    /sbin/pump -i eth0
root       484 ?        S<   /usr/local/sbin/afsd -nosettime
root       533 ?        S    syslogd -m 0
root       538 ?        S    klogd -2
rpc        563 ?        S    portmap
rpcuser    578 ?        S    rpc.statd
daemon     696 ?        S    /usr/sbin/atd
root       713 ?        S    /usr/local/etc/nanny -init
/etc/nanny.conf
mmdf       721 ?        S     \_ /usr/local/etc/deliver -b -csmtpcmu
root       732 ?        S     \_ /usr/local/sbin/named -f
root       738 ?        S     \_ /usr/local/sbin/sshd -D
root       739 ?        S<L   \_ /usr/local/etc/ntpd -n
root       752 ?        S<L   |    \_ /usr/local/etc/ntpd -n
root       753 ?        S<L   |        \_ /usr/local/etc/ntpd -n
root       744 ?        S     \_ /usr/local/sbin/zhm -n zephyr-
1.srv.cm
root       774 ?        S    gpm -t ps/2 -m /dev/mouse
root       786 ?        S    crond
```

The ps 명령어 (cont.)

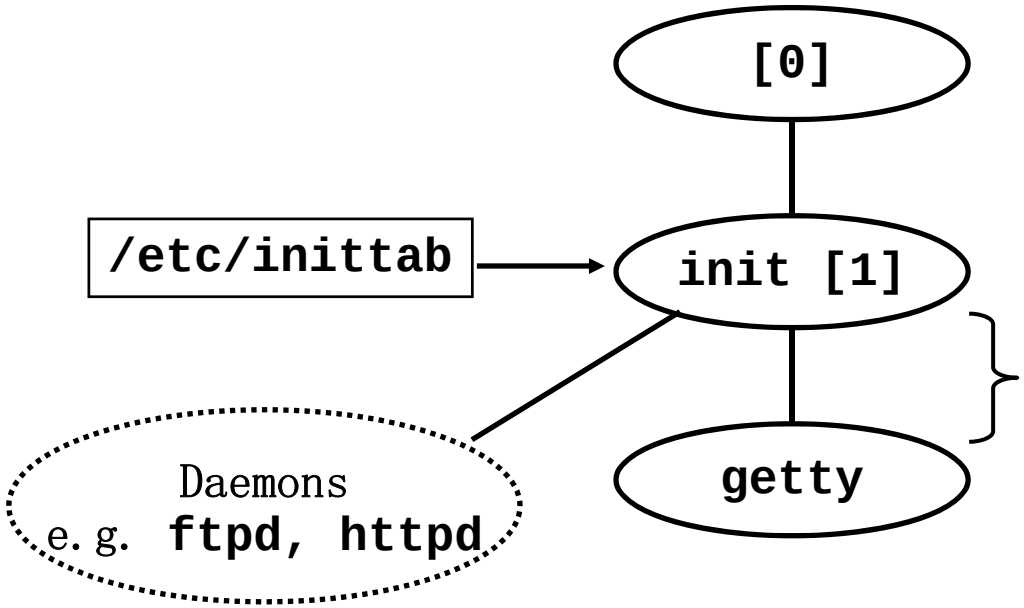
```
USER       PID TTY      STAT COMMAND
root       889 tty1     S    /bin/login -- agn
agn        900 tty1     S     \_ xinit -- :0
root       921  ?       SL        \_ /etc/X11/X -auth /usr1/agn/.Xauthority
:0
agn        948 tty1     S         \_ /bin/sh
/afs/cs.cmu.edu/user/agn/.xinitrc
agn        958 tty1     S             \_ xterm -geometry 80x45+1+1 -C -j -ls
-n
agn        966 pts/0    S             |   \_ -tcsh
agn       1184  pts/0   S             |       \_ /usr/local/bin/wish8.0 -f
/usr
agn       1212 pts/0    S             |       |   \_ /usr/local/bin/wish8.0
-f
agn       3346 pts/0    S             |       |   \_ aspell -a -S
agn       1191 pts/0    S             |       \_ /bin/sh
/usr/local/libexec/moz
agn       1204 8 pts/0  S             |           \_
/usr/local/libexec/mozilla
agn       1207 8 pts/0  S             |               \_
/usr/local/libexec/moz
agn       1208 8 pts/0  S             |               |   \_
/usr/local/libexec
agn       1209 8 pts/0  S             |               |   \_
/usr/local/libexec
agn      17814 8 pts/0  S             |               |   \_
/usr/local/libexec
agn       2469  pts/0   S             |               \_
usr/local/lib/Acrobat
agn       2483  pts/0   S             |                   \_ java_vm
agn       2484  pts/0   S             |                       \_ java_vm
agn       2485  pts/0   S             |                           \_ java_vm
agn       3042  pts/0   S             |                           \_ java_vm
agn        959 tty1     S         \_ /bin/sh
/usr/local/libexec/kde/bin/sta
agn       1020 tty1     S             \_ kwrapper ksmserver
```

Unix 초기화: Step 1

1. Pushing reset button loads the **PC** with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., **/boot/vmlinux**)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.

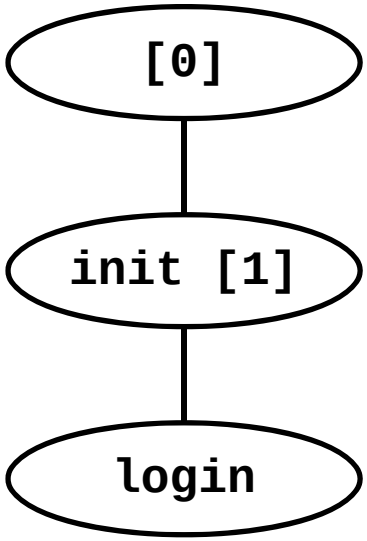


Unix 초기화: Step 2



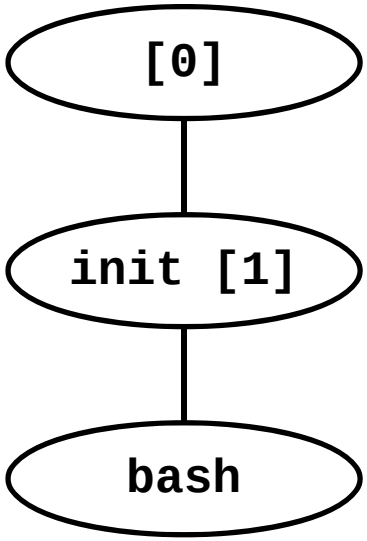
init forks and execs daemons per **/etc/inittab**, and forks and execs a **getty** program for the console

Unix 초기화: Step 3



The **getty** process execs a **login** program

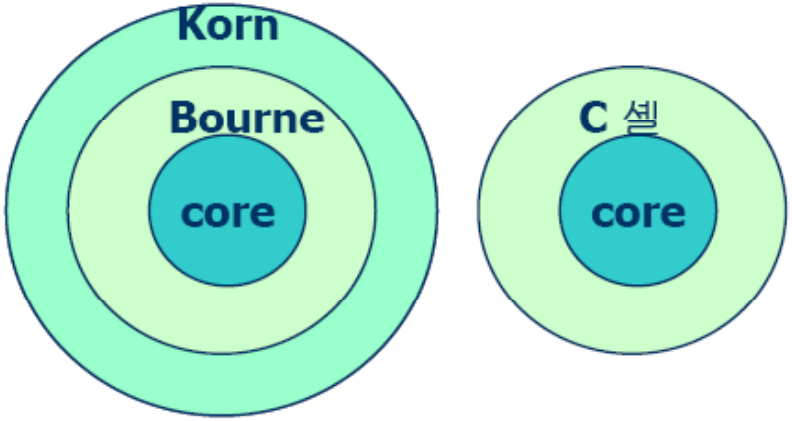
Unix 초기화: Step 4



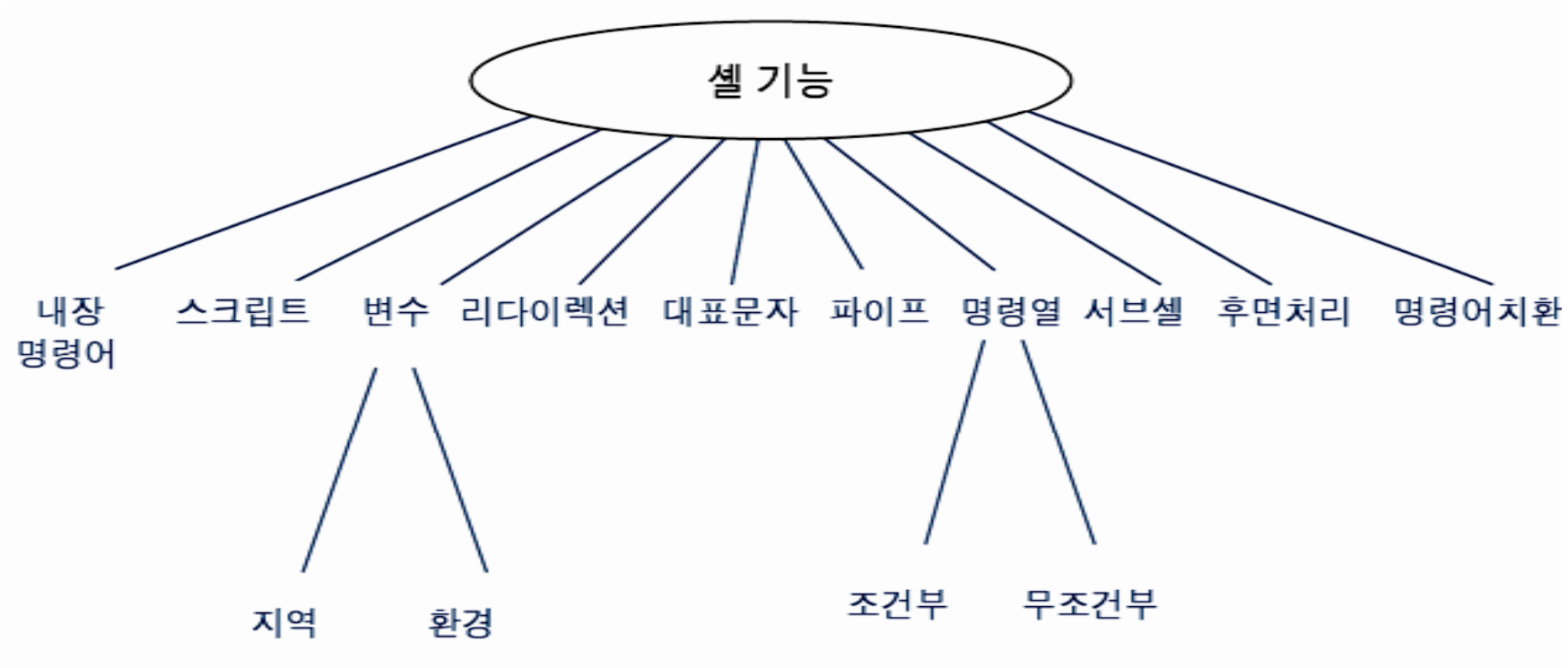
login reads login-ID and passwd.
if OK, it execs a *shell.*
if not OK, it execs another **getty**
In case of **login** on the console **xinit** may be used instead of a shell to start the window manger

쉘 : 폭탄, 조개 ? => 껍질

- 유닉스 셸(Unix Shell)
  - 사용자와 순수 Unix 운영체제 사이를 연결하는 프로그램
- Unix Shell의 종류 : 대표적인 3종류의 Shell
  - Korn Shell (ksh): David Korn 이 제작, Bourne과 C Shell 보다 포괄적, 산업체에서 선호, 유망
  - Bourne Shell (sh) : Stephen Bourne이 제작, 최초의 대중화된 Unix Shell
  - C Shell : interactive job에 유리
  - Bourne Again Shell (bash) : Best of all worlds



Core shell 기능



쉘 선택

- 계정 개설 시 선택 명시 /etc/passwd 파일 안에 login shell 정보 저장
- 자신의 login shell이 무엇인지를 확인하는 법 : $ echo $SHELL
- 현재 실행 중인 shell이 무엇인지를 확인하는 법 : $ ps
- Shell 호출 및 환경 설정
  - 자동호출 : login시에 특정 시작 파일을 읽어서 shell을 시작
    - **Bourne Shell, Korn Shell : .profile**
    - **C Shell : .login, .cshrc**
    - **Bourne-Again Shell : .bash_profile, .login, .bashrc**
  - 수동 호출 - solaris 에서는 다른 방법
    - **/bin/sh, /bin/ksh : . .profile**
    - **/bin/csh: source .cshrc**
    - **/bin/bash: . .bash_profile**

Utility와 Built-in 명령어

- Shell의 built-in 명령어는 utility program과의 차이는 shell에 내장, search path에서 찾기 전에 실행
- Q. cd ? ls ?

Background processing

- 리눅스 Job 제어
  - foreground job과 background job
  - foreground job 을 suspend 시키려면 Ctrl+z(Send SIGSTOP signal)
  - fg => resume as foreground, bg=>as background(Send SIGCONT)
- 백그라운드 작업
  - 수행시키려면 명령줄 뒤에 & 추가
  - 자식셸로 생성되어 부모 셸과 같이 수행되나 키보드를 제어하지 않음
  - 예:

    $find . –name b.c –print &

    cf. $find . –name b.c –print
  - $date & pwd & # 두개의 백그라운드 프로세스 생성

Background Job

- Users generally run one command at a time
  - Type command, read output, type another command
- Some programs run “for a long time”
  - Example: “delete this file in two hours”
  - % sleep 7200; rm/tmp/junk # shell stuck for 2 hours
- A “background” job is a process we don't want to wait for
  - % (sleep 7200 ; rm/tmp/junk) &
  - [1] 907
  - % # ready for next command

단순한 쉘 프로그램의 문제

- Shell correctly waits for and reaps foreground jobs
- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could theoretically run the kernel out of memory
  - Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1
  - % limit maxproc# cshsyntax
  - maxproc3574
  - $ ulimit-u # bash syntax
  - 3574

fork 와 execve 사용하기: 쉘 프로그램

- 쉘 *shell* 은 사용자의 명령을 처리해 주는 응용 프로그램이다
  - sh – Original Unix Bourne Shell
  - csh – BSD Unix C Shell
  - tcsh – Enhanced C Shell
  - bash –Bourne-Again Shell

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        Fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

- Execution is a sequence of read/evaluate steps

간단한 쉘의 eval 함수

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

간단한 쉘 구현의 문제점

- Shell correctly waits for and reaps foreground jobs.
- But what about background jobs?
  - **Will become zombies when they terminate.**
  - **Will never be reaped because shell (typically) will not terminate.**
  - **Creates a memory leak that will eventually crash the kernel when it runs out of memory.**
- Solution
  - **Reaping background jobs requires a mechanism called a *signal*.**