Process Algorithm (Chap7-Chap9)

Hongik University

Eunsung Jung

Disclaimer: The slides are borrowed from many sources!

Process Control

- Control of Process Context
  - **fork** : create a new process
  - **exit** : terminate process execution
  - **wait** : allow parent process to synchronize its execution with the exit of a child process
  - **exec** : invoke a new program
  - **brk** : allocate more memory dynamically
  - **signal** : inform asynchronous event
  - major loop of the **shell** and of **init**

System Calls Dealing with Memory Management			System Calls Dealing with Synchronization				Miscellaneous	
fork	exec	brk	exit	wait	signal	kill	setpgrp	setuid
dupreg attachreg	detachreg allocreg attachreg growreg loadreg mapreg	growreg	detachreg					

Sequence of Operations for fork

1. It allocates a slot in process table for the new process

2. It assigns a unique ID number to the child process

3. It makes a logical copy of the context of the parent process. Since certain portion of process, such as the text region, may be shared between processes, the kernel can sometimes increment a region reference count instead of copying the region to a new physical location in memory

4. It increments file and inode table counters for files associated with the process.

5. It returns the ID number of child process to the parent process, and a 0 value to the child process.

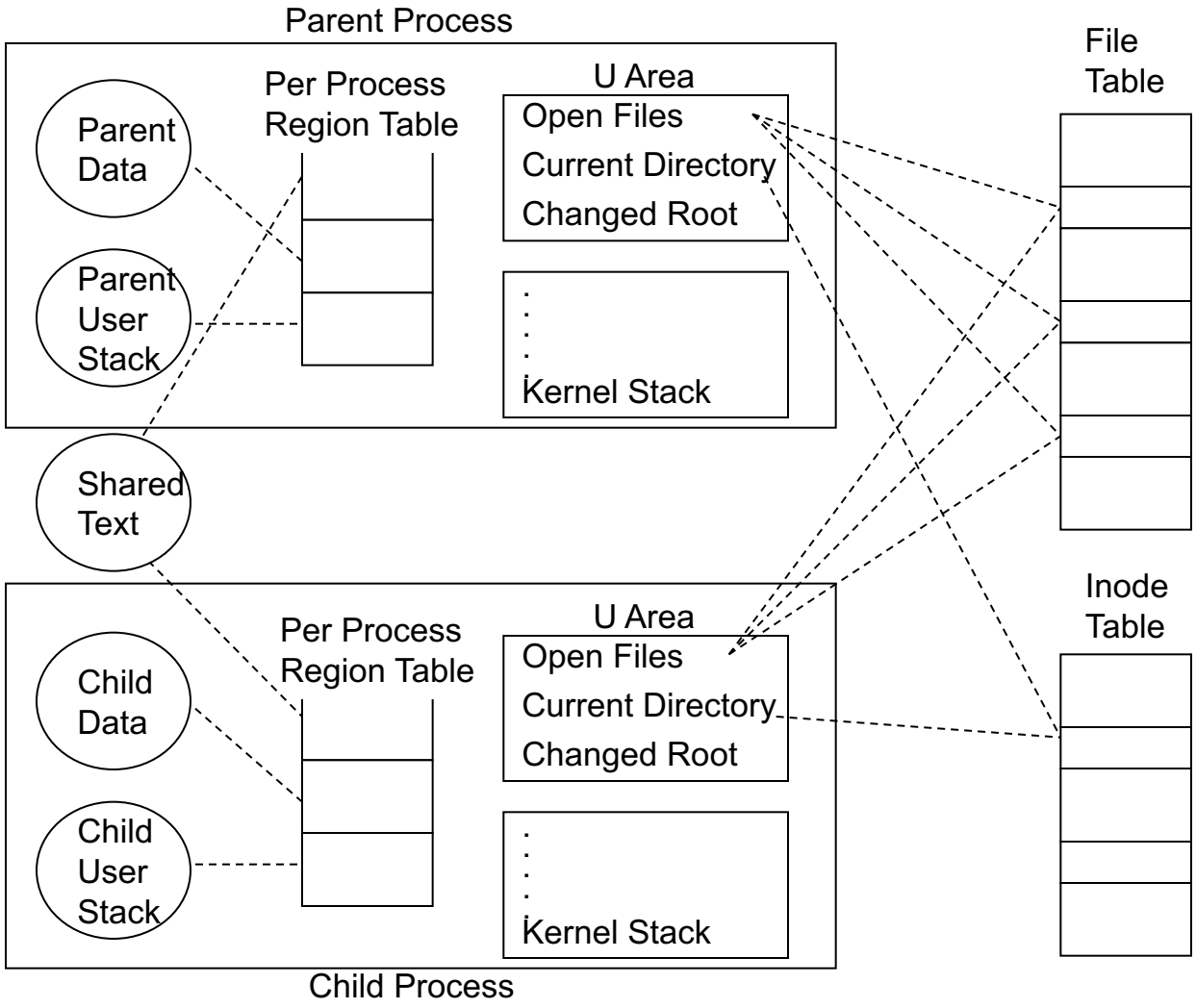
Algorithm for fork

```
input : none
output : to parent process, child PID number
         to child process, 0
{
    check for available kernel resources;
    get free process table slot, unique PID number;
    check that user not running too many process;
    mark child state "being created";
    copy data from parent process table to new child slot;
    increment counts on current directory inode and changed root(if applicable);
    increment open file counts in file table;
    make copy of parent context(u area, text, data, stack) in memory;
    push dummy system level context layer onto child system level context;
                        dummy context contains data allowing child process
                        to recognize itself, and start from here
                        when scheduled;
```

Algorithm for fork(Cont.)

```
    if ( executing process is parent )
    {
            change child state to "ready to run";
            return(child ID);                    /* from system to user */
    }
    else    /* executing process is the child process */
    {
            initialize u area timing fields;
            return(0);                  /* to user */
    }
}
```

Fork Creating a New Process Context



Example of Sharing File Access: Results?

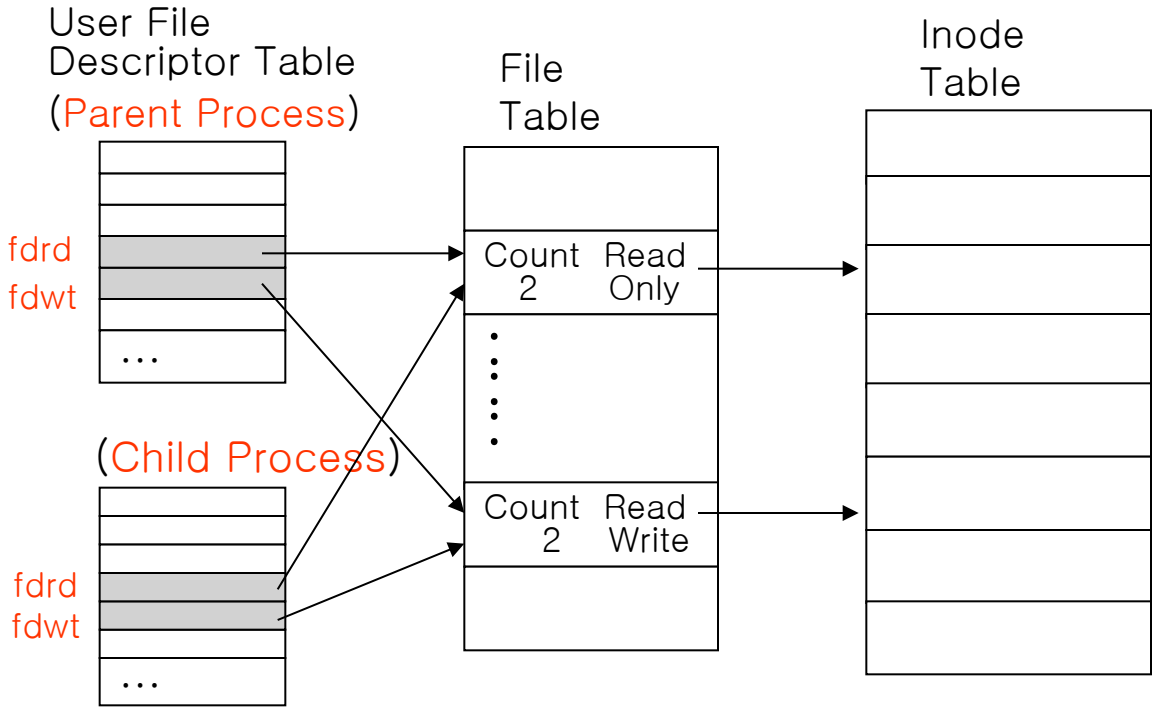
```
#include <fcntl.h>
int fdrd, fdwt;
char c;

main( argc,argv )
        int argc;
        char *argv[];
{
        if ( argc != 3 )
           exit(1);
        if ((fdrd=open(argv[1],O_RDONLY))==-1)
           exit(1);
        if ((fdwt=creat(argv[2],0666))==-1)
           exit(1);
        fork();
        /*both process execute same code*/
        rdwt();
        exit(0);
}
```

```
rdwt()
{
        for(;;)
        {
          if (read(fdrd,&c,1)!=1)
                return;
          write(fdwt,&c,1);
        }
}
```

Example of Sharing File Access(Cont.)

- *fdrd* for both process refer to the file table entry for the source file(argv[1])
- *fdwt* for both process refer to the file table entry for the target file(argv[2])
- two processes never read or write the same file offset values.

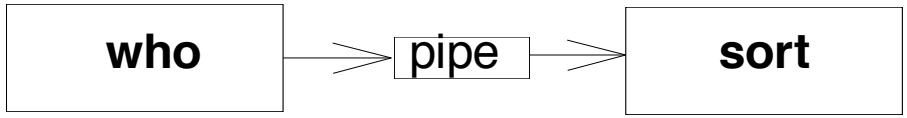


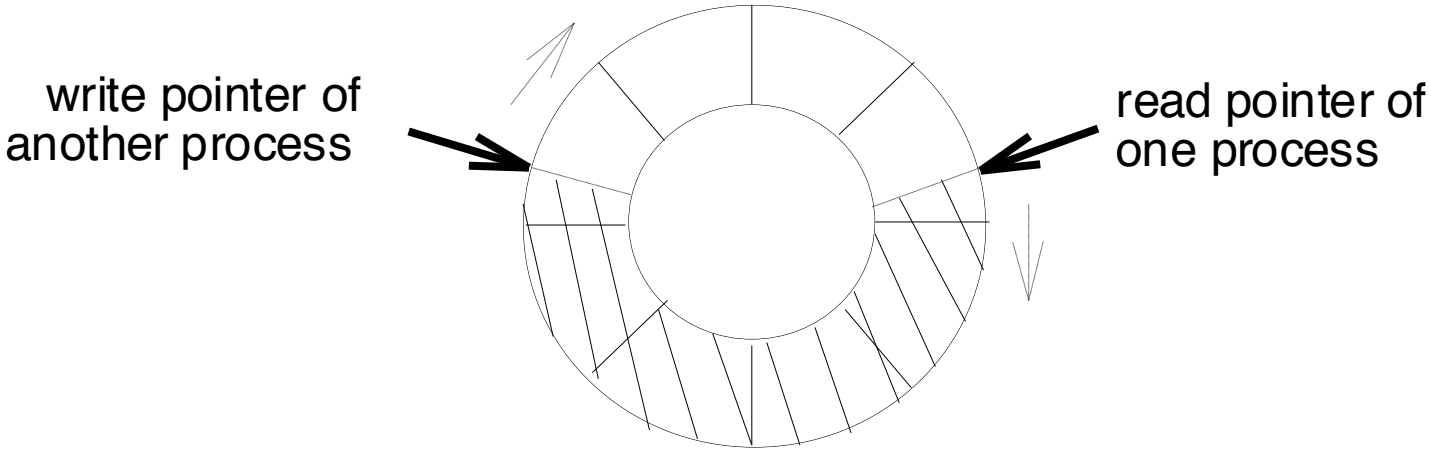
Interprocess Communication

- IPC using regular files
  - unrelated processes can share
  - fixed size
  - lack of synchronization
- IPC using pipes
  - for transmitting data between related processes
  - can transmit an unlimited amount of data
  - automatic synchronization on open()

Pipe

```
$ who | sort
```





Process Termination

- *exit* system call
  - process terminate by *exit* system call
    - enters the zombie status
    - relinquish resources (close all open files)
    - buffered output written to disk
    - dismantles its context except for its slot in the process table.
  - exit(status);
    - status : the value returned to parent process
    - can be used for unix shell (shell programming)
  - call exit explicitly or implicitly(by startup routine) at the end of program.
  - kernel may invoke internally on receipt of uncaught signals. In this case, the value of status is the signal number.

Exit handler

```
#include <stdlib.h>

void atexit(void (*func)(void));
                          returns: 0 if OK, nonzero on error
```

- Register exit handler
  - Register a function that is called when a program is terminated
  - Called in reverse order of registration

Example: Exit handler

```
/* doatexit.c */
static void my_exit1(void), my_exit2(void);

int main(void) {
  if (atexit(my_exit2) != 0)
    perror("can't register my_exit2");
  if (atexit(my_exit1) != 0)
    perror("can't register my_exit1");
  if (atexit(my_exit1) != 0)
    perror("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");
}
```

Output :

main is done
first exit handler
first exit handler
second exit handler

Algorithm for Exit

```
algorithm exit
input : return code for parent process
output : none
{
    ignore all signals;
    if ( process group leader with associated control terminal )
    {
        send hangup signal to all members of process group;
        reset process group for all members to 0;
    }
    close all open files(internal version of algorithm close)
    release current directory(algorithm iput);
    release current(changed) root, if exists (algorithm iput);
    free regions, memory associated with process(algorithm freereg);
    write accounting record;
    make process state zombie;
    assign parent process ID of all child processes to be init process(1);
        if any children were zombie, send death of child signal to init;
    send death of child signal to parent process;
    context switch;
}
```

Awaiting Process Termination

- *wait* system call
  - synchronize its execution with the termination of a child process
  - pid = wait(stat_addr);
    - pid : process id of the zombie child process
    - stat_addr : address of an integer to contain the exit status of the child

Algorithm for Awaiting Process Termination

1. searches the zombie child process
2. If no children, return error
3. if finds zombie children, extracts PID number and exit code
4. adds accumulated time the child process executes in the user and kernel mode to the fields in u area
5. Release process table slot

Algorithm for Wait

```
algorithm wait
input : address of variables to store status of exiting process
output : child ID, child exit code
{
    if (waiting process has no child process)
        return(error);
    for(;;)
    {
        if (waiting process has zombie child)
        {
            pick arbitrary zombie child;
            add child CPU usage to parent;
            free child process table entry;
            return(childID,child exit code);
        }
        if (process has no child process)
            return(error);
        sleep at interruptible priority(event child process exits);
    }
}
```

Invoking Other Programs

- *exec* system call
  - invokes another program, overlaying the memory space of a process with a copy of an executable file
  - execve(filename, argv, envp)
    - filename : the name of executable file being invoked
    - argv : a pointer to an array of character pointers that are parameters to the executable program
    - envp : a pointer array of character pointers that are environment of the executed program
  - several library functions that calls *exec* system call

    execl, execv, execle...

Algorithm for Exec

```
algorithm exec
input : (1) file name
        (2) parameter list
        (3) environment variables list
output : none
{
    get file inode(algorithm namei)
    verify file executable, user has permission to execute;
    read file headers, check that it is a load module;
    copy exec parameters from old address space to system space;
    for(every region attached to process)
            detach all old regions(algorithm detach);
    for(every region specified in load module)
    {
            allocate new regions(algorithm allocreg);
            attach the regions(algorithm attachreg);
            load region into memory if appropriate(algorithm loadreg);
    }
    copy exec parameters into new user stack region;
    special processing for setuid programs, tracing;
    initialize user register save area for return to user mode;
    release inode of file(iput);
}
```

Environment variables

- Environment variables(EV) are inherited from parent to child process

- Generally, EV are set in .login or .cshrc

```
$ env
USER=ysmoon
LOGNAME=ysmoon
HOME=/home/prof/ysmoon
PATH=/bin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/ucb:/usr/ope
   nwin/bin:/etc:.
SHELL=/bin/csh
...
...
```

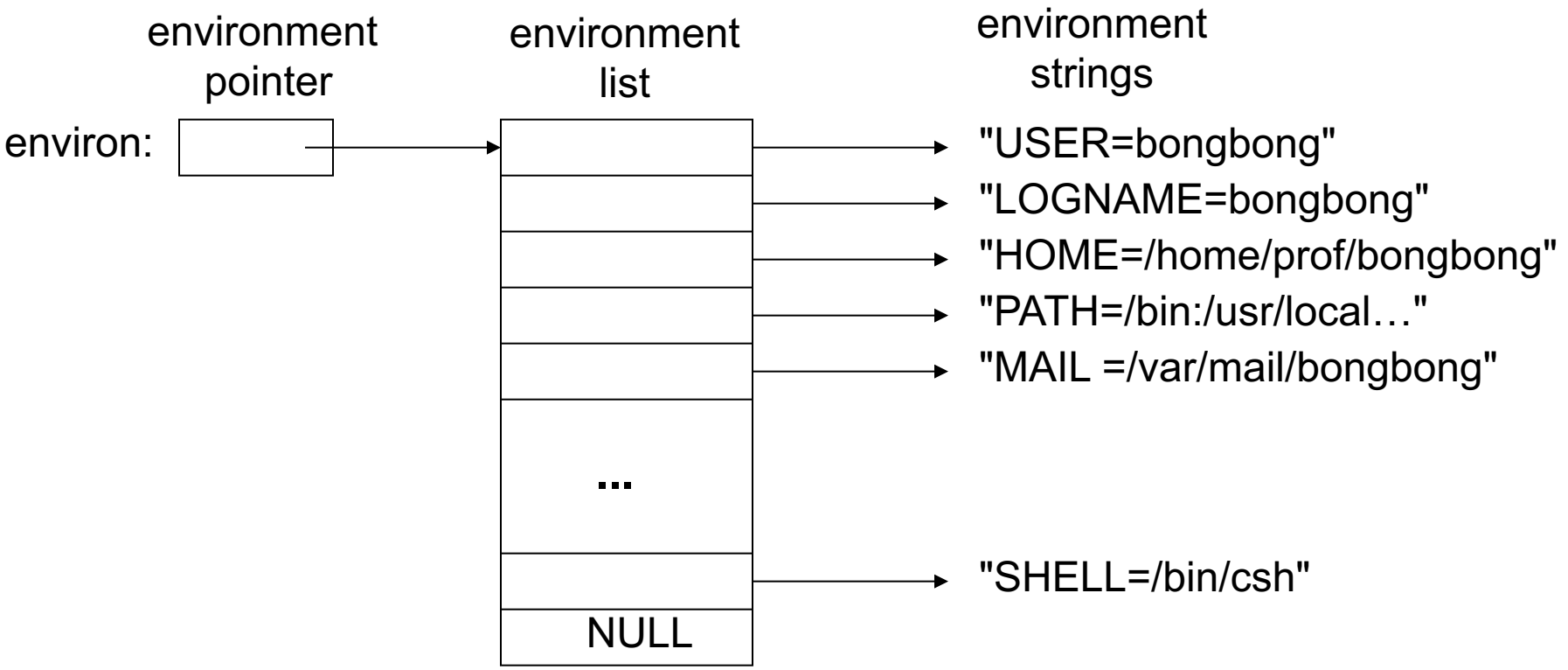
Environment list

- Environment variables are accessed through global variable environ

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

- Each element has a form of “Name=Value”
  - Each string ends with ‘\0’
  - Last element of environ is NULL pointer

Environment list



getenv/putenv

```
#include <stdlib.h>

char *getenv(const char *name);

      Returns : pointer to value associated with name, NULL if not found
```

```
#include <stdlib.h>

int putenv(const char *str);    // str : "name=value"

       Returns: 0 if OK, nonzero on error
```

setenv/unsetenv

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int rewrite);
           Returns: 0 if OK, nonzero on error

void unsetenv(const char *name);
```