

Computer Systems II

Creating and Executing Processes

Unix system calls

fork()

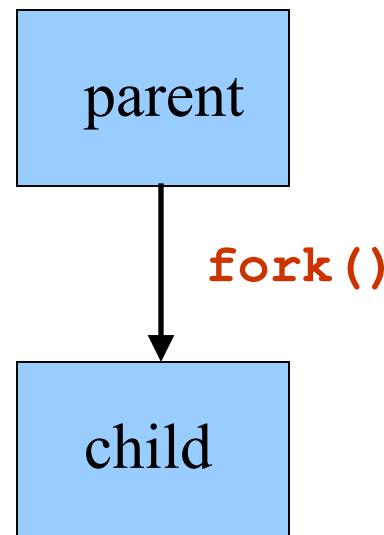
wait()

exit()

How To Create New Processes?

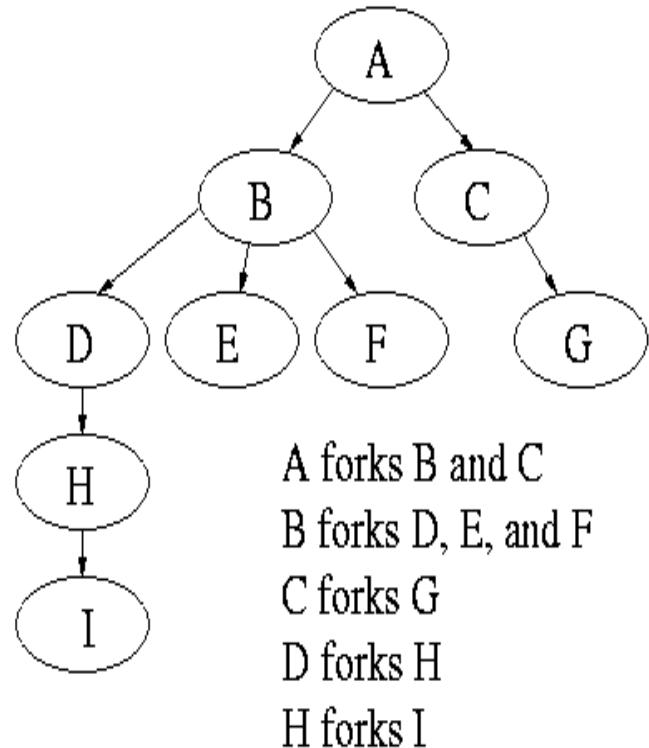
- Underlying mechanism

- A process runs **fork** to create a child process
- Parent and children execute concurrently
- Child process is a duplicate of the parent process



Process Creation

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A **process tree** results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a **wait()** system call, it would block until G finished.

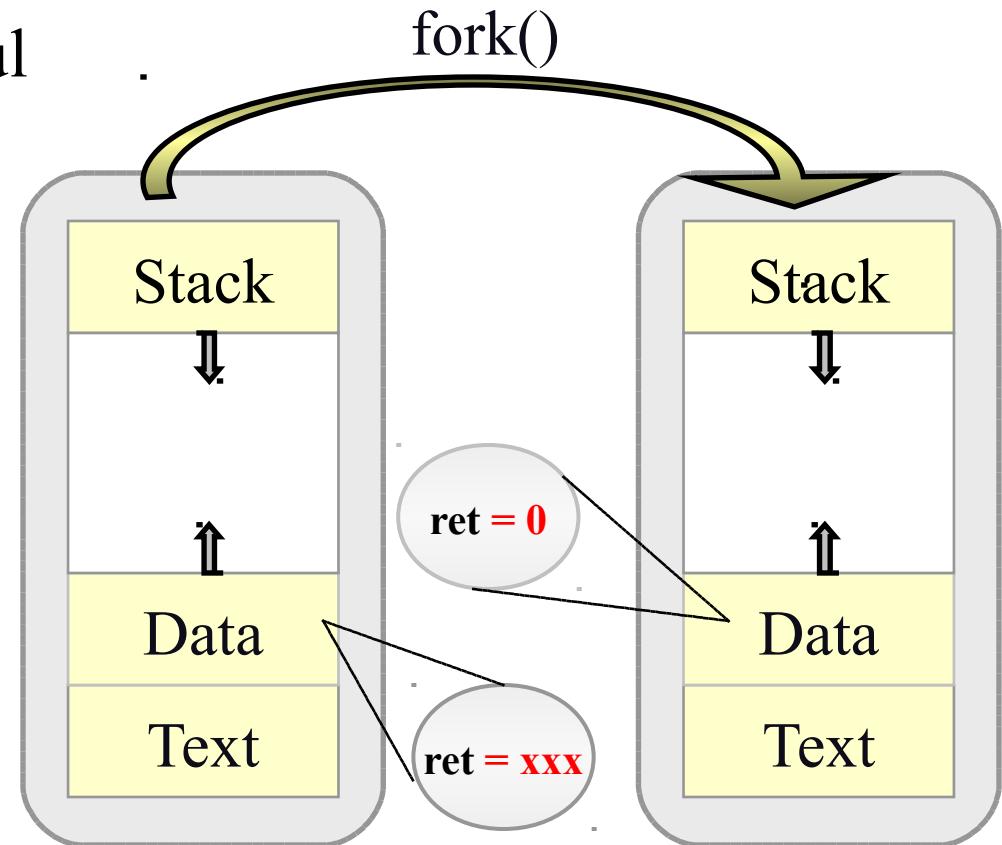


Bootstrapping

- When a computer is switched on or reset, there must be an initial program that gets the system running
- This is the bootstrap program
 - Initialize CPU registers, device controllers, memory
 - Load the OS into memory
 - Start the OS running
- OS starts the first process (such as “init”)
- OS waits for some event to occur
 - Hardware interrupts or software interrupts (traps)

Fork System Call

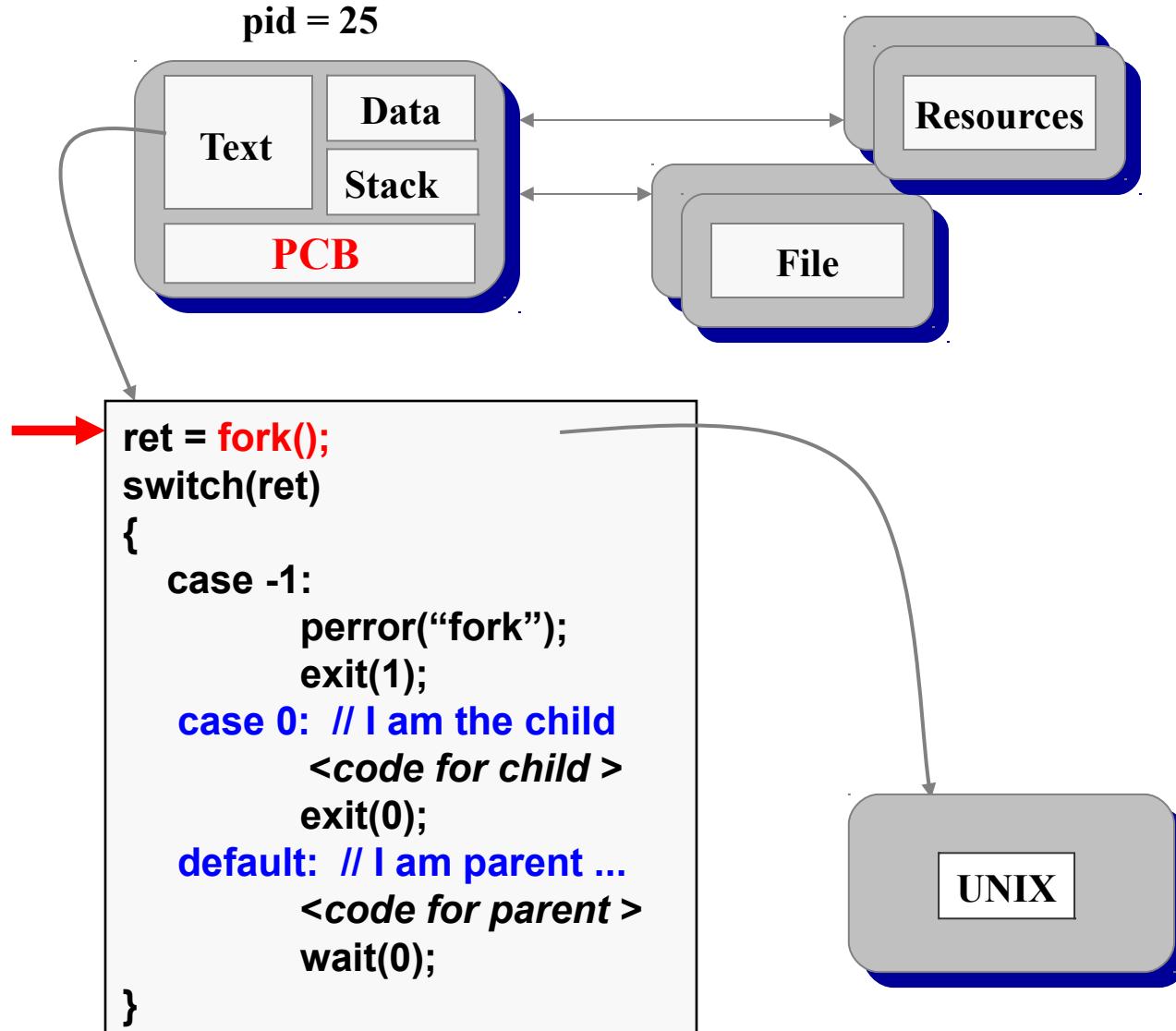
- Current process split into 2 processes: parent, child
- Returns -1 if unsuccessful
- Returns 0 in the child
- Returns the child's identifier in the parent



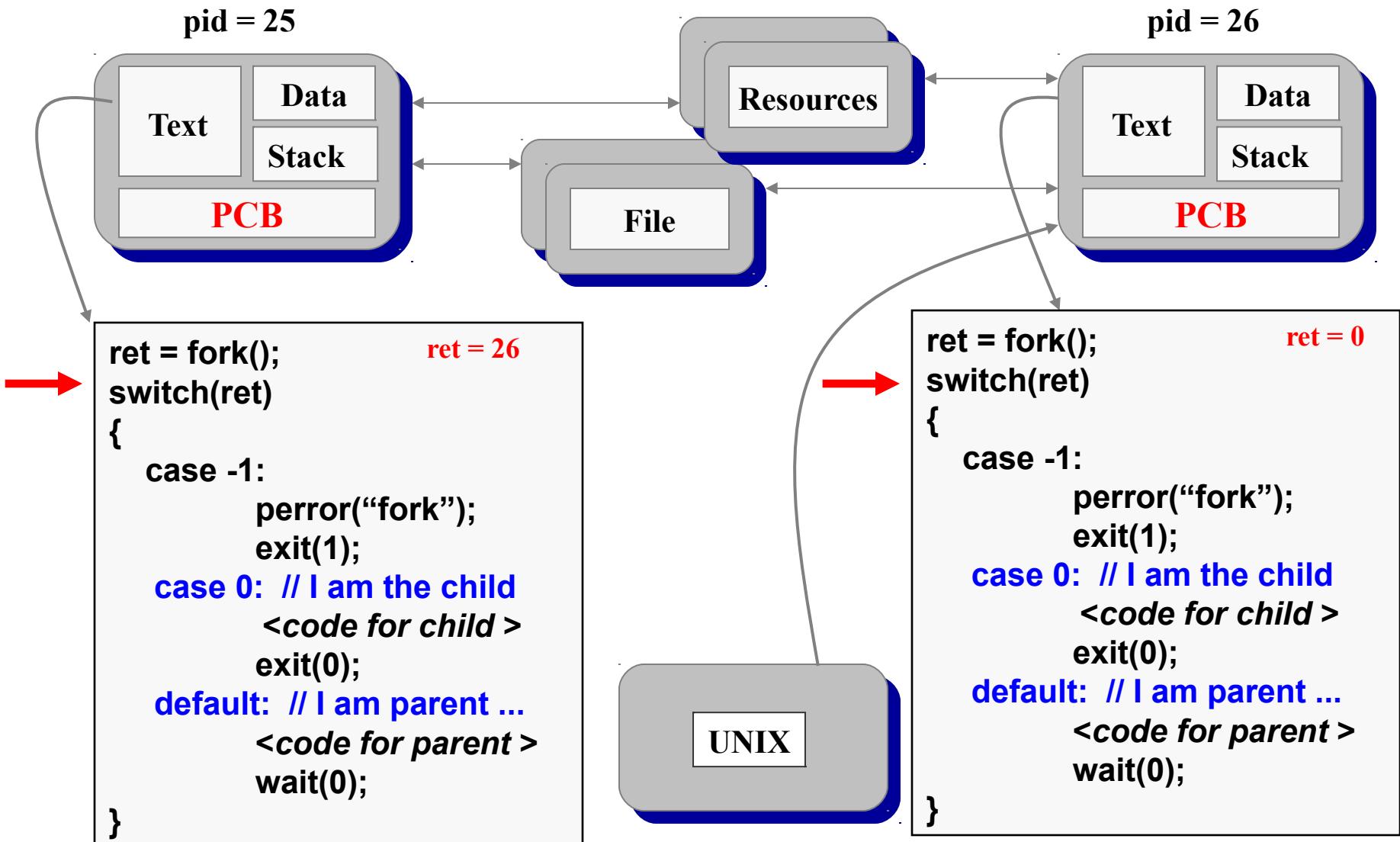
Fork System Call

- The child process inherits from parent
 - identical copy of memory
 - CPU registers
 - all files that have been opened by the parent
- Execution proceeds **concurrently** with the instruction following the fork system call
- The execution context (PCB) for the child process is a copy of the parent's context at the time of the call

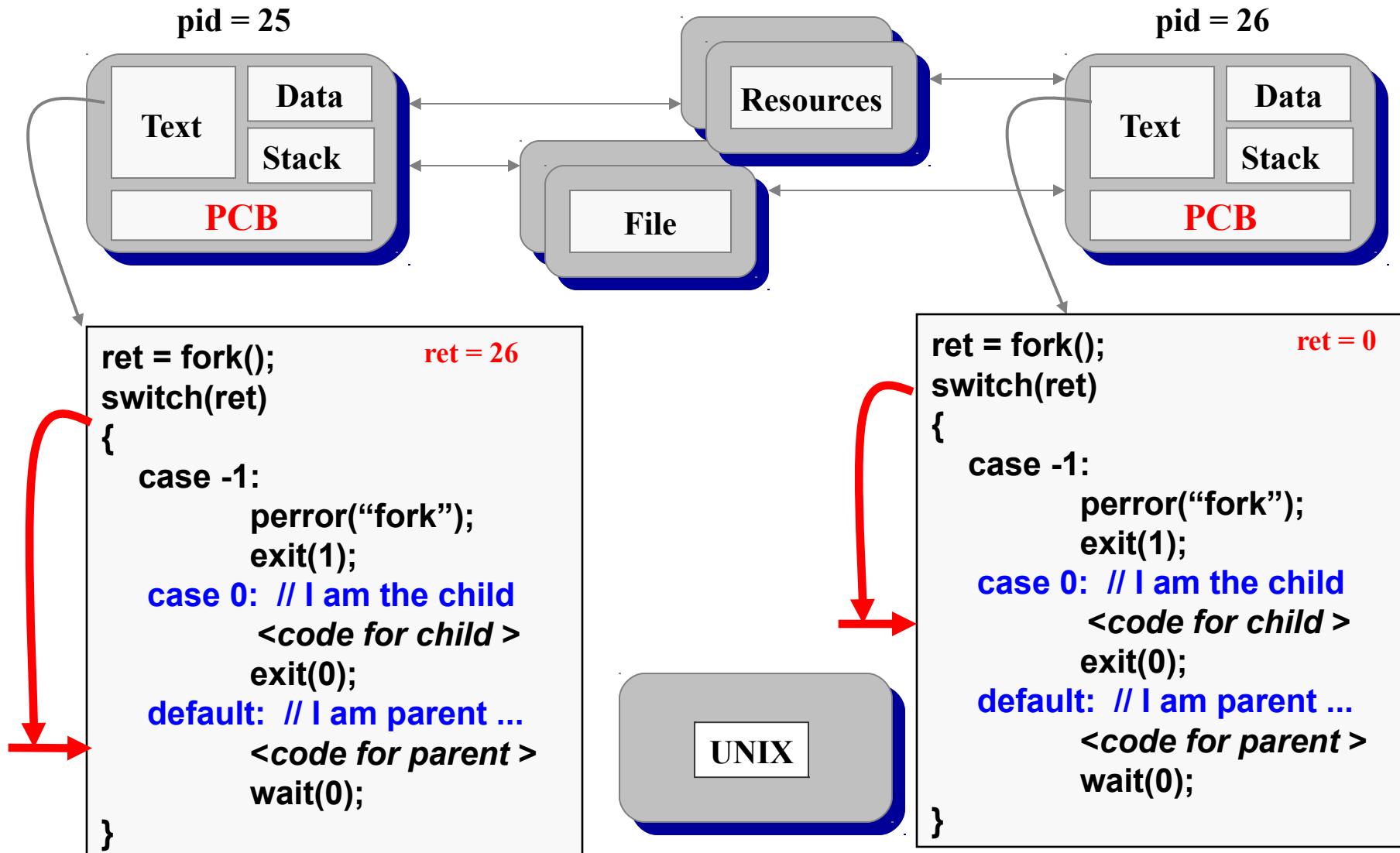
How fork Works (1)



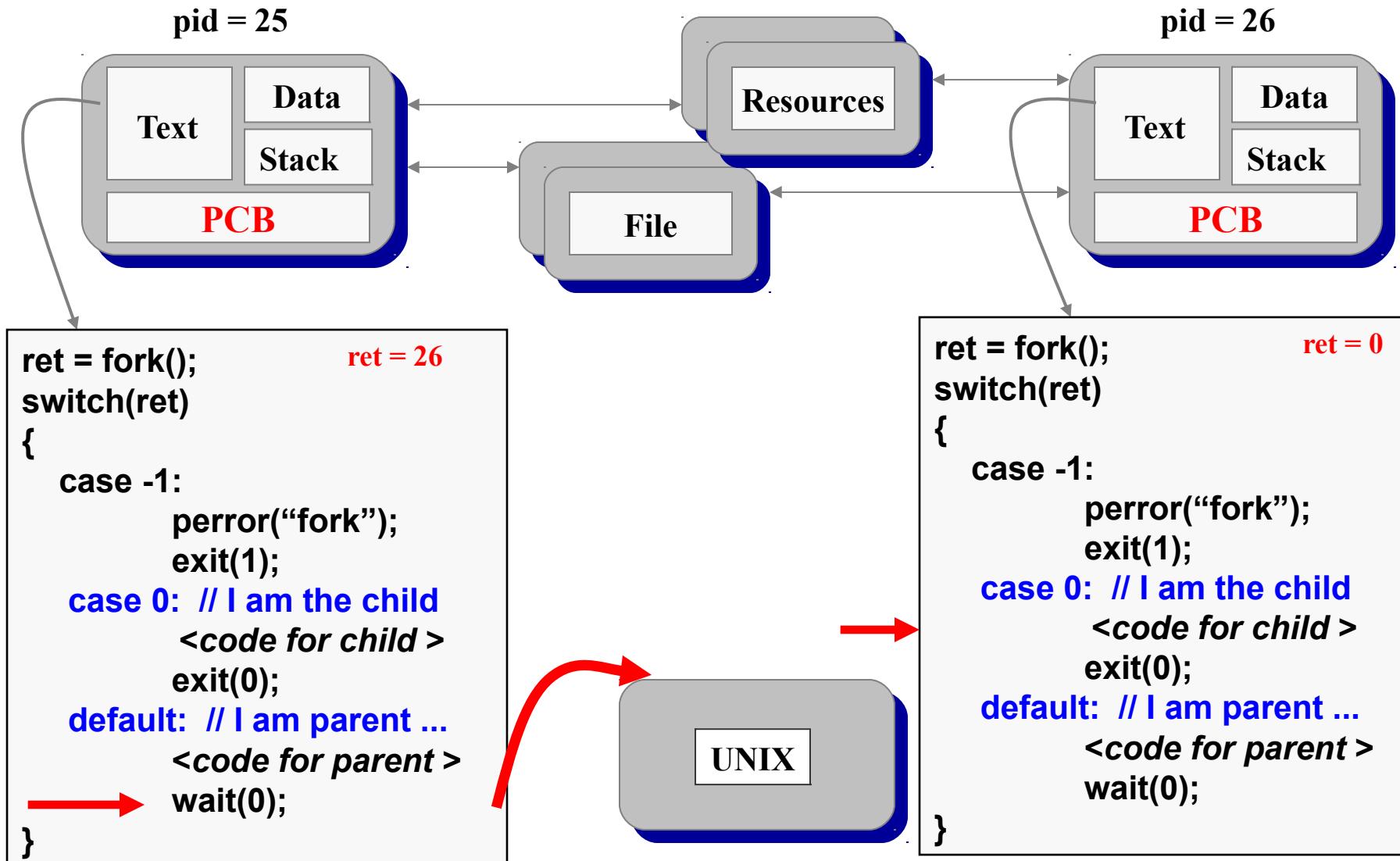
How fork Works (2)



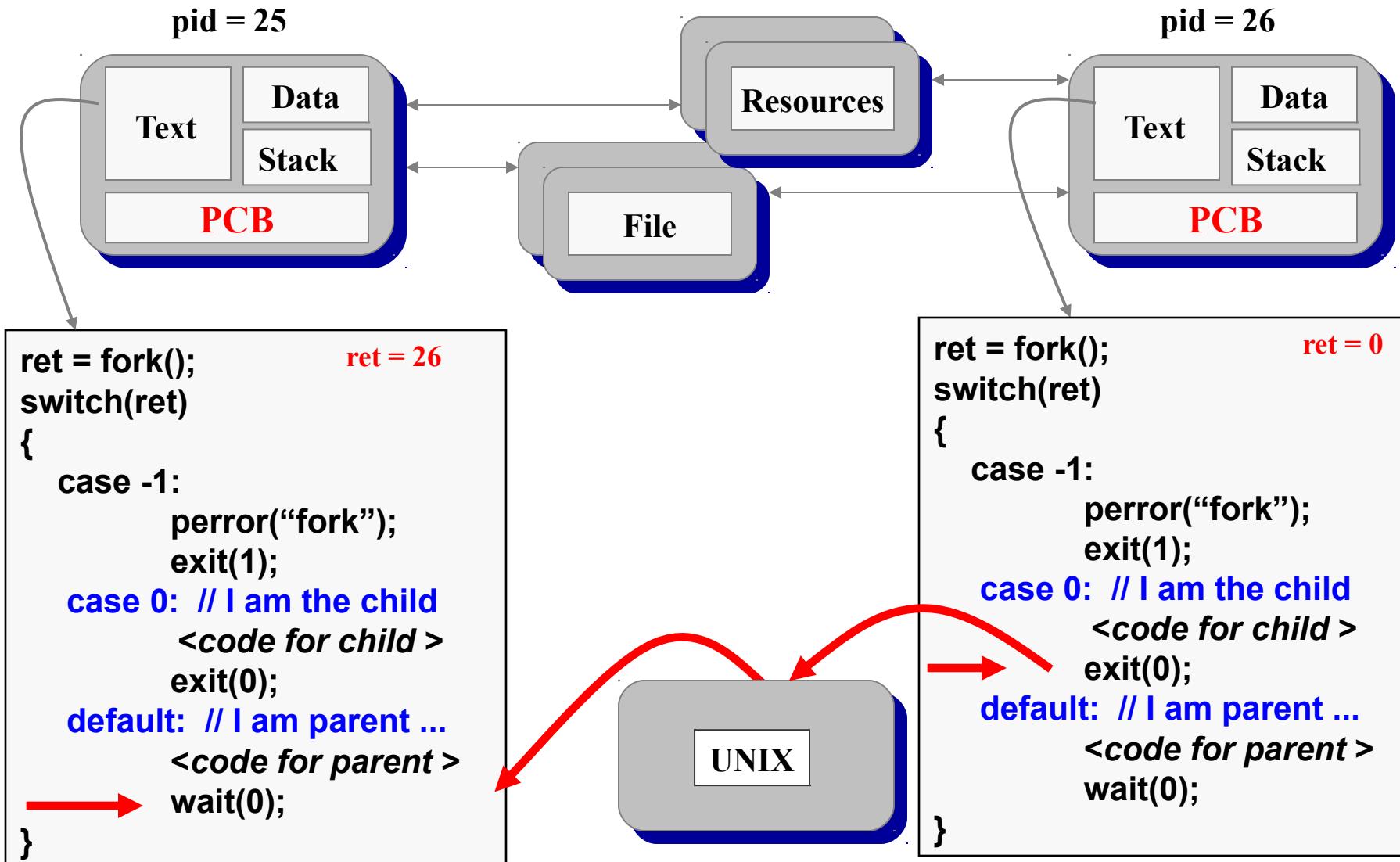
How fork Works (3)



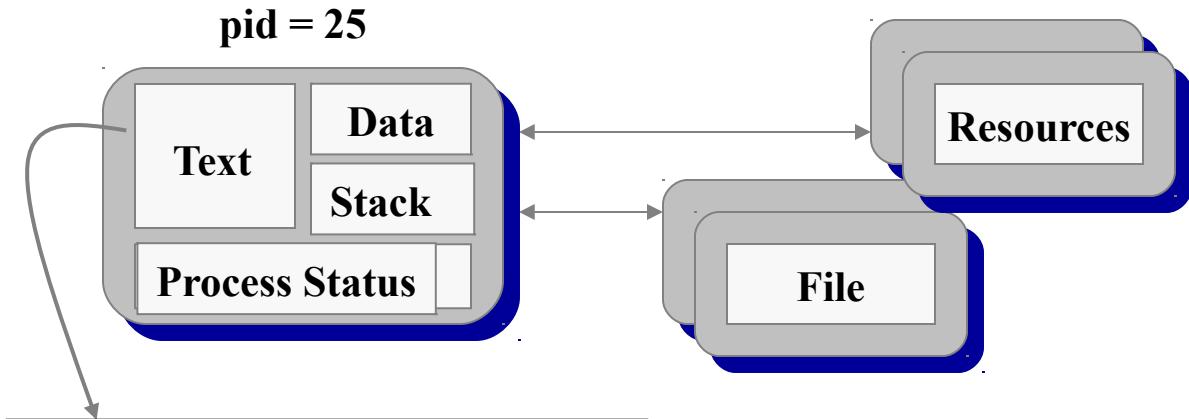
How fork Works (4)



How fork Works (5)



How fork Works (6)



```
ret = fork();
switch(ret)
{
    case -1:
        perror("fork");
        exit(1);
    case 0: // I am the child
        <code for child>
        exit(0);
    default: // I am parent ...
        <code for parent>
        wait(0);
        < ... >
    }
```



Orderly Termination: exit()

- To finish execution, a child may call `exit(number)`
- This system call:
 - Saves result = argument of exit
 - Closes all open files, connections
 - Deallocates memory
 - Checks if parent is alive
 - If parent is alive, holds the result value until the parent requests it (with `wait`); in this case, the child process does not really die, but it enters a zombie/defunct state
 - If parent is not alive, the child terminates (dies)

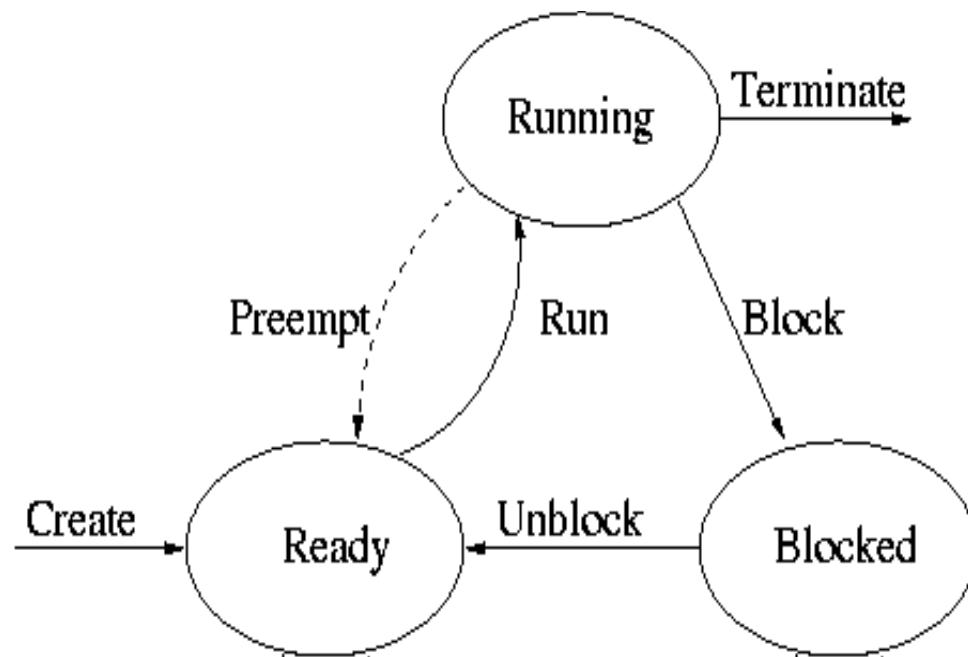
Waiting for the Child to Finish

- Parent may want to wait for children to finish
 - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: `wait()`
 - Blocks until some child terminates
 - Returns the process ID of the child process
 - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: `waitpid()`
 - Blocks till a child with particular process ID terminates

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

State Transition on wait and exit Calls



Other useful system calls: `getpid`, `getppid`

- `getpid` returns the identifier of the calling process. Example call (pid is an integer):

```
pid = getpid();
```

- `getppid` returns the identifier of the parent.
- Note:
 - Zombies can be noticed by running the '`ps`' command (shows the process list); you will see the string "<`defunct`>" as their command name:

```
ps -ef
```

```
ps -ef | grep mdamian
```

Fork Example 1: What Output?

```
int main()
{
    pid_t pid;
    int x = 1;

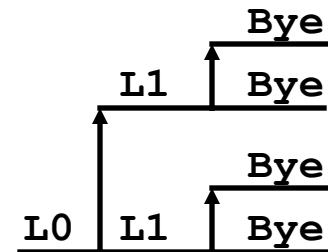
    pid = fork();
    if (pid != 0) {
        printf("parent: x = %d\n", --x);
        exit(0);
    } else {
        printf("child: x = %d\n", ++x);
        exit(0);
    }
}
```

Fork Example 2

■ Key Points

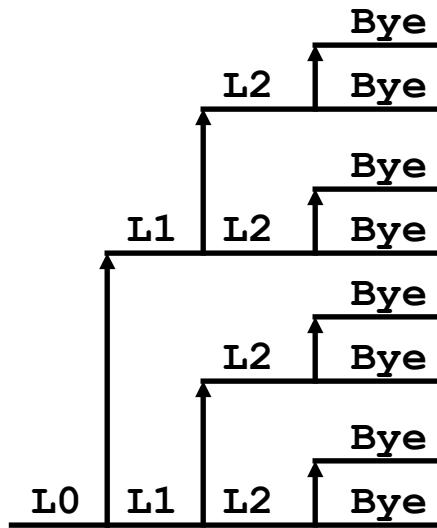
- Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example 3

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example 4

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

Fork Example 5

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```

Summary

- Fork
 - Creates a duplicate of the calling process
 - The result is two processes: parent and child
 - Both continue executing from the same point on
- Exit
 - Orderly program termination
 - Unblock waiting parent
- Wait
 - Used by parent
 - Waits for child to finish execution

Unix system calls

execv
execl

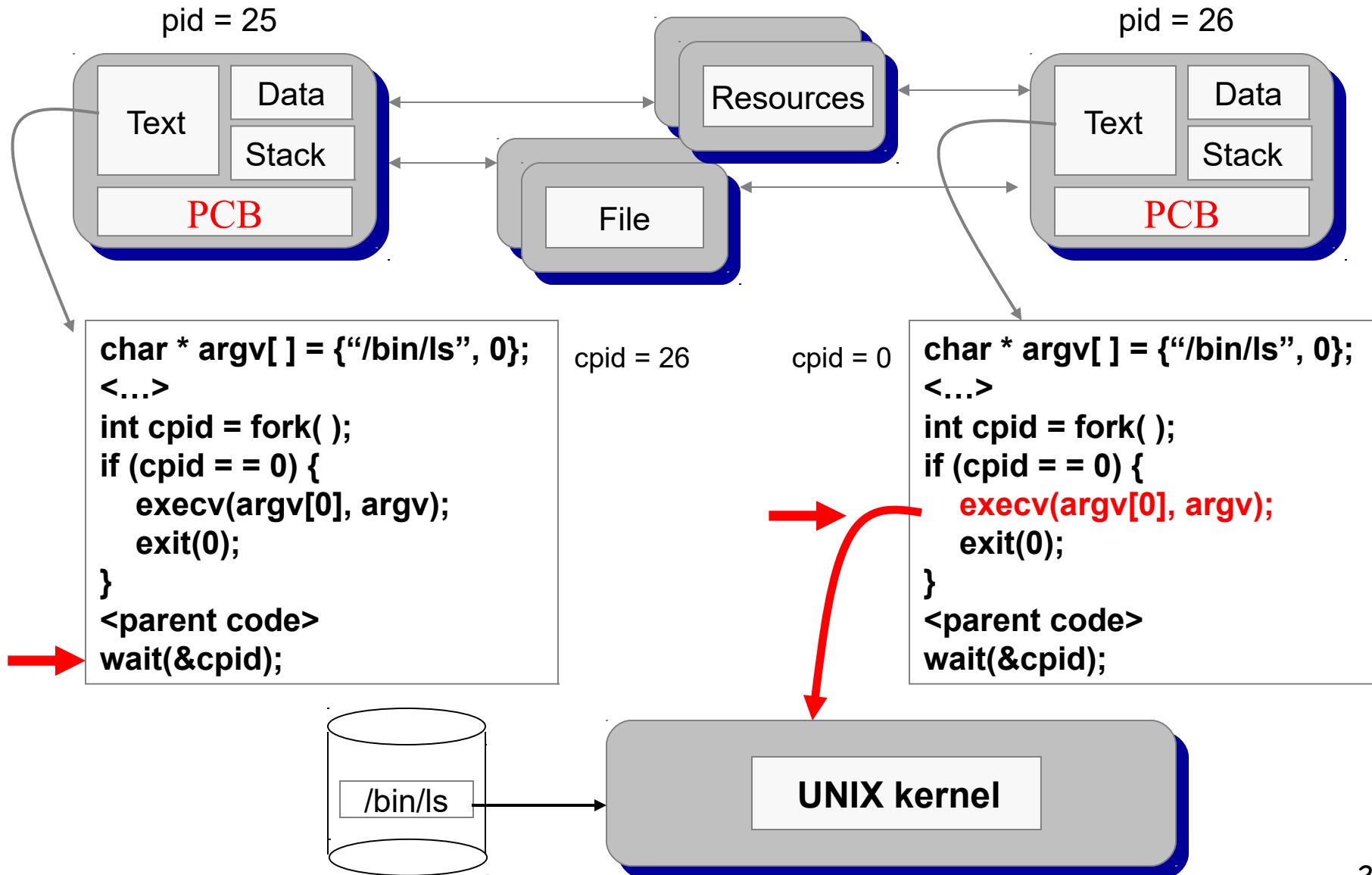
Unix's execv

- The system call **execv** executes a file, transforming the calling process into a new process. After a successful **execv**, there is no return to the calling process.

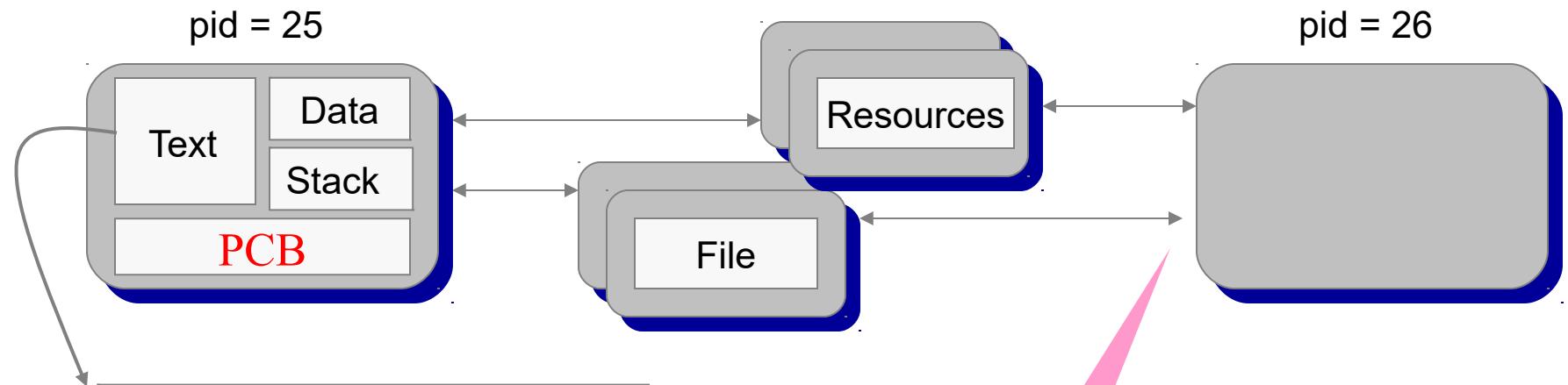
```
execv(const char * path, char * const argv[])
```

- **path** is the full path for the file to be executed
- **argv** is the array of arguments for the program to execute
 - each argument is a null-terminated string
 - the first argument is the name of the program
 - the last entry in argv is NULL

How execv Works (1)



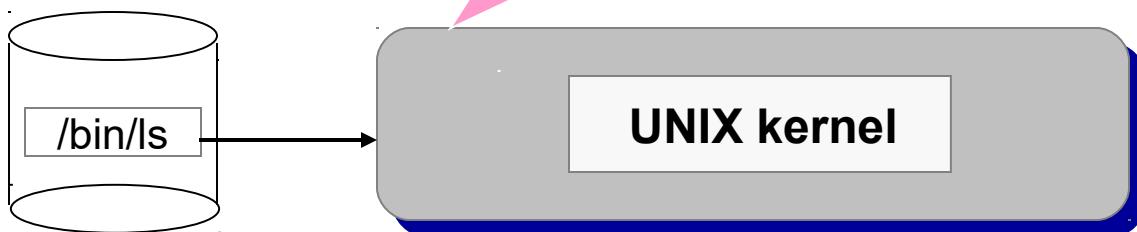
How execv Works (2)



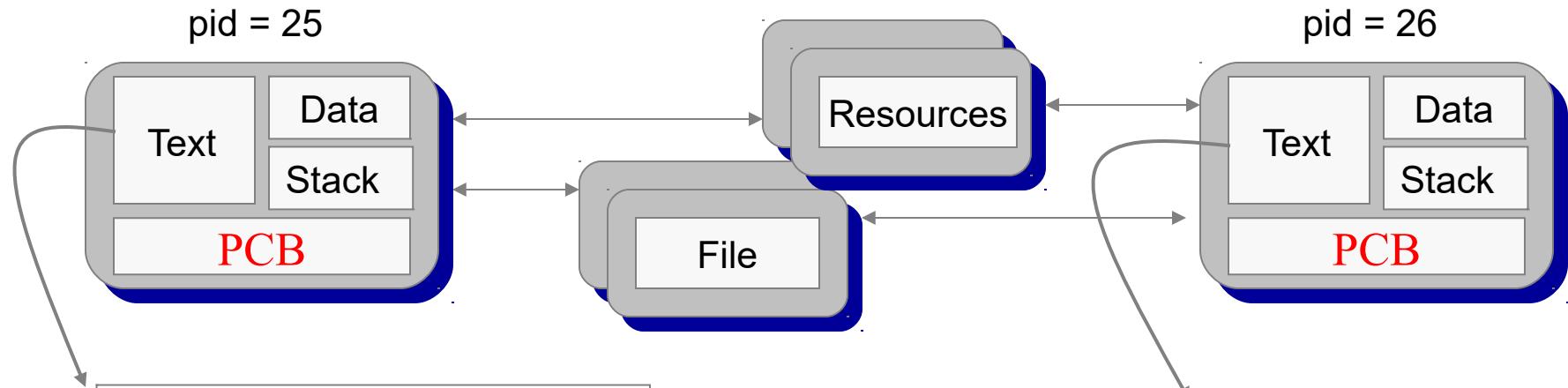
```
char * argv[ ] = {"/bin/ls", 0};  
<...>  
int cpid = fork();  
if (cpid == 0) {  
    execv(argv[0], argv);  
    exit(0);  
}  
<parent code>  
wait(&cpid);
```

cpid = 26

Exec destroys the process image of the calling process. A new process image is constructed from the executable file (ls).



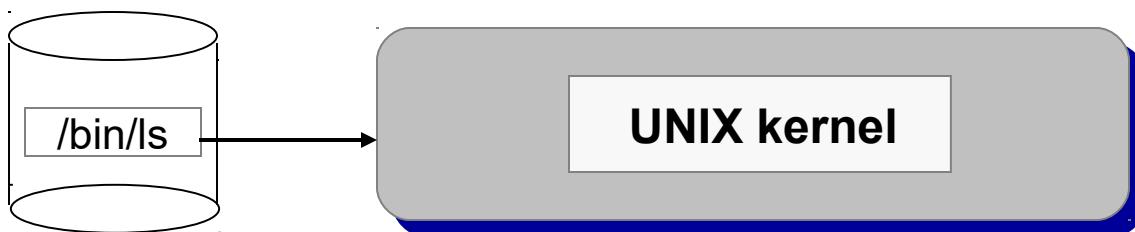
How execv Works (3)



```
char * argv[ ] = {"/bin/ls", 0};  
<...>  
int cpid = fork();  
if (cpid == 0) {  
    execv(argv[0], argv);  
    exit(0);  
}  
<parent code>  
wait(&cpid);
```

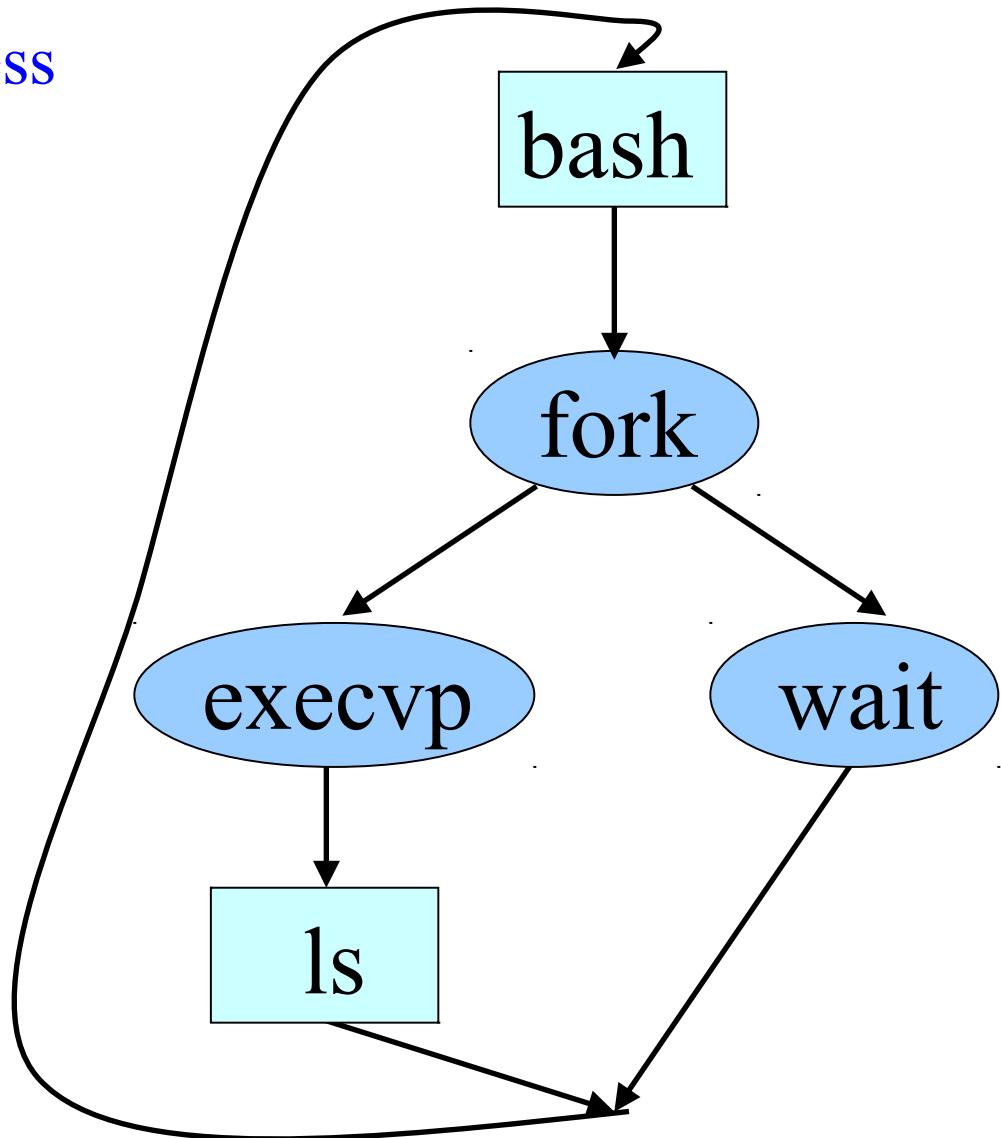
cpid = 26

```
<first line of ls>  
<...>  
<...>  
<...>  
exit(0);
```



Example: A Simple Shell

- Shell is the parent process
 - E.g., bash
- Parses command line
 - E.g., “ls -l”
- Invokes child process
 - Fork, execvp
- Waits for child
 - Wait



execv Example

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

char * argv[] = {"./bin/ls", "-l", 0};
int main()
{
    int pid, status;

    if ( (pid = fork()) < 0 )
    {
        printf("Fork error \n");
        exit(1);
    }
    if(pid == 0) { /* Child executes here */
        execv(argv[0], argv);
        printf("Exec error \n");
        exit(1);
    } else      /* Parent executes here */
        wait(&status);
    printf("Hello there! \n");
    return 0;
}
```

Note the NULL string
at the end

execv Example – Sample Output

- Sample output:

```
total 282
```

```
drwxr-xr-x 2 mdamian faculty 512 Jan 29 14:02 assignments  
-rw-r--r-- 1 mdamian faculty 3404 Jan 29 14:05 index.html  
drwxr-xr-x 2 mdamian faculty 512 Jan 28 15:02 notes
```

Hello there!

exec

- Same as execv, but takes the arguments of the new program as a list, not a vector:
- Example:

```
execl("/bin/ls", "/bin/ls", "-l", 0);
```

Note the NULL string at the end

```
char * argv[] = {"/bin/ls", "-l", 0};  
execv(argv[0], argv);
```

- execl is mainly used when the number of arguments is known in advance

General purpose process creation

- In the parent process:

```
int childPid;
char * const argv[ ] = {...};

main {
    childPid = fork();
    if(childPid == 0)
    {
        // I am child ...
        // Do some cleaning, close files
        execv(argv[0], argv);
    }
    else
    {
        // I am parent ...
        <code for parent process>
        wait(0);
    }
}
```

Combined fork/exec/wait

- Common combination of operations
 - Fork to create a new child process
 - Exec to invoke new program in child process
 - Wait in the parent process for the child to complete
- Single call that combines all three
 - int system(const char *cmd);
- Example

```
int main()
{
    system("echo Hello world");
}
```

Properties of fork / exec sequence

- In 99% of the time, we call `execv(...)` after `fork()`
 - the memory copying during `fork()` is useless
 - the child process will likely close open files and connections
 - overhead is therefore high
 - might as well combine both in one call (OS/2)
- `vfork()`
 - a system call that creates a process without creating an identical memory image
 - sometimes called “lightweight” fork
 - child process is understood to call `execv()` almost immediately

Variations of execv

- **execv**
 - Program arguments passed as an array of strings
- **execvp**
 - Extension of execv
 - Searches for the program name in the PATH environment
- **execl**
 - Program arguments passed directly as a list
- **execlp**
 - Extension of execv
 - Searches for the program name in the PATH environment

Summary

- **exec(v, vp, l, lp)**
 - Does NOT create a new process
 - Loads a new program in the image of the calling process
 - The first argument is the program name (or full path)
 - Program arguments are passed as a vector (**v, vp**) or list (**l, lp**)
 - Commonly called by a forked child
- **system:**
 - combines fork, wait, and exec all in one



High Performance Distributed Systems Lab

Linux Programming 程序

程序

- ◆ 每個程序位置以一個特定號碼指定



PID 2~32768

檢視程序

ps 命令可以顯示執行中的程序、另一位使用者正在執行的程序或所有執行中的程序

- ◆ -f : 全格式。
- ◆ -e : 顯示所有進程。
- ◆ a : 顯示終端上的所有進程，包括其他用戶的進程。
- ◆ x : 顯示沒有控制終端的進程
- ◆ TTY : 顯示啟動處理程序時所在的終端機。
- ◆ TIME : 顯示消耗掉的 CPU 時間。
- ◆ CMD : 顯示啟動處理程序的命令。

STAT，程序的狀態

- ◆ R (Running) : 該程式正在運作中；
- ◆ S (Sleep) : 該程式目前正在睡眠狀態 (idle)
，但可以被喚醒 (signal)。
- ◆ D : 不可被喚醒的睡眠狀態，通常這支程式可能在等待 I/O 的情況 (ex> 列印)
- ◆ T : 停止狀態 (stop)，可能是在工作控制
(背景暫停) 或除錯 (traced) 狀態；

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

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
    printf("Running ps with system\n");
    system("ps ax");
    printf("Done.\n");
    exit(0);
}
```

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0:00	/sbin/init
2	?	S	0:00	[kthreadd]
3	?	S	0:00	[migration/0]
4	?	S	0:00	[ksoftirqd/0]
5	?	S	0:00	[watchdog/0]
6	?	S	0:00	[migration/1]
7	?	S	0:00	[ksoftirqd/1]
8	?	S	0:00	[watchdog/1]
9	?	S	0:00	[migration/2]
10	?	S	0:00	[ksoftirqd/2]
11	?	S	0:00	[watchdog/2]
12	?	S	0:00	[migration/3]
13	?	S	0:00	[ksoftirqd/3]
14	?	S	0:00	[watchdog/3]
15	?	S	0:09	[events/0]
16	?	S	0:43	[events/1]
17	?	S	0:03	[events/2]
18	?	S	0:04	[events/3]
19	?	S	0:00	[cpuset]
				Done.
				29835 ? Ssl 2:18 /usr/sbin/mysqld
				31102 pts/1 T 0:04 vim York_2048_2.c
				31552 ? Ss 0:00 sshd: root@pts/2
				31626 pts/2 Ss 0:00 -bash
				31815 pts/1 S+ 0:29 vim -r York_2048_2.c



程序的替換

- ◆ 有一系列的相關函數，都是以 exec 為首。他們起動程式的方法不同還包含程式的參數。exec 函數會以一個新的處理程序取代目前的處理程序，新處理程序就是 path 或 file 參數所指定程式。

- ◆ `#include<unistd>`
- ◆ `int exec1(const char *path, const char *arg0, ...,
 (char *) 0);`
- ◆ `int execlp(const char *file, const char *arg0, ...
 ,
 (char *) 0);`
- ◆ `int execle(const char *path, const char *arg0, ...
 ,
 (char *) 0, const char *envp[]);`
- ◆ `int execv(const char *path, const char *argv[]);`
- ◆ `int execvp(const char *file, const char *argv[]);`
- ◆ `int execve(const char *path, const char *argv[],
 const char *envp[]);`

- ◆ path 參數表示你要啟動程序的名稱。
- ◆ arg 參數表示啟動程序所帶的參數，一般第一個參數為要執行命令名，不是帶路徑且 arg 必須以 NULL 結束。
- ◆ 帶 l 的 exec 函數：execl, execlp, execle，表示後邊的參數以可變參數的形式給出且都以一個空指針結束。
- ◆ 字尾為 p 的函數期不同之處在於他們會搜尋 PATH 環境變數來尋找新程式的執行檔。

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

int main(void)
{
    printf("entering main process---\n");
    execl("/bin/ls","ls","-l",NULL);
    printf("exiting main process ----\n");
    return 0;
}
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy      5976 Jul 12 22:54 a.out
-rw-r--r--. 1 zxy zxy       527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy       426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy       287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy       472 Jul 10 12:39 creathole.c
```



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

int main(void)
{
    printf("entering main process---\n");
    int ret;
    char *argv[] = {"ls", "-l", NULL};
    ret = execvp("ls", argv);
    if(ret == -1)
        perror("execl error");
    printf("exiting main process ----\n");
    return 0;
}
```

```
[zxy@test unixenv_c]$ cc execl.c
[zxy@test unixenv_c]$ ./a.out
entering main process---
total 104
-rwxrwxr-x. 1 zxy zxy      5976 Jul 12 22:54 a.out
-rw-r--r--. 1 zxy zxy       527 Jul 12 15:48 atexit02.c
-rw-r--r--. 1 zxy zxy       426 Jul 12 15:59 atexit03.c
-rw-r--r--. 1 zxy zxy       287 Jul 12 15:44 atexit.c
-rw-r--r--. 1 zxy zxy       472 Jul 10 12:39 creathole.c
```



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

int main(int argc, char *argv[])
{
    //char * const envp[] = {"AA=11", "BB=22",
NULL};
    printf("Entering main ...\\n");
    int ret;
    ret = execl("./hello", "hello", NULL);
    //execle("./hello", "hello", NULL, envp);
    if(ret == -1)
        perror("execl error");
    printf("Exiting main ...\\n");
    return 0;
```

```
[zxy@test unixenv_c]$ cc execle.c -o execle
[zxy@test unixenv_c]$ cc hello.c -o hello
[zxy@test unixenv_c]$ ./execle
Entering main ...
hello pid=4267
HOSTNAME=test
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=172.17.1.112 61324 22
SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib/qt-3.3
QTINC=/usr/lib/qt-3.3/include
SSH_TTY=/dev/pts/0
```

```
#include <unistd.h>
#include <stdio.h>
extern char** environ;

int main(void)
{
    printf("hello pid=%d\\n", getpid());
    int i;
    for (i=0; environ[i]!=NULL; ++i)
    {
        printf("%s\\n", environ[i]);
    }
    return 0;
```



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

int main()
{
    printf("Running ps with execvp\n");
    execvp("ps", "ps", "ax", 0);
    printf("Done.\n");
    exit(0);
}
```

- ◆ 程式在顯示第一個訊息後，呼叫 execvp，搜尋由 ps 程式的 PATH 環境變數所指定的路徑，隨後執行 ps 程式，取代原本的程式。
- ◆ 沒有 Done 的資訊出現。

複製程序

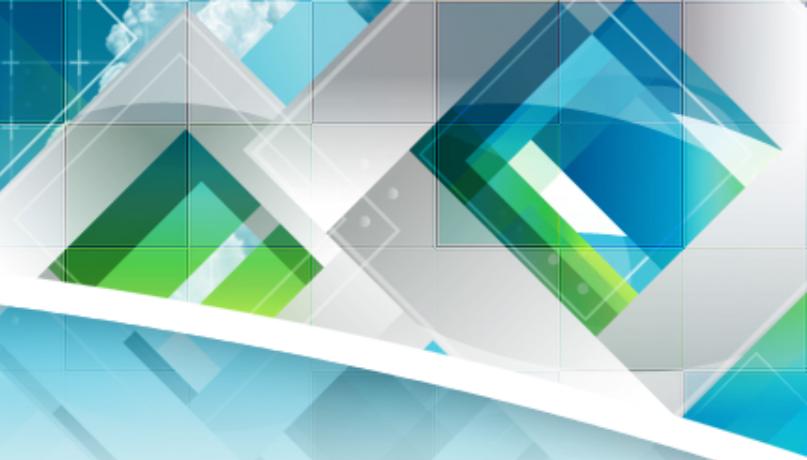
- ◆ 呼叫 fork 來建立新的程序，這個系統呼叫會複製目前的程序，在程序表中建立以目前執行程序有相同屬性的新程序。
- ◆ `#include<sys/types.h>`
- ◆ `#include<unisted.h>`
- ◆ `pid_t fork(void);`

```
◆ pid_t new_pid ;  
◆ new_pid = fork( );  
◆ Switch(new_pid) {  
◆ case -1 : //error  
◆           break;  
◆ case 0 : //we are child  
◆           break;  
default      : //we are parent  
           break;  
}
```

```
int main()
{
    pid_t pid;
    char *message;
    int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
        case -1:
            perror("fork failed");
            exit(1);
        case 0:
            message = "This is the child";
            n = 5;
            break;
        default:
            message = "This is the parent";
            n = 3;
            break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
    exit(0);
}
```



呼叫 fork 時，這個程式會變成兩個獨立的處理程序父處理程序的辨識方式是經由 fork 回傳的的非零值

```
root@cuda04:~/Frank/ex/ch10# ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
root@cuda04:~/Frank/ex/ch10# This is the child
This is the child
```

等待程序

- ◆ `#include<sys/types.h>`
- ◆ `#include<sys/wait.h>`
- ◆ `pid_t wait(int *stat_loc);`
- ◆ `wait` 系統呼叫會讓父處理程序暫停，直到他的子處理程序結束。這個函數會回傳子處理程序的 PID。狀態資訊，讓父處理程序可以判斷子處理程序的結束狀態，也就是 `exit` 回傳值。
- ◆ `loc` 的值並不是 `null` 指標，所以狀態資訊會被寫入目前所指的位置。

- ◆ **WIFEXITED**(stat_val) : 子程序正常終止傳回非零值。
- ◆ **WEXITSTATUS**(stat_val): WIFSIGNALED 非零，傳回子程序離開碼。
- ◆ **WIFSIGNALED**(stat_val): 子程序因漏失訊息而終止則為非零。
- ◆ **WTERMSIG**(stat_val) : WIFSIGNALED 非零則傳回訊息號碼。
- ◆ **WIFSTOPPED**(stat_val) : 子程序停止則為非零。
- ◆ **WSTOPSIG**(stat_val) : WIFSTOPPED 非零則傳回訊息號碼。

```

int main()
{
    pid_t pid;
    char *message;
    int n;
    int exit_code;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
    case -1:
        exit(1);
    case 0:
        message = "This is the child";
        n = 5;
        exit_code = 37;
        break;
    default:
        message = "This is the parent";
        n = 3;
        exit_code = 0;
        break;
    }

    for(; n > 0; n--) {
        puts(message);
        sleep(1);
    }
}

This section of the program waits for the child process to finish. */

if(pid!=0) {
    int stat_val;
    pid_t child_pid;

    child_pid = wait(&stat_val);

    printf("Child has finished: PID = %d\n", child_pid);
    if(WIFEXITED(stat_val))
        printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
    else
        printf("Child terminated abnormally\n");
}
exit (exit_code);
}

```



```
root@cuda04:~/Frank/ex/ch10# ./wait
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 3744
Child exited with code 37
```

- ◆ 父處理程序從 fork 取得一個非零值，隨後使用 wait 暫停本身的程式，等待子處理程序回復狀態資訊。

輸出入的轉向

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
~
```

```
root@cuda04:~/Frank/ex/ch10# ./upper
hello
HELLO
```



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

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

    if(argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
        exit(1);
    }

    filename = argv[1];

    if(!freopen(filename, "r", stdin)) {
        fprintf(stderr, "could not redirect stdin to file %s\n", filename);
        exit(2);
    }

    execl("./upper", "upper", 0);

    perror("could not exec ./upper");
    exit(3);
}
```

```
root@cuda04:~/Frank/ex/ch10# ./useupper file.txt  
THIS IS THE FILE, FILE.TXT; IT IS ALL LOWER CASE.
```

- ◆ useupper 使用 freopen 來關閉標準輸入，並使用程式引數來關聯檔案資料流 stdin ，呼叫 exec 以上面的程式來取代執行中的程序。

訊息名稱，signal.h

- ◆ SIGABORT * 程序停止
- ◆ SIGALRM 警示
- ◆ SIGFPE * 浮點數例外
- ◆ SIGHUP 掛斷
- ◆ SIGILL * 非法指令
- ◆ SIGINT 終端機插斷

Ctrl-C (in older Unixes, DEL) sends an INT signal (SIGINT); by default, this causes the process to terminate.

Ctrl-Z sends a TSTP signal (SIGTSTP); by default, this causes the process to suspend execution.

Ctrl-\ sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and dump core.

- ◆ `#include<signal.h>`
- ◆ `void(*signal(int sig, void (*func) (int)))(int) ;`
- ◆ 兩個參數：sig 與 func，sig 指名要處理的訊號類型，func 描述與信號關聯的動作。
- ◆ SIG_IGN: 表示忽略該信號，執行 signal() 後，進程會忽略類型為 sig 的信號。
- ◆ SIG_DFL: 這個符號表示恢復系統對信號的默認處理。

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
    (void) signal(SIGINT, SIG_DFL);
}

int main()
{
    (void) signal(SIGINT, ouch);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

```
root@cuda04:~/Frank/ex/ch10# ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^COUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
```

- ◆ `int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);`
`sig`: 要處理的訊號，若 `act` 指針非空， 則根據 `act` 修改該信號的處理動作。若 `oact` 指針非空， 則通過 `oact` 傳出該信號原來的處理動作。
- ◆ `Void(*)(int) sa_handler`
- ◆ `sigset_t sa_mask`
- ◆ `int sa_flags`
- ◆ `sa_handler` 代表新的信號處理。
- ◆ `sa_mask` 用來設置在處理該信號時暫時將 `sa_mask` 指定的信號集擋置。
- ◆ `sa_flags` 用來設置信號處理的其他相關操作。

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

void ouch(int sig)
{
    printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
    struct sigaction act;

    act.sa_handler = ouch;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    sigaction(SIGINT, &act, 0);

    while(1) {
        printf("Hello World!\n");
        sleep(1);
    }
}
```

```
root@cuda04:~/Frank/ex/ch10# ./ctrlc2
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
^COUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^COUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
^C\離開
```

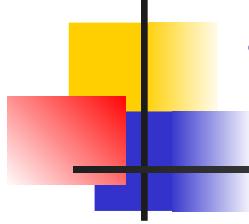




Thanks for your listening !!

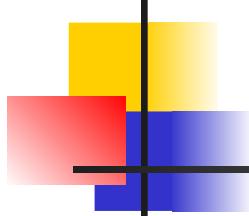


High Performance
Distributed Systems Lab



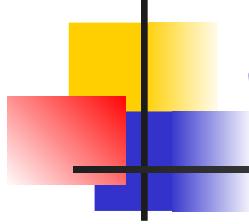
Today's topic

- Pthread
 - Some materials and figures are obtained from the POSIX threads Programming tutorial at
<https://computing.llnl.gov/tutorials/pthreads>



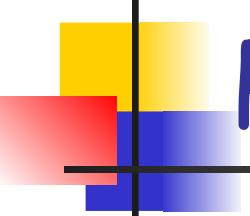
What is a Thread?

- OS view: A thread is an independent stream of instructions that can be scheduled to run by the OS.
- Software developer view: a thread can be considered as a “procedure” that runs independently from the main program.
 - Sequential program: a single stream of instructions in a program.
 - Multi-threaded program: a program with multiple streams
 - Multiple threads are needed to use multiple cores/CPUs



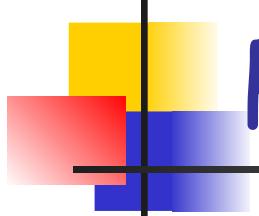
Example multithread programs?

- Computer games
 - each thread controls the movement of an object.
- Scientific simulations
 - Hurricane movement simulation: each thread simulates the hurricane in a small domain.
 - Molecular dynamic: each thread simulates a subset of particulars.
 -
- Web server
 - Each thread handles a connection.
-



Process and Thread

- **Process context**
 - Process ID, process group ID, user ID, and group ID
 - Environment
 - Working directory.
 - Program instructions
 - Registers (including PC)
 - Stack
 - Heap
 - File descriptors
 - Signal actions
 - Shared libraries
 - Inter-process communication tools
- Two parts in the context: self-contained domain (protection) and execution of instructions.



Process and Thread

- What are absolutely needed to support a stream of instructions, given the process context?
 - Process ID, process group ID, user ID, and group ID
 - Environment
 - Working directory.
 - Program instructions
 - Registers (including PC)
 - Stack
 - Heap
 - File descriptors
 - Signal actions
 - Shared libraries
 - Inter-process communication tools

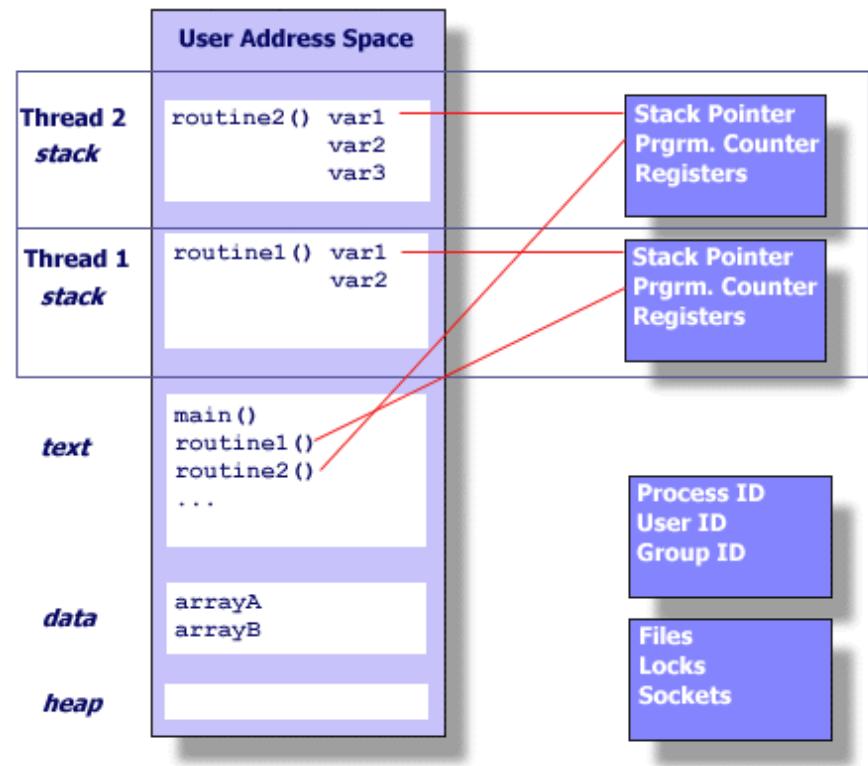
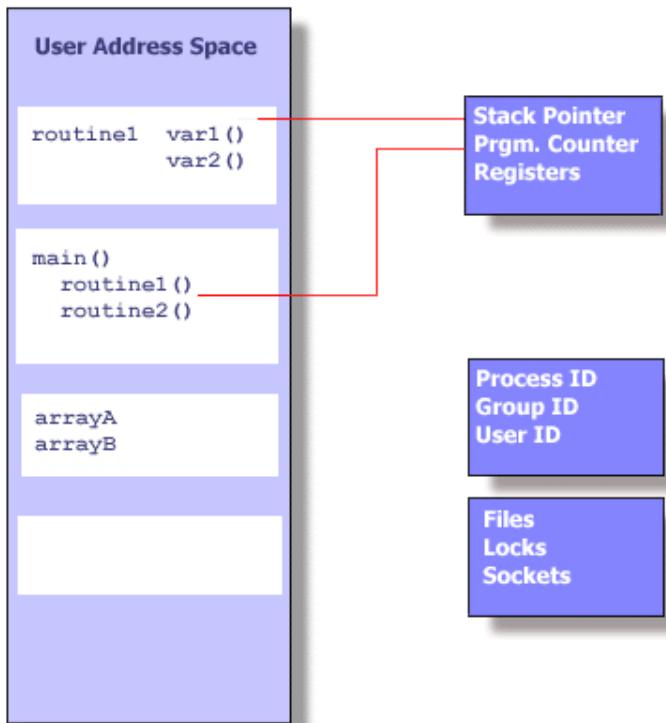
Process and Thread

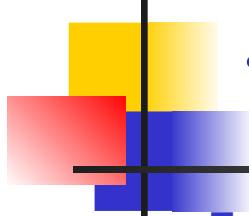
stack

text

data

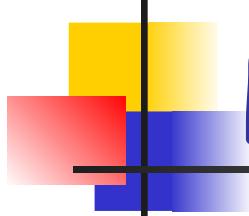
heap





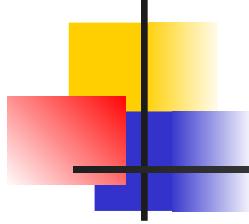
Threads...

- Exist within processes
- Die if the process dies
- Use process resources
- Duplicate only the essential resources for OS to schedule them independently
- Each thread maintains
 - Stack
 - Registers
 - Scheduling properties (e.g. priority)
 - Set of pending and blocked signals (to allow different react differently to signals)
 - Thread specific data



Pthreads...

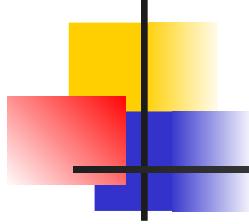
- Hardware vendors used to implement proprietary versions of threads
 - Thread programs are not portable
- Pthreads = POSIX threads, specified in IEEE POSIX 1003.1c (1995)



Advantages of Threads

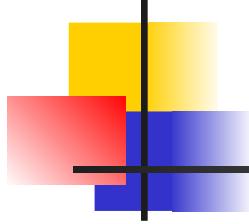
- Light-weight
 - Lower overhead for thread creation
 - Lower Context Switching Overhead
 - Fewer OS resources

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.1	60.1	9.0	0.7	0.2	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.8	27.7	1.8	0.7	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.1	48.6	47.2	2.0	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	55.0	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.3	0.7



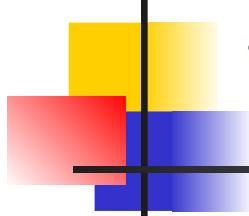
Advantages of Threads

- **Shared State**
 - Don't need IPC-like mechanism to communicate between threads of same process



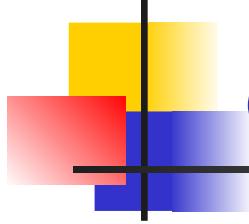
Disadvantages of Threads

- Shared State!
 - Global variables are shared between threads.
Accidental changes can be fatal.
- Many library functions are not **thread-safe**
 - Library Functions that return pointers to static internal memory. E.g. **gethostbyname()**
- Lack of robustness
 - Crash in one thread will crash the entire process.



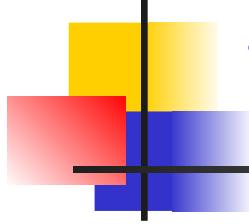
The Pthreads API

- Three types of routines:
 - Thread management: create, terminate, join, and detach
 - Mutexes: mutual exclusion, creating, destroying, locking, and unlocking mutexes
 - Condition variables: event driven synchronization.
- Mutexes and condition variables are concerned about synchronization.
- Why not anything related to inter-thread communication?
- The concept of opaque objects pervades the design of the API.



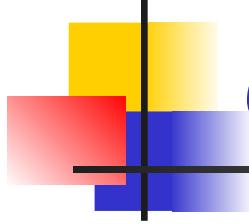
The Pthreads API naming convention

Routine Prefix	Function
Pthread_	General pthread
Pthread_attr_	Thread attributes
Pthread_mutex_	mutex
Pthread_mutexattr	Mutex attributes
Pthread_cond_	Condition variables
Pthread_condaddr	Conditional variable attributes
Pthread_key_	Thread specific data keys



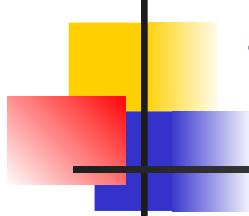
Thread management routines

- Creation: `pthread_create`
- Termination:
 - Return
 - `Pthread_exit`
 - Can we still use `exit`?
- Wait (parent/child synchronization):
`pthread_join`
- Pthread header file `<pthread.h>`
- Compiling pthread programs: `gcc -lpthread aaa.c`



Creation

- Thread equivalent of **fork()**
- ```
int pthread_create(
 pthread_t * thread,
 pthread_attr_t * attr,
 void * (*start_routine)(void *),
 void * arg
);
```
- Returns 0 if OK, and non-zero (> 0) if error.
- Parameters for the routines are passed through **void \* arg**.
  - What if we want to pass a structure?



# Termination

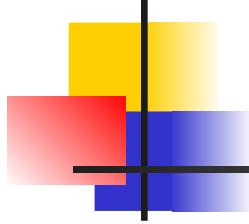
---

## Thread Termination

- `void pthread_exit(void * status)`

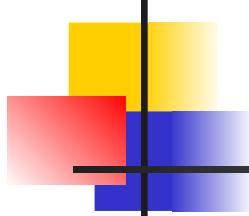
## Process Termination

- `exit()`
- `main()`



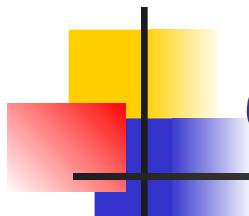
# Waiting for child thread

- `int pthread_join( pthread_t tid, void **status)`
- Equivalent of `waitpid()` for processes



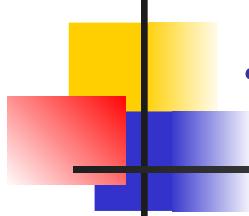
# Detaching a thread

- The detached thread can act as daemon thread
- The parent thread doesn't need to wait
- `int pthread_detach(pthread_t tid)`
- Detaching self :  
`pthread_detach(pthread_self())`



# Some multi-thread program examples

- A multi-thread program example:  
Example1.c
- Making multiple producers:  
example2.c
  - What is going on in this program?

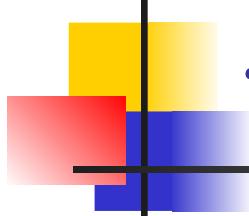


# Matrix multiply and threaded matrix multiply

- Matrix multiply:  $C = A \times B$

$$C[i, j] = \sum_{k=1}^N A[i, k] \times B[k, j]$$

$$\begin{pmatrix} C[0,0], & C[0,1], \dots, & C[0,N-1] \\ C[1,0], & \boxed{C[1,1]}, \dots, & C[1,N-1] \\ \dots & \dots & \dots \\ C[N-1,0], C[N-1,1], \dots, C[N-1,N-1] \end{pmatrix} = \begin{pmatrix} A[0,0], & A[0,1], \dots, & A[0,N-1] \\ \boxed{A[1,0]}, & A[1,1], \dots, & A[1,N-1] \\ \dots & \dots & \dots \\ A[N-1,0], A[N-1,1], \dots, A[N-1,N-1] \end{pmatrix} \times \begin{pmatrix} B[0,0], & B[0,1], \dots, & B[0,N-1] \\ B[1,0], & \boxed{B[1,1]}, \dots, & B[1,N-1] \\ \dots & \dots & \dots \\ B[N-1,0], \boxed{B[N-1,1]}, \dots, B[N-1,N-1] \end{pmatrix}$$



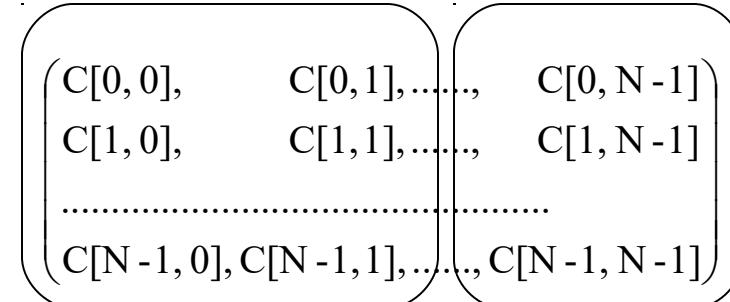
# Matrix multiply and threaded matrix multiply

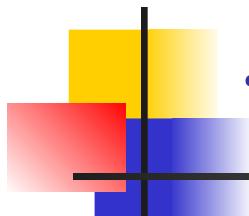
- Sequential code:

```
For (i=0; i<N; i++)
 for (j=0; j<N; j++)
 for (k=0; k<N; k++) C[I, j] = C[I, j] + A[I, k] * A[k, j]
```

- Threaded code program

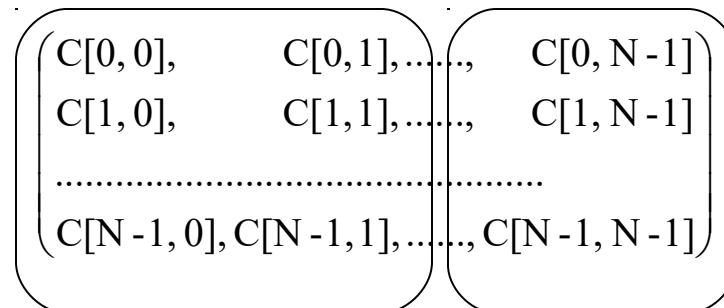
- Do the same sequential operation, different threads work on different part of the C array. How to decide who does what? Need three parameters: N, nthreads, myid

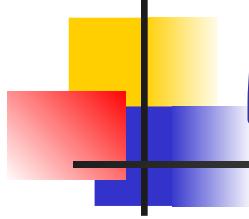




# Matrix multiply and threaded matrix multiply

- Threaded code program
  - From N, nthreads, myid
    - I am responsible for sub-array $C[0..N-1][N/Nthreads*myrank .. N/Nthreads*(myrank+1))$
  - The calculation of  $c[I,j]$  does not depend on other C term. Mm\_pthread.c.





# PI calculation

$$PI = \lim_{n \rightarrow \infty} \left( \frac{1}{n} \sum_{i=1}^n \frac{4.0}{1.0 + \left( \frac{i - 0.5}{n} \right)^2} \right)$$

- Sequential code: pi.c
- Multi-threaded version: pi\_pthread.c
  - Again domain partition based on N, nthreads, and myid.

High

Performance

Distributed

System

**KUAS – High Performance  
Distributed System**

Linux Programming - Pthread

Reporter: Po-Sen Wang

# Pthread Function(1/3)

- `#include <pthread.h>`
- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);`  
**//create a thread**
- `void pthread_exit(void *retval);`  
**//terminate a thread**
- `int pthread_join(pthread_t th, void **thread_return);`  
**//wait for thread termination**

# Pthread Function(2/3)

- `int pthread_create(pthread_t *thread,  
pthread_attr_t *attr, void *(*start_routine)(void *),  
void *arg);`
- `pthread_t *thread`: thread 的識別字
- `pthread_attr_t *attr`: thread 的屬性。設定為 `NULL`  
表示使用預設
- `void *(*start_routine)(void *)`: thread 要執行的  
function
- `void *arg`: 傳遞給 thread 的參數

# Pthread Function(3/3)

- `void pthread_exit(void *retval);`
  - `void *retval`: thread 結束時回傳的變數
- `int pthread_join(pthread_t th, void **thread_return);`
  - `pthread_t th`: thread 識別字
  - `void **thread_return`: 接收 `pthread_exit` 傳回的變數

## Example 1 (1/3)

- gcc thread1.c -o thread1 -L/usr/lib/nptl –lpthread
- thread1.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
 int res;
 pthread_t a_thread;
 void *thread_result;
 res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
 if (res != 0) {
 perror("Thread creation failed");
 exit(EXIT_FAILURE);
 }
}
```

## Example 1 (2/3)

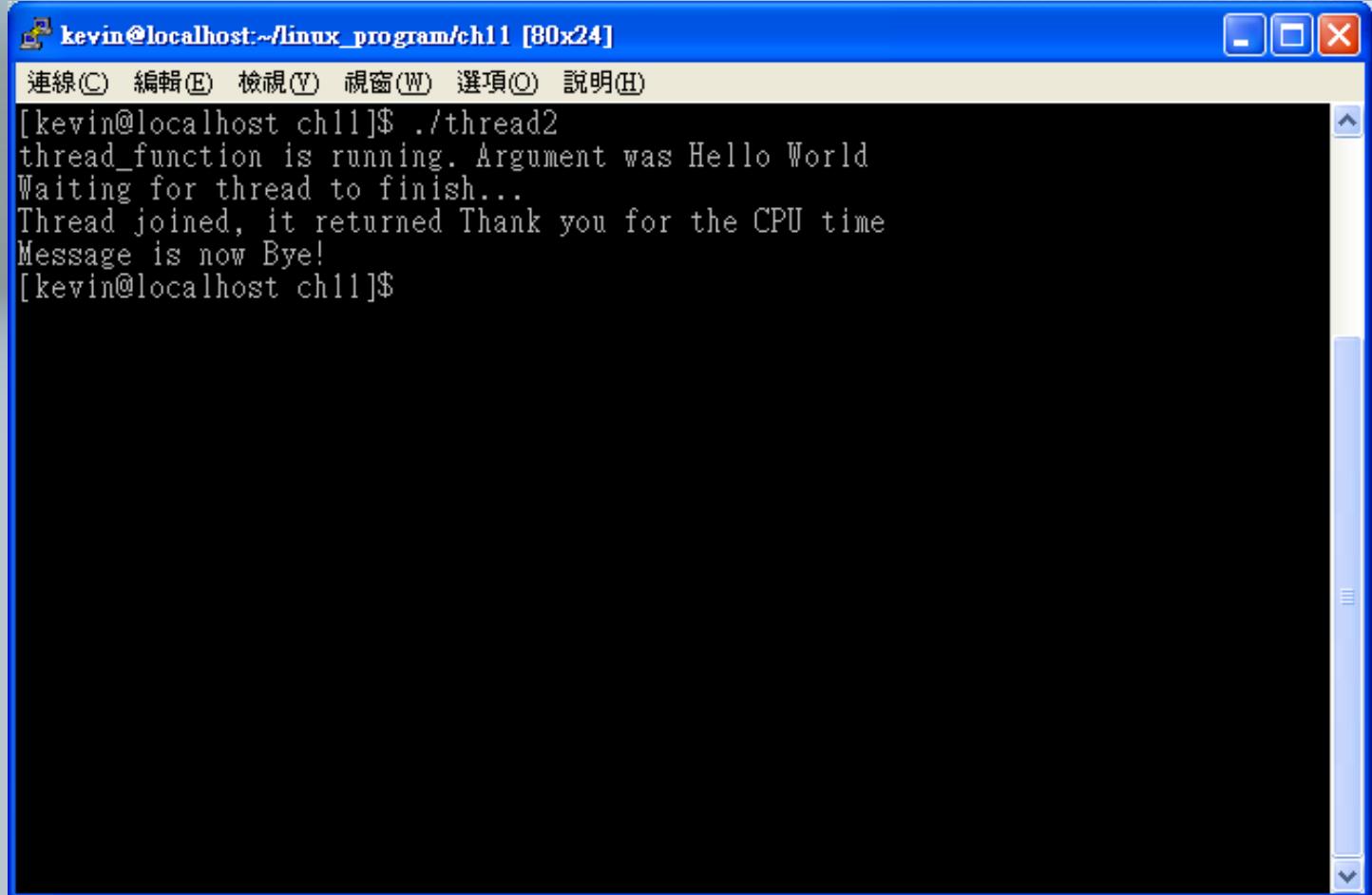
## ■ thread1.c

```
printf("Waiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
if (res != 0) {
 perror("Thread join failed");
 exit(EXIT_FAILURE);
}
printf("Thread joined, it returned %s\n", (char *)thread_result);
printf("Message is now %s\n", message);
exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
 printf("thread_function is running. Argument was %s\n", (char *)arg);
 sleep(3);
 strcpy(message, "Bye!");
 pthread_exit("Thank you for the CPU time");
}
```

## Example 1 (3/3)

## ■ thread1.c



```
kevin@localhost:~/linux_program/ch11 [80x24]
連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)
[kevin@localhost ch11]$./thread2
thread_function is running. Argument was Hello World
Waiting for thread to finish...
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
[kevin@localhost ch11]$
```

# Synchronization – Using Semaphore (1/3)

- `#include <semaphore.h>`
- `int sem_init(sem_t *sem, int pshared, unsigned int value);`  
**//create a semaphore**
- `int sem_wait(sem_t *sem);`  
**//lock a semaphore**
- `int sem_post(sem_t *sem);`  
**//unlock a semaphore**
- `int sem_destroy(sem_t *sem);`  
**//delete a semaphore**

# Synchronization – Using Semaphore (2/3)

- `int sem_init(sem_t *sem, int pshared, unsigned int value);`
  - `sem_t *sem`: semaphore 識別字
  - `int pshared`: 設定為 0 表示僅供目前的 process 及其 thread 使用。非零值表示此 semaphore 與其他 process 共用
  - `unsigned int value`: semaphore 的初始值

# Synchronization – Using Semaphore (3/3)

- `int sem_wait(sem_t *sem);`
  - 若 semaphore 為非零值，則 semaphore 值減 1；若 semaphore 為 0，則呼叫此 function 的 thread 會被 block，直到 semaphore 值不為零。
  
- `int sem_post(sem_t *sem);`
  - 對 semaphore 值加 1。

# Example 2 – Using Semaphore (1/4)

## ■ thread3.c

```
kevin@localhost:~/linux_program/ch11 [80x24]
[kevin@localhost ch11]$./thread4
Input some text. Enter 'end' to finish
hello
You input 5 characters
hihi
You input 4 characters
end

Waiting for thread to finish...
Thread joined
[kevin@localhost ch11]$
```

# Example 2 – Using Semaphore (2/4)

## ■ thread3.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
 int res;
 pthread_t a_thread;
 void *thread_result;

 res = sem_init(&bin_sem, 0, 0);
 res = pthread_create(&a_thread, NULL, thread_function, NULL);
```

# Example 2 – Using Semaphore (3/4)

## ■ thread3.c

```
printf("Input some text. Enter 'end' to finish\n");
while(strncmp("end", work_area, 3) != 0) {
 fgets(work_area, WORK_SIZE, stdin);
 sem_post(&bin_sem);
}
printf("\nWaiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
if (res != 0) {
 perror("Thread join failed");
 exit(EXIT_FAILURE);
}
printf("Thread joined\n");
sem_destroy(&bin_sem);
exit(EXIT_SUCCESS);
}
```

# Example 2 – Using Semaphore (4/4)

- thread3.c

```
void *thread_function(void *arg) {
 sem_wait(&bin_sem);
 while(strncmp("end", work_area, 3) != 0) {
 printf("You input %d characters\n", strlen(work_area) - 1);
 sem_wait(&bin_sem);
 }
 pthread_exit(NULL);
}
```

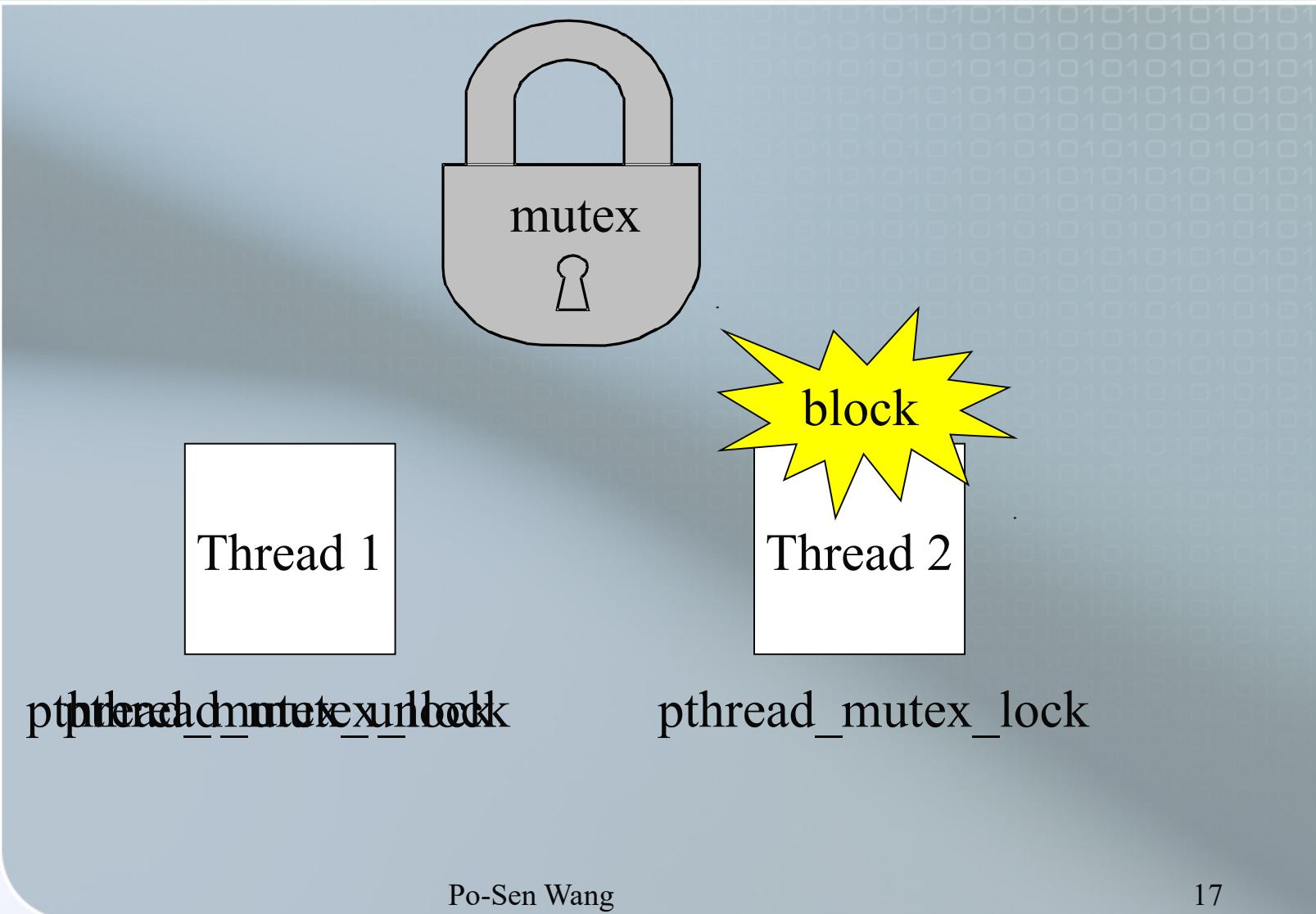
# Synchronization – Using Mutex (1/3)

- `#include <pthread.h>`
- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`  
**//create a mutex**
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`  
**//lock a mutex**
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`  
**//unlock a mutex**
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`  
**//delete a mutex**

# Synchronization – Using Mutex (2/3)

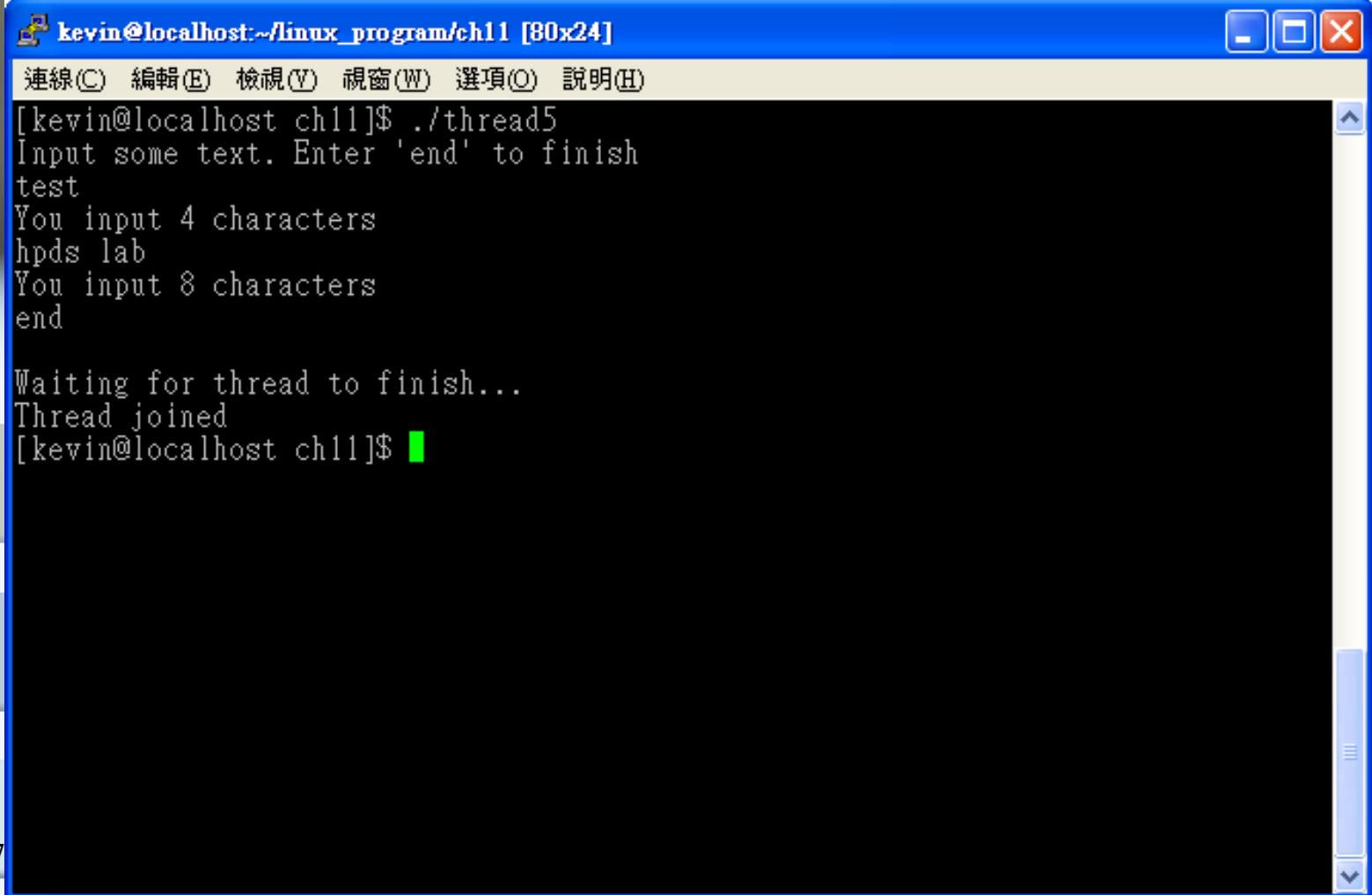
- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);`
  - `pthread_mutex_t *mutex`: mutex 識別字
  - `const pthread_mutexattr_t *mutexattr`:  
mutex 的屬性。設定為 `NULL` 表示使用預設。

# Synchronization – Using Mutex (2/3)



# Example 3 – Using Mutex(1/4)

## ■ thread4.c



```
kevin@localhost:~/linux_program/ch11 [80x24]
連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)
[kevin@localhost ch11]$./thread5
Input some text. Enter 'end' to finish
test
You input 4 characters
hpds lab
You input 8 characters
end

Waiting for thread to finish...
Thread joined
[kevin@localhost ch11]$
```

# Example 3 – Using Mutex(2/4)

## ■ thread4.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
 int res;
 pthread_t a_thread;
 void *thread_result;
 res = pthread_mutex_init(&work_mutex, NULL);
 res = pthread_create(&a_thread, NULL, thread_function, NULL);
```

# Example 3 – Using Mutex(3/4)

## thread4.c

```
pthread_mutex_lock(&work_mutex);
printf("Input some text. Enter 'end' to finish\n");
while(!time_to_exit) {
 fgets(work_area, WORK_SIZE, stdin);
 pthread_mutex_unlock(&work_mutex);
 while(1) {
 pthread_mutex_lock(&work_mutex);
 if (work_area[0] != '\0') {
 pthread_mutex_unlock(&work_mutex);
 sleep(1);
 }
 else {
 break;
 }
 }
 pthread_mutex_unlock(&work_mutex);
}
printf("\nWaiting for thread to finish...\n");
res = pthread_join(a_thread, &thread_result);
printf("Thread joined\n");
pthread_mutex_destroy(&work_mutex);
exit(EXIT_SUCCESS);
```

# Example 3 – Using Mutex(4/4)

## ■ thread4.c

```
void *thread_function(void *arg) {
 sleep(1);
 pthread_mutex_lock(&work_mutex);
 while(strncmp("end", work_area, 3) != 0) {
 printf("You input %d characters\n", strlen(work_area) -1);
 work_area[0] = '\0';
 pthread_mutex_unlock(&work_mutex);
 sleep(1);
 pthread_mutex_lock(&work_mutex);
 while (work_area[0] == '\0') {
 pthread_mutex_unlock(&work_mutex);
 sleep(1);
 pthread_mutex_lock(&work_mutex);
 }
 }
 time_to_exit = 1;
 work_area[0] = '\0';
 pthread_mutex_unlock(&work_mutex);
 pthread_exit(0);
}
```

# Cancellation(1/2)

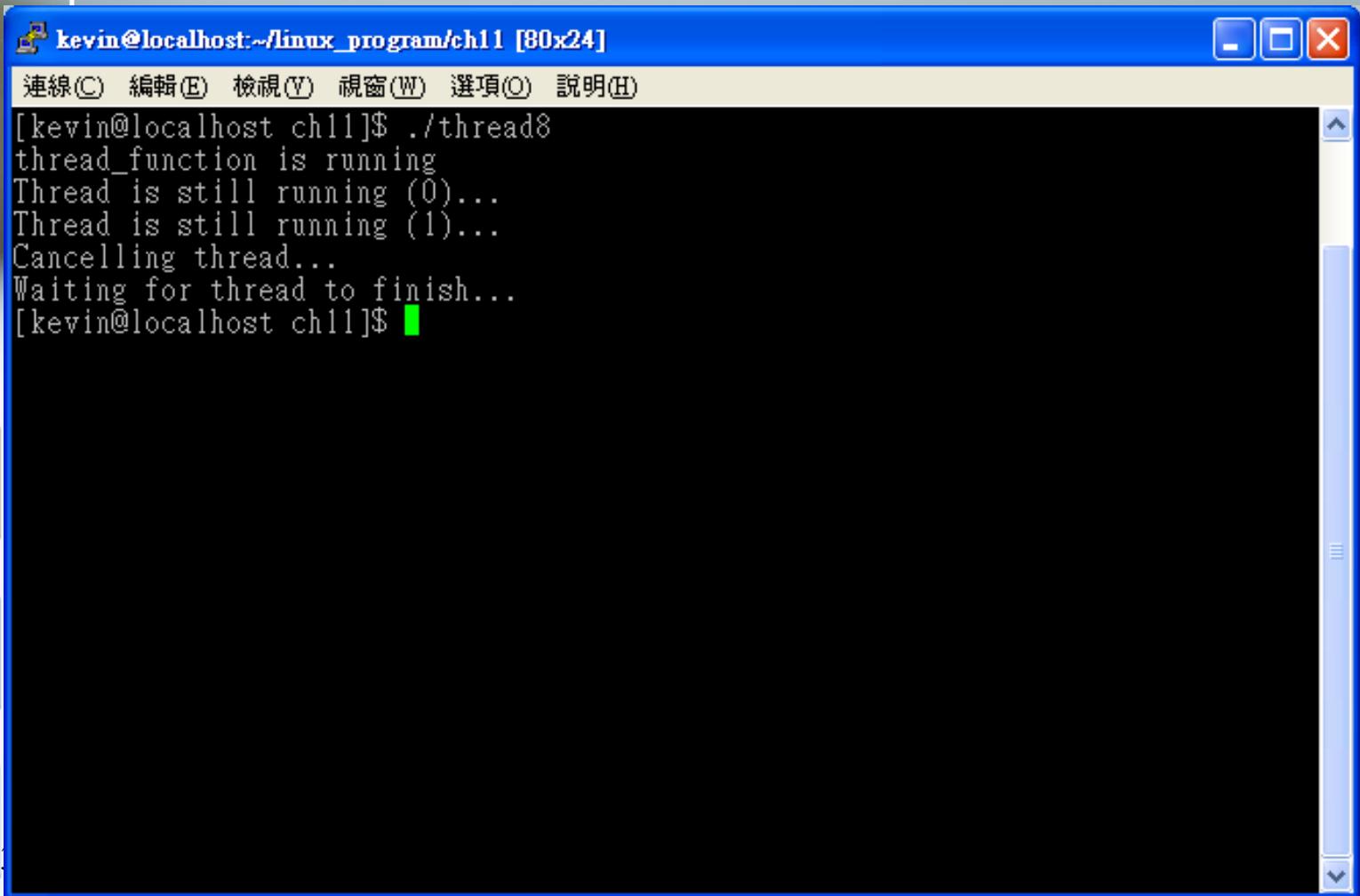
- `#include <pthread.h>`
- `int pthread_cancel(pthread_t thread);`  
**//cancel a thread**
- `int pthread_setcancelstate(int state, int *oldstate);`  
**//set cancellation state**
- `int pthread_setcanceltype(int type, int *oldtype);`  
**//set cancellation type**

## Cancellation(2/2)

- `int pthread_setcancelstate(int state, int *oldstate);`
  - `int state`: 設定為 `PTHREAD_CANCEL_ENABLE` 即表示允許取消 `thread` 的請求；  
設定為 `PTHREAD_CANCEL_DISABLE` 即表示忽略取消的請求。
  - `int *oldstate`: 此指標指向前一個狀態
- `int pthread_setcanceltype(int type, int *oldtype);`
  - `int type`: 設定為 `PTHREAD_CANCEL_ASYNCHRONOUS` 則立即取消 `thread`；  
設定為 `PTHREAD_CANCEL_DEFERRED` 則會遇到取消點才會取消 `thread`。取消點即是下列函數：  
`pthread_join`、`pthread_cond_wait`、`pthread_testcancel` 等
  - `int *oldtype`: 此指標指向前一個型態

## Example 4 (1/3)

## ■ thread7.c



The screenshot shows a terminal window titled "kevin@localhost:~/linux\_program/ch11 [80x24]". The window contains the following text:

```
[kevin@localhost ch11]$./thread8
thread_function is running
Thread is still running (0)...
Thread is still running (1)...
 Cancelling thread...
Waiting for thread to finish...
[kevin@localhost ch11]$ █
```

## Example 4 (2/3)

## ■ thread7.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

int main() {
 int res;
 pthread_t a_thread;
 void *thread_result;

 res = pthread_create(&a_thread, NULL, thread_function, NULL);
 sleep(2);
 printf("Cancelling thread...\n");
 res = pthread_cancel(a_thread);
 printf("Waiting for thread to finish...\n");
 res = pthread_join(a_thread, &thread_result);
 exit(EXIT_SUCCESS);
}
```

## Example 4 (3/3)

## ■ thread7.c

```
void *thread_function(void *arg) {
 int i, res, j;
 res = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
 res = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
 printf("thread_function is running\n");
 for(i = 0; i < 10; i++) {
 printf("Thread is still running (%d)...\n", i);
 sleep(1);
 }
 pthread_exit(0);
}
```

# Example – Multi-Thread (1/4)

## ■ thread8a.c

kevin@localhost:~/linux\_program/ch11 [80x24]

連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)

```
[kevin@localhost ch11]$./thread9a
thread_function is running. Argument was 0
thread_function is running. Argument was 1
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 1
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
All done
[kevin@localhost ch11]$
```

# Example – Multi-Thread (2/4)

## ■ thread8a.c

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

#include <pthread.h>

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {

 int res;
 pthread_t a_thread[NUM_THREADS];
 void *thread_result;
 int lots_of_threads;
```

# Example – Multi-Thread (3/4)

## ■ thread8a.c

```
for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++)
 res = pthread_create(&(a_thread[lots_of_threads]), NULL,
 thread_function, (void *) lots_of_threads);
printf("Waiting for threads to finish...\n");
for(lots_of_threads = NUM_THREADS - 1; lots_of_threads >= 0; lots_of_threads--) {
 res = pthread_join(a_thread[lots_of_threads], &thread_result);
 if (res == 0) {
 printf("Picked up a thread\n");
 } else {
 perror("pthread_join failed");
 }
}

printf("All done\n");

exit(EXIT_SUCCESS);
}
```

# Example – Multi-Thread (4/4)

## ■ thread8a.c

```
void *thread_function(void *arg) {
 int my_number = (int)arg;
 int rand_num;

 printf("thread_function is running. Argument was %d\n", my_number);
 rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
 sleep(rand_num);
 printf("Bye from %d\n", my_number);

 pthread_exit(NULL);
}
```

# Condition Variables

- `pthread_cond_init (condition,attr)`  
`pthread_cond_destroy (condition)`
- `pthread_condattr_init (attr)`
- `pthread_condattr_destroy (attr)`

# Condition Variables

- Example :condition\_variable

# Homework 1

- 撰寫一個程式，create 10 threads，共同計算  
計算  $1+2+3+4+\dots+10000$
- Thread 0, 計算 1~1000 總和，並將總和累計  
至共用變數 total
- Thread 1, 計算 1001~2000 總和，並將總和  
累計至共用變數 total
- 以此類推
- MAIN thread join 10 個 child thread 後，列印  
出總和 total

# Homework 2

- 定義一結構 barrier 如下：
  - count\_mutex
  - cond\_var
  - count // the number of threads that has arrived at the barrier
  - limit // the number of threads that will arrive at the barrier
- void barrier\_init( struct barrier \*, int num);
- void barrier\_arrive(struct barrier \*);
- 寫一個程式 create 10 threads
- 每一個 thread 都執行下列 function

# Homework 2

- work(void\* arg)
- { int myid= (int) arg;
- int i;
- for(i=0 ; i<3 ; i++){
- printf("Thread %d echo in %d iteration\n", myid, i);
- barrier\_arrive(&bar);
- }



# Signals



# Context of Remaining Lectures

Second half of COS 217 takes 2 tours:

1. “Language levels” tour

C → assembly language → machine language

Illustrated by assembly language asgt, buffer overrun asgt

2. “Service levels” tour

C → C standard library → operating system (OS)

Illustrated by heap manager asgt, shell asgt

The 2 remaining lectures flesh out the “service levels” tour



# Goals of Remaining Lectures

Two fundamental questions:

- Q1: How does the **OS** communicate to an **application process**?
- Q2: How does an **application process** communicate to the **OS**?

This lecture: Q1

Next lecture: Q2

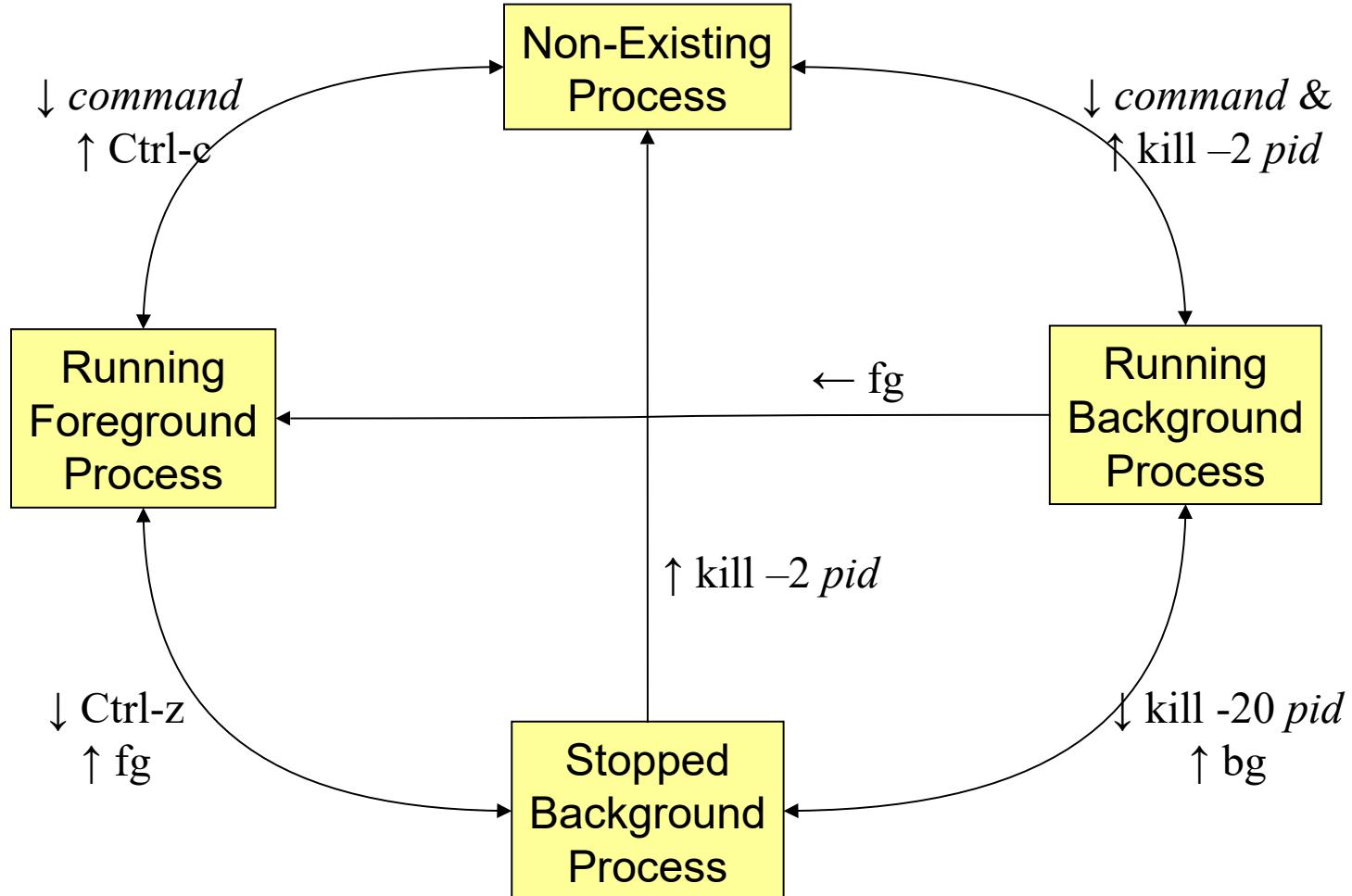


# Outline

1. **UNIX Process Control**
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers



# UNIX Process Control





# UNIX Process Control

[Demo of UNIX process control using infloop.c]



# Process Control Implementation

Exactly what happens when you:

- Type Ctrl-c?
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a 2/SIGINT **signal**
- Type Ctrl-z?
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a 20/SIGTSTP **signal**
- Issue a “kill –sig *pid*” command?
  - OS sends a *sig* **signal** to the process whose id is *pid*
- Issue a “fg” or “bg” command?
  - OS sends a 18/SIGCONT **signal** (and does some other things too!) 7



# Outline

1. UNIX Process Control
2. **Signals**
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers



# Signals

Q1: How does the **OS** communicate to an **application process**?

A1: **Signals**

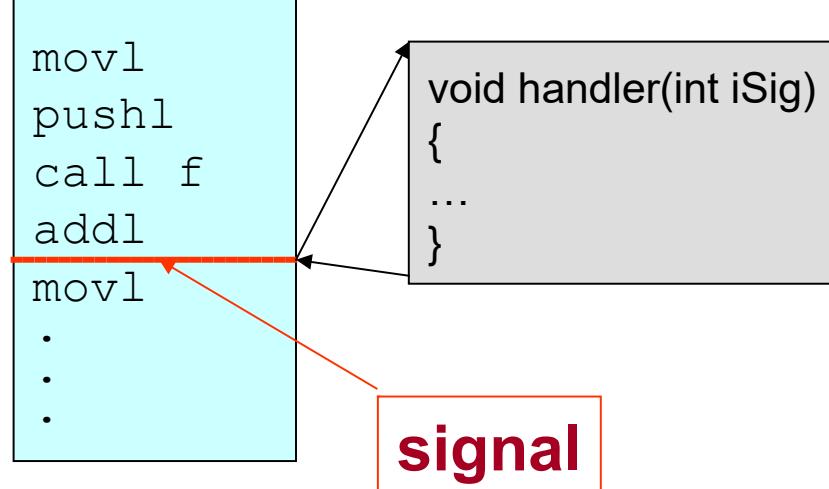


# Definition of Signal

## Signal: A notification of an event

- Event gains attention of the OS
- OS stops the application process immediately, sending it a signal
- **Signal handler** executes to completion
- Application process resumes where it left off

Process





# Examples of Signals

## User types Ctrl-c

- Event gains attention of OS
- OS stops the application process immediately, sending it a 2/SIGINT signal
- Signal handler for 2/SIGINT signal executes to completion
  - Default signal handler for 2/SIGINT signal exits process

## Process makes illegal memory reference

- Event gains attention of OS
- OS stops application process immediately, sending it a 11/SIGSEGV signal
- Signal handler for 11/SIGSEGV signal executes to completion
  - Default signal handler for 11/SIGSEGV signal prints “segmentation fault” and exits process





# Sending Signals via Keystrokes

Three signals can be sent from keyboard:

- Ctrl-c → 2/SIGINT signal
  - Default handler exits process
- Ctrl-z → 20/SIGTSTP signal
  - Default handler suspends process
- Ctrl-\ → 3/SIGQUIT signal
  - Default handler exits process



# Sending Signals via Commands

## kill Command

`kill -signal pid`

- Send a signal of type **signal** to the process with id **pid**
- Can specify either signal type name (-SIGINT) or number (-2)
- No signal type name or number specified => sends 15/SIGTERM signal
  - Default 15/SIGTERM handler exits process
- Editorial comment: Better command name would be **sendsig**

## Examples

`kill -2 1234`

`kill -SIGINT 1234`

- Same as pressing Ctrl-c if process 1234 is running in foreground



# Sending Signals via Function Call

## raise()

```
int raise(int iSig);
```

- Commands OS to send a signal of type `iSig` to current process
- Returns 0 to indicate success, non-0 to indicate failure

## Example

```
int ret = raise(SIGINT); /* Process commits suicide. */
assert(ret != 0); /* Shouldn't get here. */
```

Note: C90 function



# Sending Signals via Function Call

## kill()

```
int kill(pid_t iPid, int iSig);
```

- Sends a `iSig` signal to the process whose id is `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process
- Editorial comment: Better function name would be `sendsig()`

## Example

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT); /* Process sends itself a
 SIGINT signal (commits
 suicide?) */
```

Note: POSIX (not C90) function



# Signal Handling

Each signal type has a default handler

- Most default handlers exit the process

A program can install its own handler for signals of any type

Exceptions: A program *cannot* install its own handler for signals of type:

- 9/SIGKILL
  - Default handler exits the process
  - Catchable termination signal is 15/SIGTERM
- 19/SIGSTOP
  - Default handler suspends the process
  - Can resume the process with signal 18/SIGCONT
  - Catchable suspension signal is 20/SIGTSTP



# Outline

1. UNIX Process Control
2. Signals
3. **C90 Signal Handling**
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers



# Installing a Signal Handler

## `signal()`

```
sighandler_t signal(int iSig,
 sighandler_t pfHandler);
```

- Installs function **pfHandler** as the handler for signals of type **iSig**
- **pfHandler** is a function pointer:  
`typedef void (*sighandler_t) (int);`
- Returns the old handler on success, **SIG\_ERR** on error
- After call, **pfHandler** is invoked whenever process receives a signal of type **iSig**



# Installing a Handler Example 1

Program testsignal.c:

```
#define _GNU_SOURCE /* Use modern handling style */
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
 printf("In myHandler with argument %d\n", iSig);
}
...
```



# Installing a Handler Example 1 (cont.)

Program testsignal.c (cont.):

```
...
int main(void) {
 void (*pfRet)(int);
 pfRet = signal(SIGINT, myHandler);
 assert(pfRet != SIG_ERR);

 printf("Entering an infinite loop\n");
 for (;;)
 ;
 return 0;
}
```



# Installing a Handler Example 1 (cont.)

[Demo of testsignal.c]



# Installing a Handler Example 2

Program testsignalall.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
 printf("In myHandler with argument %d\n", iSig);
}
...
```



# Installing a Handler Example 2 (cont.)

## Program testsignalall.c (cont.):

```
...
int main(void) {
 void (*pfRet)(int);
 pfRet = signal(SIGHUP, myHandler); /* 1 */
 pfRet = signal(SIGINT, myHandler); /* 2 */
 pfRet = signal(SIGQUIT, myHandler); /* 3 */
 pfRet = signal(SIGILL, myHandler); /* 4 */
 pfRet = signal(SIGTRAP, myHandler); /* 5 */
 pfRet = signal(SIGABRT, myHandler); /* 6 */
 pfRet = signal(SIGBUS, myHandler); /* 7 */
 pfRet = signal(SIGFPE, myHandler); /* 8 */
 pfRet = signal(SIGKILL, myHandler); /* 9 */
 ...
}
```



# Installing a Handler Example 2 (cont.)

## Program testsignalall.c (cont.):

```
...
/* Etc., for every signal. */

printf("Entering an infinite loop\n");
for (;;)
 ;
return 0;
}
```



# Installing a Handler Example 2 (cont.)

[Demo of testsignalall.c]



# Installing a Handler Example 3

Program generates lots of temporary data

- Stores the data in a temporary file
- Must delete the file before exiting

```
...
int main(void) {
 FILE *psFile;
 psFile = fopen("temp.txt", "w");
 ...
 fclose(psFile);
 remove("temp.txt");
 return 0;
}
```



# Example 3 Problem

What if user types Ctrl-c?

- OS sends a 2/SIGINT signal to the process
- Default handler of 2/SIGINT exits the process

Problem: The temporary file is not deleted

- Process dies before `remove ("tmp.txt")` is executed

Challenge: Ctrl-c could happen at any time

- Which line of code will be interrupted???

Solution: Install a signal handler

- Define a “clean up” function to delete the file
- Install the function as a signal handler for 2/SIGINT



# Example 3 Solution

```
...
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig) {
 fclose(psFile);
 remove("tmp.txt");
 exit(EXIT_FAILURE);
}
int main(void) {
 void (*pfRet)(int);
 psFile = fopen("temp.txt", "w");
 pfRet = signal(SIGINT, cleanup);
 ...
 raise(SIGINT);
 return 0; /* Never get here. */
}
```



# Predefined Signal Handler: SIG\_IGN

Pre-defined signal handler: SIG\_IGN

Can install to ignore signals

```
int main(void) {
 void (*pfRet)(int);
 pfRet = signal(SIGINT, SIG_IGN);
 ...
}
```

Subsequently, process will ignore 2/SIGINT signals



# Predefined Signal Handler: SIG\_DFL

Pre-defined signal handler: SIG\_DFL

Can install to restore default signal handler

```
int main(void) {
 void (*pfRet)(int);
 pfRet = signal(SIGINT, somehandler);
 ...
 pfRet = signal(SIGINT, SIG_DFL);
 ...
}
```

Subsequently, process will handle 2/SIGINT signals using the default handler for 2/SIGINT signals



# Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. **C90 Signal Blocking**
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers



# Race Conditions in Signal Handlers

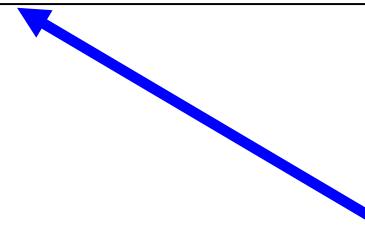
A **race condition** is a flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of other events.

Race conditions can occur in signal handlers...



# Race Condition Example

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance;
 iTemp += iMonthlySalary;
 iSavingsBalance = iTemp;
}
```



Handler for hypothetical  
“update monthly salary” signal



# Race Condition Example (cont.)

(1) Signal arrives; handler begins executing

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary;
 iSavingsBalance = iTemp;
}
```



# Race Condition Example (cont.)

(2) Another signal arrives; first instance of handler is interrupted; second instance of handler begins executing

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary;
 iSavingsBalance = iTemp;
}
```

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary;
 iSavingsBalance = iTemp;
}
```



# Race Condition Example (cont.)

(3) Second instance executes to completion

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary;
 iSavingsBalance = iTemp;
}
```

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary; 2050
 iSavingsBalance = iTemp; 2050
}
```



# Race Condition Example (cont.)

(4) Control returns to first instance, which executes to completion

```
void addSalaryToSavings(int iSig) {
 int iTemp;
 iTemp = iSavingsBalance; 2000
 iTemp += iMonthlySalary; 2050
 iSavingsBalance = iTemp; 2050
}
```

Lost 50 !!!



# Blocking Signals in Handlers

## Blocking signals

- To **block** a signal is to **queue** it for delivery at a later time

## Why block signals when handler is executing?

- Avoid race conditions when another signal of type x occurs while the handler for type x is executing

## How to block signals when handler is executing?

- **Automatic** during execution of signal handler!!!
- Previous sequence **cannot happen!!!**
- While executing a handler for a signal of type x, all signals of type x are blocked
- When/if signal handler returns, block is removed



# Race Conditions in General

Race conditions can occur elsewhere too

```
int iFlag = 0;

void myHandler(int iSig) {
 iFlag = 1;
}

int main(void) {
 if (iFlag == 0) {
 /* Do something */
 }
}
```

Problem: `myflag` might become 1 just after the comparison!

Must make sure that **critical sections** of code are not interrupted



# Blocking Signals in General

How to block signals in general?

- Not possible in C90
- Possible using POSIX functions...



# Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. **POSIX Signal Handling/Blocking**
6. Conclusion
7. (optional) Alarms and Interval Timers



# POSIX Signal Handling

## C90 standard

- Defines `signal()` and `raise()` functions
  - Work across all systems (UNIX, LINUX, Windows), but...
  - Work **differently** across some systems!!!
    - On some systems, signals are blocked during execution of handler for that type of signal -- but not so on other (older) systems
    - On some (older) systems, handler installation for signals of type x is cancelled after first signal of type x is received; must reinstall the handler -- but not so on other systems
- Does not provide mechanism to block signals in general



# POSIX Signal Handling

## POSIX standard

- Defines `kill()`, `sigprocmask()`, and `sigaction()` functions
  - Work the same across all POSIX-compliant UNIX systems (Linux, Solaris, etc.), but...
  - Do not work on non-UNIX systems (e.g. Windows)
- Provides mechanism to block signals in general



# Blocking Signals in General

Each process has a signal mask in the kernel

- OS uses the mask to decide which signals to deliver
- User program can modify mask with `sigprocmask()`

## `sigprocmask()`

```
int sigprocmask(int iHow,
 const sigset_t *psSet,
 sigset_t *psOldSet);
```

- `psSet`: Pointer to a signal set
- `psOldSet`: (Irrelevant for our purposes)
- `iHow`: How to modify the signal mask
  - `SIG_BLOCK`: Add `psSet` to the current mask
  - `SIG_UNBLOCK`: Remove `psSet` from the current mask
  - `SIG_SETMASK`: Install `psSet` as the signal mask
- Returns 0 iff successful

Functions for constructing signal sets

- `sigemptyset()`, `sigaddset()`, ...

Note: No parallel function in C90



# Blocking Signals Example

```
sigset_t sSet;

int main(void) {
 int iRet;
 sigemptyset(&sSet);
 sigaddset(&sSet, SIGINT);
 iRet = sigprocmask(SIG_BLOCK, &sSet, NULL);
 assert(iRet == 0);
 if (iFlag == 0) {
 /* Do something */
 }
 iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
 assert(iRet == 0);
 ...
}
```



# Blocking Signals in Handlers

Signals of type x automatically are blocked when executing handler for signals of type x

Additional signal types to be blocked can be defined at time of handler installation...



# Installing a Signal Handler

## `sigaction()`

```
int sigaction(int iSig,
 const struct sigaction *psAction,
 struct sigaction *psOldAction);
```

- **iSig**: The type of signal to be affected
- **psAction**: Pointer to a structure containing instructions on how to handle signals of type **iSig**, including signal handler name and which signal types should be blocked
- **psOldAction**: (Irrelevant for our purposes)
- Installs an appropriate handler
- Automatically blocks signals of type **iSig**
- Returns 0 iff successful

Note: More powerful than C90 `signal()`



# Installing a Handler Example

Program testsigaction.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void myHandler(int iSig) {
 printf("In myHandler with argument %d\n", iSig);
}
...
```



# Installing a Handler Example (cont.)

Program testsigaction.c (cont.):

```
...
int main(void) {
 int iRet;
 struct sigaction sAction;
 sAction.sa_flags = 0;
 sAction.sa_handler = myHandler;
 sigemptyset(&sAction.sa_mask);
 iRet = sigaction(SIGINT, &sAction, NULL);
 assert(iRet == 0);

 printf("Entering an infinite loop\n");
 for (;;)
 ;
 return 0;
}
```



# Installing a Handler Example (cont.)

[Demo of testsigaction.c]



# Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. **Conclusion**
7. (optional) Alarms and Interval Timers



# Predefined Signals

List of the predefined signals:

```
$ kill -1
 1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL
 5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE
 9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD
18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN
22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR 31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1
36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4 39) SIGRTMIN+5
40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8 43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6 59) SIGRTMAX-5
60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1
64) SIGRTMAX
```

Applications can define their own signals

- An application can define signals with unused values



# Summary

## Signals

- A **signal** is an asynchronous event mechanism
- C90 **raise()** or POSIX **kill()** sends a signal
- C90 **signal()** or POSIX **sigaction()** installs a signal handler
  - Most predefined signals are “catchable”
- Beware of race conditions
- Signals of type x automatically are blocked while handler for type x signals is running
- POSIX **sigprocmask()** blocks signals in any critical section of code



# Summary

Q: How does the OS communicate to application programs?

A: Signals

For more information:

Bryant & O'Hallaron, *Computer Systems: A Programmer's Perspective*, Chapter 8



# Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. **(optional) Alarms and Interval Timers**



# Alarms

## alarm()

```
unsigned int alarm(unsigned int uiSec);
```

- Sends 14/SIGALRM signal after **uiSec** seconds
- Cancels pending alarm if **uiSec** is 0
- Uses **real time**, alias **wall-clock time**
  - Time spent executing other processes counts
  - Time spent waiting for user input counts
- Return value is meaningless

Used to implement time-outs





# Alarm Example 1

Program testalarm.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig) {
 printf("In myHandler with argument %d\n", iSig);

 /* Set another alarm. */
 alarm(2);
}

...
```



# Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```
...
int main(void)
{
 void (*pfRet) (int);
 sigset_t sSet;
 int iRet;

 /* Make sure that SIGALRM is not blocked. */
 sigemptyset(&sSet);
 sigaddset(&sSet, SIGALRM);
 iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
 assert(iRet == 0);

 pfRet = signal(SIGALRM, myHandler);
 assert(pfRet != SIG_ERR);

 ...
}
```



# Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```
...
/* Set an alarm. */
alarm(2);

printf("Entering an infinite loop\n");
for (;;) {
 ;

 return 0;
}
```



# Alarm Example 1 (cont.)

[Demo of testalarm.c]



# Alarm Example 2

Program testalarmtimeout.c:

```
#define __GNUC__ SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
 printf("\nSorry. You took too long.\n");
 exit(EXIT_FAILURE);
}
```



# Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
int main(void) {
 int i;
 void (*pfRet)(int);
 sigset_t sSet;
 int iRet;

 /* Make sure that SIGALRM is not blocked. */
 sigemptyset(&sSet);
 sigaddset(&sSet, SIGALRM);
 iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
 assert(iRet == 0);

 ...
}
```



# Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
...
pfRet = signal(SIGALRM, myHandler);
assert(pfRet != SIG_ERR);

printf("Enter a number: ");
alarm(5);
scanf("%d", &i);
alarm(0);

printf("You entered the number %d.\n", i);
return 0;
}
```



# Alarm Example 2 (cont.)

[Demo of testalarmtimeout.c]



# Interval Timers

## `setitimer()`

```
int setitimer(int iWhich,
 const struct itimerval *psValue,
 struct itimerval *psOldValue);
```

- Sends 27/SIGPROF signal continually
- Timing is specified by **psValue**
- **psOldValue** is irrelevant for our purposes
- Uses **virtual time**, alias **CPU time**
  - Time spent executing other processes does not count
  - Time spent waiting for user input does not count
- Returns 0 iff successful

Used by execution profilers



# Interval Timer Example

Program testitimer.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <sys/time.h>

static void myHandler(int iSig) {
 printf("In myHandler with argument %d\n", iSig);
}

...
```



# Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
...
int main(void)
{
 int iRet;
 void (*pfRet)(int);
 struct itimerval sTimer;

 pfRet = signal(SIGPROF, myHandler);
 assert(pfRet != SIG_ERR);

 ...
}
```



# Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
...
/* Send first signal in 1 second, 0 microseconds. */
sTimer.it_value.tv_sec = 1;
sTimer.it_value.tv_usec = 0;

/* Send subsequent signals in 1 second,
 0 microseconds intervals. */
sTimer.it_interval.tv_sec = 1;
sTimer.it_interval.tv_usec = 0;

iRet = setitimer(ITIMER_PROF, &sTimer, NULL);
assert(iRet != -1);

printf("Entering an infinite loop\n");
for (;;)
 ;
return 0;
}
```



# Interval Timer Example (cont.)

[Demo of testitimer.c]



# Summary

## Alarms

- Call **alarm()** to deliver 14/SIGALRM signals in real/wall-clock time
- Alarms can be used to implement time-outs

## Interval Timers

- Call **setitimer()** to deliver 27/SIGPROF signals in virtual/CPU time
- Interval timers are used by execution profilers

# Socket Programming

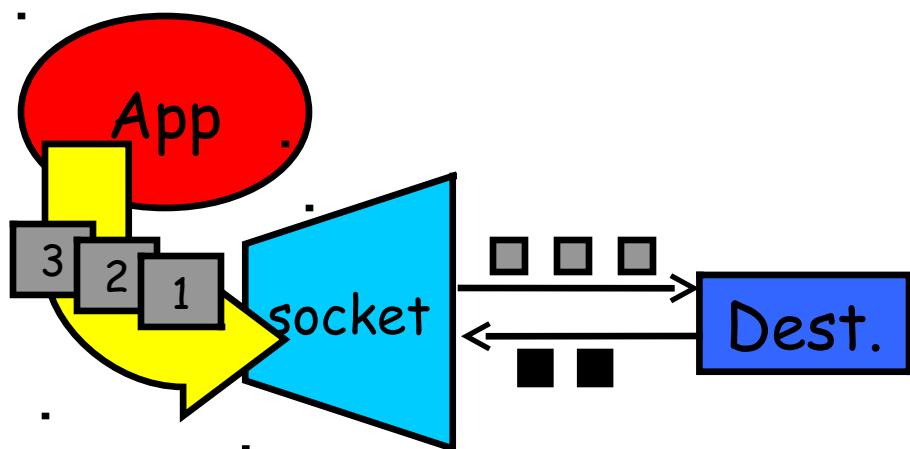
- What is a socket?
- Using sockets
  - Types (Protocols)
  - Associated functions
  - Styles
- We will look at using sockets in C
- For Java, see Chapter 2.6-2.8 (optional)
  - Note: Java sockets are conceptually quite similar

# What is a socket?

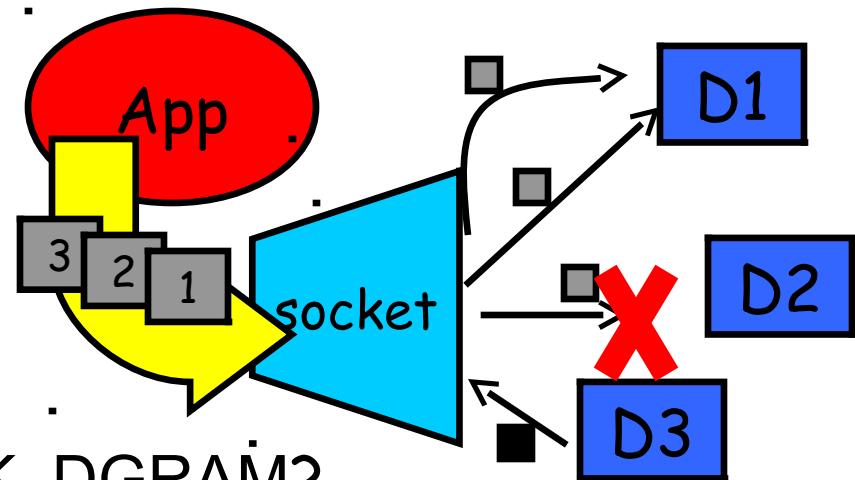
- An interface between application and network
  - The application creates a socket
  - The socket type dictates the style of communication
    - reliable vs. best effort
    - connection-oriented vs. connectionless
- Once configured the application can
  - pass data to the socket for network transmission
  - receive data from the socket (transmitted through the network by some other host)

# Two essential types of sockets

- SOCK\_STREAM
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional



- SOCK\_DGRAM
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" - app indicates dest. for each packet
  - can send or receive



Q: why have type SOCK\_DGRAM?

# Socket Creation in C: `socket`

- `int s = socket(domain, type, protocol);`
  - `s`: socket descriptor, an integer (like a file-handle)
  - `domain`: integer, communication domain
    - e.g., `PF_INET` (IPv4 protocol) - typically used
  - `type`: communication type
    - `SOCK_STREAM`: reliable, 2-way, connection-based service
    - `SOCK_DGRAM`: unreliable, connectionless,
    - other values: need root permission, rarely used, or obsolete
  - `protocol`: specifies protocol (see file `/etc/protocols` for a list of options) - usually set to 0
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to - it just creates the interface!

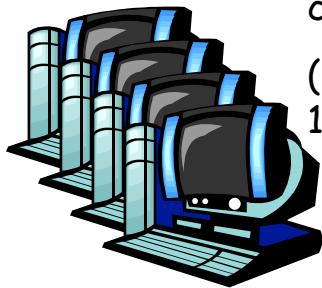
# A Socket-eye view of the Internet



medellin.cs.columbia.edu  
(128.59.21.14)



newworld.cs.umass.edu  
(128.119.245.93)

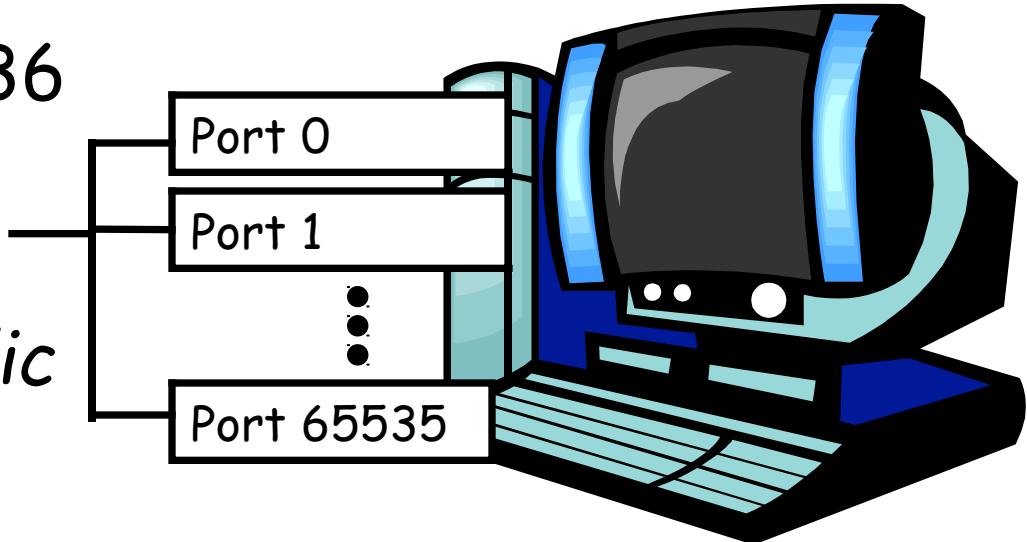


cluster.cs.columbia.edu  
(128.59.21.14, 128.59.16.7,  
128.59.16.5, 128.59.16.4)

- Each host machine has an IP address
- When a packet arrives at a host

# Ports

- Each host has 65,536 ports
- Some ports are reserved for specific apps
  - 20,21: FTP
  - 23: Telnet
  - 80: HTTP
  - see RFC 1700 (about 2000 ports are reserved)



- A socket provides an interface to send data to/from the network through a port

# Addresses, Ports and Sockets

- Like apartments and mailboxes
  - You are the application
  - Your apartment building address is the address
  - Your mailbox is the port
  - The post-office is the network
  - The socket is the key that gives you access to the right mailbox (one difference: assume outgoing mail is placed by you in your mailbox)
  
- Q: How do you choose which port a socket connects to?

# The bind function

- associates and (can exclusively) reserves a port for use by the socket
- `int status = bind(sockid, &addrport, size);`
  - `status`: error status, = -1 if bind failed
  - `sockid`: integer, socket descriptor
  - `addrport`: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR\_ANY - chooses a local address)
  - `size`: the size (in bytes) of the addrport structure
- bind can be skipped for both types of sockets.  
When and why?

# Skipping the bind

## ❑ SOCK\_DGRAM:

- if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
- if receiving, need to bind

## ❑ SOCK\_STREAM:

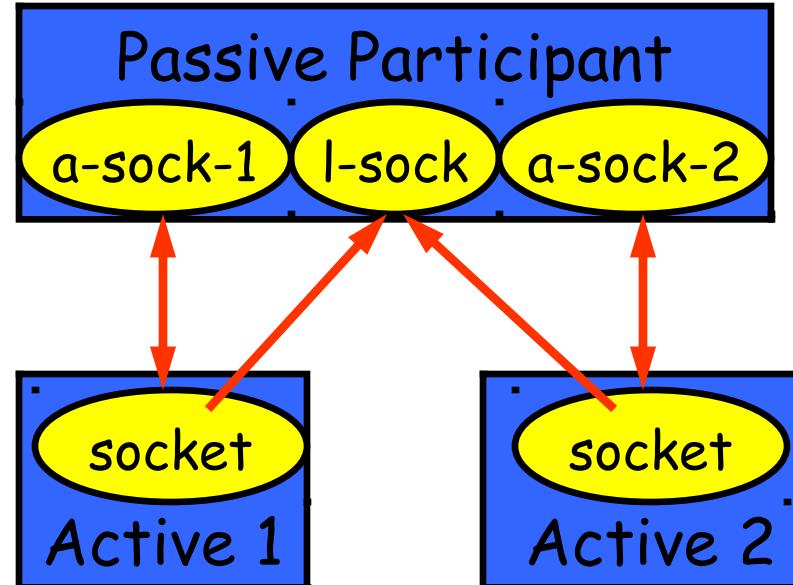
- destination determined during conn. setup
- don't need to know port sending from (during connection setup, receiving end is informed of port)

# Connection Setup (SOCK\_STREAM)

- Recall: no connection setup for SOCK\_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are "similar"
  - both can send & receive data
  - either can terminate the connection

# Connection setup cont'd

- Passive participant
  - step 1: **listen** (for incoming requests)
  - step 3: **accept** (a request)
  - step 4: data transfer
- Active participant
  - step 2: request & establish **connection**
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants
- Why?



# Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can "wait" for a connection
  - listen is non-blocking: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is blocking: waits for connection before returning

## connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)
- connect is blocking

# Sending / Receiving Data

- With a connection (SOCK\_STREAM):
  - int count = send(sock, &buf, len, flags);
    - count: # bytes transmitted (-1 if error)
    - buf: char[], buffer to be transmitted
    - len: integer, length of buffer (in bytes) to transmit
    - flags: integer, special options, usually just 0
  - int count = recv(sock, &buf, len, flags);
    - count: # bytes received (-1 if error)
    - buf: void[], stores received bytes
    - len: # bytes received
    - flags: integer, special options, usually just 0
  - Calls are blocking [returns only after data is sent (to socket buf) / received]

# Sending / Receiving Data (cont'd)

- Without a connection (SOCK\_DGRAM):
  - int count = `sendto(sock, &buf, len, flags, &addr, addrlen);`
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = `recvfrom(sock, &buf, len, flags, &addr, &addrlen);`
    - count, sock, buf, len, flags: same as recv
    - name: struct sockaddr, address of the source
    - namelen: sizeof(name): value/result parameter
- Calls are blocking [returns only after data is sent (to socket buf) / received]

# close

- When finished using a socket, the socket should be closed:
- `status = close(s);`
  - status: 0 if successful, -1 if error
  - s: the file descriptor (socket being closed)
- Closing a socket
  - closes a connection (for SOCK\_STREAM)
  - frees up the port used by the socket

# The struct sockaddr

- The generic:

```
struct sockaddr {
 u_short sa_family;
 char sa_data[14];
};
```

- sa\_family

- specifies which address family is being used
    - determines how the remaining 14 bytes are used

- The Internet-specific:

```
struct sockaddr_in {
 short sin_family;
 u_short sin_port;
 struct in_addr sin_addr;
 char sin_zero[8];
};
```

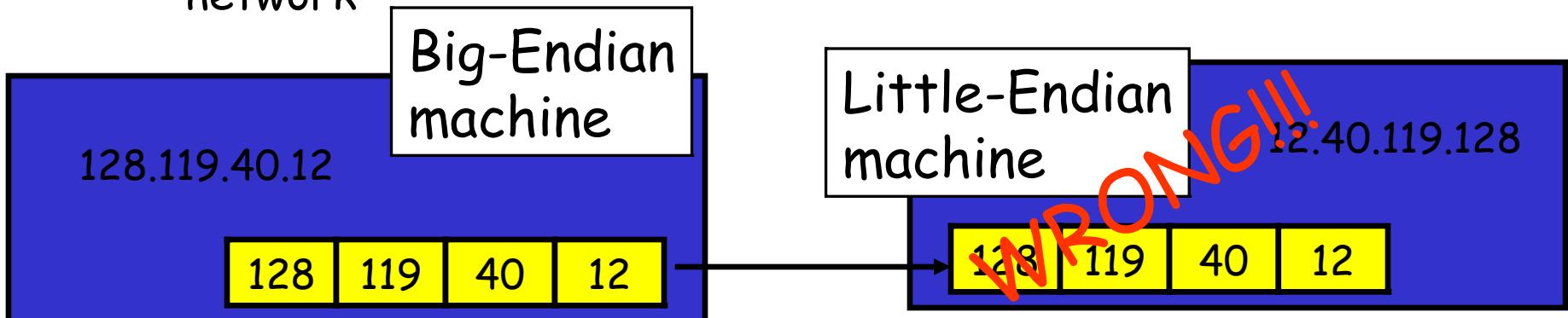
- sin\_family = AF\_INET
  - sin\_port: port # (0-65535)
  - sin\_addr: IP-address
  - sin\_zero: unused

# Address and port byte-ordering

- Address and port are stored as integers
  - `u_short sin_port; (16 bit)`
  - `in_addr sin_addr; (32 bit)`

```
struct in_addr {
 u_long s_addr;
};
```

- Problem:
  - different machines / OS's use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - these machines may communicate with one another over the network

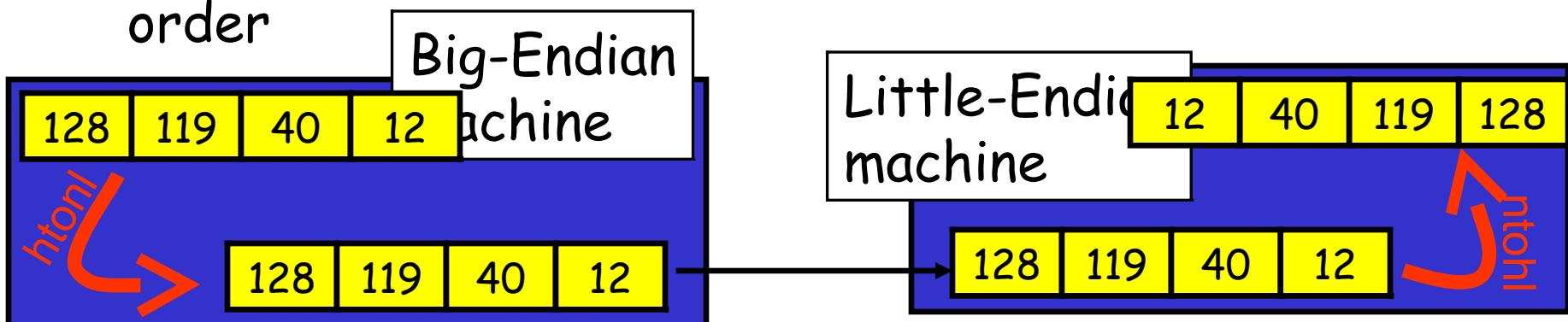


# Solution: Network Byte-Ordering

- Defs:
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network - always big-endian
- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)
- Q: should the socket perform the conversion automatically?
- Q: Given big-endian machines don't need conversion routines and little-endian machines do, how do we avoid writing two versions of code?

# UNIX's byte-ordering funcs

- `u_long htonl(u_long x);`
  - `u_short htons(u_short x);`
  - `u_long ntohl(u_long x);`
  - `u_short ntohs(u_short x);`
- 
- On big-endian machines, these routines do nothing
  - On little-endian machines, they reverse the byte order



- Same code would have worked regardless of endian-ness of the two machines

# Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket's buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing

# Dealing w/ blocking (cont'd)

- Options:
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the `fcntl` file-descriptor control function)
  - use the `select` function call.
- What does `select` do?
  - can be permanent blocking, time-limited blocking or non-blocking
  - input: a set of file-descriptors
  - output: info on the file-descriptors' status
  - i.e., can identify sockets that are "ready for use": calls involving that socket will return immediately

# select function call

- `int status = select(nfds, &readfds, &writefds, &exceptfds, &timeout);`
  - `status`: # of ready objects, -1 if error
  - `nfds`: 1 + largest file descriptor to check
  - `readfds`: list of descriptors to check if read-ready
  - `writefds`: list of descriptors to check if write-ready
  - `exceptfds`: list of descriptors to check if an exception is registered
  - `timeout`: time after which select returns, even if nothing ready - can be 0 or  $\infty$   
(point timeout parameter to NULL for  $\infty$ )

# To be used with select:

- Recall select uses a structure, `struct fd_set`
  - it is just a bit-vector
  - if bit  $i$  is set in [readfds, writefds, exceptfds], select will check if file descriptor (i.e. socket)  $i$  is ready for [reading, writing, exception]
- Before calling select:
  - `FD_ZERO(&fdvar)`: clears the structure
  - `FD_SET(i, &fdvar)`: to check file desc.  $i$
- After calling select:
  - `int FD_ISSET(i, &fdvar)`: boolean returns TRUE iff  $i$  is "ready"

# Other useful functions

- `bzero(char* c, int n)`: 0's n bytes starting at c
- `gethostname(char *name, int len)`: gets the name of the current host
- `gethostbyaddr(char *addr, int len, int type)`: converts IP hostname to structure containing long integer
- `inet_addr(const char *cp)`: converts dotted-decimal char-string to long integer
- `inet_ntoa(const struct in_addr in)`: converts long to dotted-decimal notation
  
- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order

# Release of ports

- Sometimes, a “rough” exit from a program (e.g., ctrl-c) does not properly free up a port
- Eventually (after a few minutes), the port will be freed
- To reduce the likelihood of this problem, include the following code:

```
#include <signal.h>
void cleanExit(){exit(0);}
```

- in socket code:

```
signal(SIGTERM, cleanExit);
signal(SIGINT, cleanExit);
```

# Final Thoughts

- Make sure to #include the header files that define used functions
- Check man-pages and course web-site for additional info

High

Performance

Distributed

System

## KUAS – High Performance Distributed System

### Linux Programming – Socket #1

Reporter: Po-Sen Wang

# Socket Function (1/7)

- `int socket(int domain, int type, int protocol)`  
**//Create a socket.**
- `int bind(int socket, const struct sockaddr *address, size_t address_len);`  
**//Name a socket.**
- `int listen(int socket, int backlog);`  
**//Create a queue of socket.**
- `int accept(int socket, struct sockaddr *address, size_t *address_len);`  
**//Accept connection.**
- `int connect(int socket, const struct sockaddr *address, size_t address_len);`  
**//Request connection.**

# Socket Function (2/7)

- `#include <sys/types.h>`
- `#include <sys/socket.h>`
- `int socket(int domain, int type, int protocol)`
  - *domain:*

|              |                                                |
|--------------|------------------------------------------------|
| AF_UNIX      | UNIX internal (file system sockets)            |
| AF_INET      | ARPA internet protocols (UNIX network sockets) |
| AF_ISO       | ISO standard protocols                         |
| AF_NS        | Xerox network systems protocols                |
| AF_IPX       | Novell IPX protocol                            |
| AF_APPLETALK | Apple talk DDS                                 |

- *type:* SOCK\_STREAM (TCP) 、 SOCK\_DGRAM (UDP)
- *protocol:* 0 代表使用預設的協定。

# Socket Function (3/7)

- `#include <sys/socket.h>`
- `int bind(int socket, const struct sockaddr *address, size_t address_len);`
  - *socket*: File descriptor.
  - *address*: Socket address.
  - *address\_len*: Address length.

# Socket Function (4/7)

- AF\_UNIX:

```
struct sockaddr_un {
 sa_family_t sun_family;
 char sun_path[];
};
```

- AF\_INET:

```
struct sockaddr_in {
 short int sin_family;
 unsigned short int sin_port;
 struct in_addr sin_addr;
};
```

```
struct in_addr {
 unsigned long int s_addr;
};
```

# Socket Function (5/7)

- `#include <sys/socket.h>`
- `int listen(int socket, int backlog);`
  - *socket*: File descriptor.
  - *backlog*: Maximum of queue for listen.

# Socket Function (6/7)

- `#include <sys/socket.h>`
- `int accept(int socket, struct sockaddr *address, size_t *address_len);`
  - *socket*: File descriptor.
  - *address*: Socket address.
  - *address\_len*: Address length.

# Socket Function (7/7)

- `#include <sys/socket.h>`
- `int connect(int socket, const struct sockaddr *address, size_t address_len);`
  - *socket*: File descriptor.
  - *address*: Socket address.
  - *address\_len*: Address length.

## Socket Example 1 – Use AF\_UNIX (1/6)

A screenshot of a terminal window titled "kevin@localhost:~/linux\_program/ch14 [80x24]". The window contains the following text:

```
[kevin@localhost ch14]$./server1 &
[4] 8921
[kevin@localhost ch14]$ server waiting
./client1
server waiting
char from server = B
[kevin@localhost ch14]$
```

■ client1.c

```
/* Make the necessary includes and set up the variables. */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
 int sockfd;
 int len;
 struct sockaddr_un address;
 int result;
 char ch = 'A';

 /* Create a socket for the client. */
 sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

# Socket Example 1 – Use AF\_UNIX (3/6)

## client1.c

```
/* Name the socket, as agreed with the server. */

address.sun_family = AF_UNIX;
strcpy(address.sun_path, "server_socket");
len = sizeof(address);

/* Now connect our socket to the server's socket. */

result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
 perror("oops: client1");
 exit(1);
}

/* We can now read/write via sockfd. */

write(sockfd, &ch, 1);
read(sockfd, &ch, 1);
printf("char from server = %c\n", ch);
close(sockfd);
exit(0);
}
```

# Socket Example 1 – Use AF\_UNIX (4/6)

## ■ server1.c

```
/* Make the necessary includes and set up the variables. */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

int main()
{
 int server_sockfd, client_sockfd;
 int server_len, client_len;
 struct sockaddr_un server_address;
 struct sockaddr_un client_address;

/* Remove any old socket and create an unnamed socket for the server. */

 unlink("server_socket");
 server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

- server1.c

```
/* Name the socket. */

server_address.sun_family = AF_UNIX;
strcpy(server_address.sun_path, "server_socket");
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/* Create a connection queue and wait for clients. */

listen(server_sockfd, 5);
while(1) {
 char ch;

 printf("server waiting\n");
```

## ■ server1.c

```
/* Accept a connection. */

client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
 (struct sockaddr *) &client_address, &client_len);

/* We can now read/write to client on client_sockfd. */

read(client_sockfd, &ch, 1);
ch++;
write(client_sockfd, &ch, 1);
close(client_sockfd);

}
```

## ■ client2.c

```
/* Make the necessary includes and set up the variables. */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
 int sockfd;
 int len;
 struct sockaddr_in address;
 int result;
 char ch = 'A';

 /* Create a socket for the client. */
 sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

# Socket Example 2 – Use AF\_INET (2/5)

## ■ client2.c

```
/* Name the socket, as agreed with the server. */

address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");
address.sin_port = 9734;
len = sizeof(address);

/* Now connect our socket to the server's socket. */

result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
 perror("oops: client2");
 exit(1);
}

/* We can now read/write via sockfd. */

write(sockfd, &ch, 1);
read(sockfd, &ch, 1);
printf("char from server = %c\n", ch);
close(sockfd);
exit(0);
}
```

# Socket Example 2 – Use AF\_INET (3/5)

## server2.c

```
/* Make the necessary includes and set up the variables. */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

int main()
{
 int server_sockfd, client_sockfd;
 int server_len, client_len;
 struct sockaddr_in server_address;
 struct sockaddr_in client_address;

 /* Create an unnamed socket for the server. */
 server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

## ■ server2.c

```
/* Name the socket. */

server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);

/* Create a connection queue and wait for clients. */

listen(server_sockfd, 5);
while(1) {
 char ch;

 printf("server waiting\n");
}
```

## ■ server2.c

```
/* Accept a connection. */

client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
 (struct sockaddr *) &client_address, &client_len);

/* We can now read/write to client on client_sockfd. */

read(client_sockfd, &ch, 1);
ch++;
write(client_sockfd, &ch, 1);
close(client_sockfd);
}

}
```

## 網路位元組排序 (1/4)

kevin@localhost:~/linux\_program/ch14 [80x24]



連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)

```
[kevin@localhost ch14]$./server2 &
[1] 8958
[kevin@localhost ch14]$ server waiting
./client2
server waiting
char from server = B
```

```
[kevin@localhost ch14]$ netstat
Active Internet connections (w/o servers)
```

| Proto | Recv-Q | Send-Q | Local Address           | Foreign Address         | State       |
|-------|--------|--------|-------------------------|-------------------------|-------------|
| tcp   | 0      | 0      | localhost.localdom:9010 | localhost.localdo:32803 | ESTABLISHED |
| tcp   | 0      | 0      | localhost.localdom:1574 | localhost.localdo:40214 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdom:8649 | localhost.localdo:40211 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdom:8649 | localhost.localdo:40210 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdom:8649 | localhost.localdo:40213 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdom:8649 | localhost.localdo:40212 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdom:8649 | localhost.localdo:40215 | TIME_WAIT   |
| tcp   | 0      | 0      | localhost.localdo:32803 | localhost.localdom:9010 | ESTABLISHED |
| tcp   | 1      | 0      | localhost.localdo:32805 | localhost.localdoma:ipp | CLOSE_WAIT  |
| tcp   | 0      | 0      | 203.64.102:microsoft-ds | 140.127.114.41:2909     | ESTABLISHED |
| tcp   | 0      | 0      | ::ffff:203.64.102.1:ssh | 218-164-105-195.d:65243 | ESTABLISHED |
| tcp   | 0      | 1380   | ::ffff:203.64.102.1:ssh | ::ffff:140.127.114:1094 | ESTABLISHED |
| udp   | 0      | 0      | 203.64.102.188:32769    | 239.2.11.71:8649        | ESTABLISHED |

```
Active UNIX domain sockets (w/o servers)
```

| Proto | RefCount | Flags | Type | State | I-Node Path |
|-------|----------|-------|------|-------|-------------|
|-------|----------|-------|------|-------|-------------|

## 網路位元組排序 (2/4)

- `#include <netinet/in.h>`
- `unsigned long int htonl(unsigned long int hostlong);`
- `unsigned short int htons(unsigned short int hostshort);`
- `unsigned long int ntohl(unsigned long int hostlong);`
- `unsigned short int ntohs(unsigned short int hostshort);`

- server3.c:

```
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
```

- client3.c:

```
address.sin_port = htons(9734);
```

## 網路位元組排序 (4/4)

kevin@localhost:~/linux\_program/ch14 [80x24]

連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)

```
[kevin@localhost ch14]$./server3 &
[1] 8999
[kevin@localhost ch14]$ server waiting
./client3
server waiting
char from server = B
[kevin@localhost ch14]$ netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address Foreign Address State
tcp 0 0 localhost.localdom:9010 localhost.localdo:32803 ESTABLISHED
tcp 0 0 203.64.102.188:ftp 218-164-105-195.d:64956 ESTABLISHED
tcp 0 0 203.64.102.188:9574 218-164-105-195.d:65010 TIME_WAIT
tcp 0 0 localhost.localdom:8649 localhost.localdo:40263 TIME_WAIT
tcp 0 0 localhost.localdom:8649 localhost.localdo:40262 TIME_WAIT
tcp 0 0 localhost.localdom:8649 localhost.localdo:40264 TIME_WAIT
tcp 0 0 localhost.localdom:8649 localhost.localdo:40266 TIME_WAIT
tcp 0 0 localhost.localdom:9734 localhost.localdo:40265 TIME_WAIT
tcp 0 0 203.64.102.188:48103 218-164-105-195.d:64948 TIME_WAIT
tcp 0 0 203.64.102.188:ftp 218-164-105-195.d:64575 ESTABLISHED
tcp 0 0 localhost.localdo:32803 localhost.localdom:9010 ESTABLISHED
tcp 1 0 localhost.localdo:32805 localhost.localdom:a:ipp CLOSE_WAIT
tcp 0 0 203.64.102:microsoft-ds 140.127.114.41:2909 ESTABLISHED
tcp 0 0 ::ffff:203.64.102.1:ssh 218-164-105-195.d:65243 ESTABLISHED
tcp 0 264 ::ffff:203.64.102.1:ssh ::ffff:140.127.114:1094 ESTABLISHED
```



High Performance Distributed Systems Lab

# Linux Programming socket

報告者：彭皓廷

- ◆ 通訊端是通訊機制，它讓 client/server 系統可以在單一機台上或是在網路上進行開發。
- ◆ 首先。server 應用程式建立一通訊端，它是被指定到 server 程序的作業系統資源
- ◆ 接下來，server 程序給通訊端名稱，對於網路 socket 檔案名稱就是服務的辨識子，客戶端可以藉由它連結到伺服器。

◆ socket 的命名必須透過 bind 系統呼叫，伺服器就在這個具名的 socket 上等待客戶端的連結請求，listen 是用來產生一個佇列接應連結請求，而 accept 系統呼叫則讓伺服器接受連結。

# 產生 socket

- ◆ `int socket(int domain , int type , int protocol );`
- ◆ AF\_UNIX
- ◆ AF\_INET
- ◆ AF\_ISO
- ◆ type: 指定 socket 類型，可用值為：
  - ◆ SOCK\_STREAM: 連續性可靠性連結性的雙向位元組串流。
  - ◆ SOCK\_DGRAM: 傳送固定大小的訊息，可靠度不佳。
    -

# Socket 位址格式 AF\_UNIX,AF\_INET

◆ struct sockaddr\_un {  
    sa\_family\_t sun\_family; //AF\_UNIX  
    char          sun\_path[ ]; //path name  
};

struct sockaddr\_in {  
    short int                  sin\_family; //AF\_INET  
    unsigned short int       sin\_port; //port num  
    struct in\_addr          sin\_addr; //internet  
    address                  };

# 通訊端命名

- ◆ `#include<sys/socket.h>`
- ◆ `int bind(int socket,const struct sockaddr *address,size_t address_len);`
- ◆ `bind` 呼叫參數指定的位址 `address` 到一未命名的通訊端，此通訊端關連於 `socket` 檔案 `descriptor`，位址結構的長度以 `address_len` 傳遞。

# 建立通訊端佇列

- ◆ `#include<sys/socket.h>`
- ◆ `int listen(int socket,int backlog);`
- ◆ 根據上限，限制佇列中延後的連結數目，`listen` 將佇列長度設定為 `backlog`，根據佇列長度限制安排通訊端上被暫緩的連結。

# 接受連結

- ◆ `int accept( int socket, struct sockaddr *addr , size_t socklen_t* len );`
- ◆ 當一個客戶端試圖連結 socket ， accept 系統呼叫會回覆，而此客戶端是在佇列中第一個未解決的連結請求， accept 函數建立新的通訊端以便與 client 連接傳回 descriptor 。

# 連結請求

- ◆ `#include<sys/socket.h>`
- ◆ `int connect(int socket,const struct sockaddr *address,size_t address_len);`
- ◆ 參數 socket 指定的通訊端連接到參數 address 指定的 serve 通訊端， address 的長度為 address\_len
- ◆ EBADF 一無效的檔案描述被傳給 socket
- ◆ EALREADY socket 中已有一連結執行中
- ◆ ETIMEDOUT 連結發生
- ◆ ECONNREFUSED 請求之連結被 SERVER 拒絕

# client

- ◆ 建立 client 的通訊端

```
sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

- ◆ 命名 server 也同意的通訊端

```
address.sun_family = AF_UNIX;
strcpy(address.sun_path, "server_socket");
len = sizeof(address);
```

- ◆ 將我們的通訊端連上 server 通訊端

```
result = connect(sockfd, (struct sockaddr *)&address, len);

if(result == -1) {
 perror("oops: client1");
 exit(1);
```

## ◆ 現在透過 sockfd 來讀寫

```
write(sockfd, &ch, 1);
read(sockfd, &ch, 1);
printf("char from server = %c\n", ch);
close(sockfd);
exit(0);
```

# server

- ◆ 移除舊的通訊端與為 server 建立一未具名的通訊端

```
unlink("server_socket");
server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

- ◆ 為通訊端命名

```
server_address.sun_family = AF_UNIX;
strcpy(server_address.sun_path, "server_socket");
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address,
 server_len);
```

## ◆ 建立一連結佇列與等待 client

```
listen(server_sockfd, 5);
while(1) {
 char ch;

 printf("server waiting\n");
```

## ◆ 接受連結

```
client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
 (struct sockaddr *)&client_address, &client_len);
```

## ◆ 讀取與寫入位在 client\_sockfd 上的 client

```
read(client_sockfd, &ch, 1);
ch++;
write(client_sockfd, &ch, 1);
close(client_sockfd);
```

```
root@cuda04:~/Frank/ex/socket# ./server1 &
[8] 8246
root@cuda04:~/Frank/ex/socket# server waiting
./client1
server waiting
char from server = B
```



# 網路 client

## ◆ 建立 client 的通訊端

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

## ◆ 命名通訊端並且與 server 相符

```
address.sin_family = AF_INET;
address.sin_addr.s_addr = inet_addr("127.0.0.1");
address.sin_port = 9734;
len = sizeof(address);
```



# 網路 server

- ◆ 為 server 建立一未命名的通訊端

```
server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

- ◆ 命名通訊端

```
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
server_address.sin_port = 9734;
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

```
root@cuda04:~/Frank/ex/socket# ./server2 &
[10] 8309
root@cuda04:~/Frank/ex/socket# server waiting
./client2
server waiting
char from server = B
```

# 網路資訊

- ◆ #include<netdb.h>
- ◆ struct hostent \*gethostbyaddr(const void \*addr, size\_t len,int type);
- ◆ struct hostent \*gethostbyname(const char \*name);
- ◆ struct hostnet{
  - ◆ char \*h\_name; //name of the host
  - ◆ char \*\*h\_aliases; //list of aliases
  - ◆ int h\_addrtype; //address type
  - ◆ int h\_length; // length in bytes of the address
  - ◆ char \*\*h\_addr\_list //list of address
- ◆ };

- ◆ `struct servent *getservbyname(const char *name,const char *proto);`
- ◆ `struct servent *getservbyport(int port,const char *proto);`
- ◆ `struct servent{`
- ◆ `char *s_name; //name of the service`
- ◆ `char **s_aliases; //list of aliases`
- ◆ `int s_port; // the ip port number`
- ◆ `char *s_proto //the service type · tcp or udp`
- ◆ `};`

# 取得主機的電腦資訊

```
int main(int argc, char *argv[])
{
 char *host, **names, **addrs;
 struct hostent *hostinfo;
```

- ◆ 利用 gethostname 取得 host 名稱，或直接根據使用者傳入的參數

```
if(argc == 1) {
 char myname[256];
 gethostname(myname, 255);
 host = myname;
}
else
 host = argv[1];
```

## ◆ 呼叫 gethostname ，若沒有發現資訊回傳錯誤

```
hostinfo = gethostbyname(host);
if(!hostinfo) {
 fprintf(stderr, "cannot get info for host: %s\n", host);
 exit(1);
}
```

## ◆ 顯示主機名與其他別名 (alias)

```
printf("results for host %s:\n", host);
printf("Name: %s\n", hostinfo -> h_name);
printf("Aliases:");
names = hostinfo -> h_aliases;
while(*names) {
 printf(" %s", *names);
 names++;
}
printf("\n");
```

## ◆ 若有問題的主機非一 IP 主機則離開

```
if(hostinfo -> h_addrtype != AF_INET) {
 fprintf(stderr, "not an IP host!\n");
 exit(1);
```

## ◆ 否則顯示 IP 位址

```
addrs = hostinfo -> h_addr_list;
while(*addrs) {
 printf("%s", inet_ntoa(*(struct in_addr *)*addrs));
 addrs++;
}
printf("\n");
exit(0);
```

```
root@cuda04:~/Frank/ex/socket# ./getname cuda04
results for host cuda04:
Name: cuda04
Aliases:
 192.168.1.204
```



# 多重客戶端

- ◆ 呼叫 fork 產生第二個處理程序，socket 被子處理程序繼承，主伺服器繼續接聽其他客戶端的連結請求，子處理程序負責與客戶通訊

```
int server_sockfd, client_sockfd;
int server_len, client_len;
struct sockaddr_in server_address;
struct sockaddr_in client_address;

server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = htonl(INADDR_ANY);
server_address.sin_port = htons(9734);
server_len = sizeof(server_address);
bind(server_sockfd, (struct sockaddr *)&server_address, server_len);
```

- ◆ 產生一個連結佇列，忽略子處理程序的離開信號，等待客戶端

```
listen(server_sockfd, 5);

signal(SIGCHLD, SIG_IGN);

while(1) {
 char ch;
 printf("server waiting\n");
}
```

## ◆ 接受連結

```
client_len = sizeof(client_address);
client_sockfd = accept(server_sockfd,
 (struct sockaddr *)&client_address, &client_len);
```

## ◆ 利用 fork 產生一個處理程序，並檢查目前的是父處理程序還是子處理程序

```
if(fork() == 0) {
```

- ◆ 如果是子處理程序，利用 client\_sockfd 讀寫客戶端。途中五秒的延遲只是為了展示目的

```
read(client_sockfd, &ch, 1);
sleep(5);
ch++;
write(client_sockfd, &ch, 1);
close(client_sockfd);
exit(0);
```

- ◆ 如果父處理程序，對客戶端工作已經完成

```
else {
 close(client_sockfd);
}
```

```
root@cuda04:~/Frank/ex/socket# ./server4 &
[17] 9855
root@cuda04:~/Frank/ex/socket# server waiting
./client3
server waiting
char from server = B
root@cuda04:~/Frank/ex/socket# vim server4.c
root@cuda04:~/Frank/ex/socket# ./server4 &
[18] 9930
root@cuda04:~/Frank/ex/socket# server waiting
./client3 & ./client3 & ./client3 &
[19] 9931
[20] 9932
[21] 9933
root@cuda04:~/Frank/ex/socket# server waiting
server waiting
server waiting
char from server = B
char from server = B
char from server = B
```



# select

- ◆ 讓一個程式可以同時針對很多低階的檔案描述子，等待輸入到達（或輸出完成）這代表終端模擬程式可以被擱置，直到有事情發生。
- ◆ select 函數在 fd\_set 資料結構上運作
- ◆ #include <sys/types.h>
- ◆ #include <sys/time.h>
- ◆ void FD\_ZERO(fd\_set \*fdset);
- ◆ void FD\_CLR(int fd,fd\_set \*fdset);
- ◆ void FD\_SET(int fd,fd\_set \*fdset);
- ◆ int FD\_ISSET(int fd,fd\_set \*fdset);

- ◆ FD\_ZERO 初始 fd\_set ，將它設為空集合。
- ◆ FD\_CLR ， FD\_SET 設定和清除集合中的 fd 檔案描述子。
- ◆ 如果 fd 是 fdset 中的一個項目 FD\_ISSET 就會回傳一個非零值。
- ◆ FD\_SETSIZE 定義 fd\_set 結構中，可以容納的檔案描述子數量。

- ◆ select 函數也可以定義一個逾時時間值，來防止無限制地被擋置。
- ◆ struct timeval{
- ◆     Time\_t tv\_sec; // seconds
- ◆     long tv\_usec; // microseconds
- ◆ }

- ◆ `int select (int nfds ,fd_set *readfds,fd_set *writefds,fd_set *errorfds, struct timeval * timeout);`
- ◆ nfds 代表要檢驗的檔案描述子數量， readfds, writefds, errorfds ，可以為空指標
- ◆ readfds : 用於檢查可讀性
- ◆ writefds : 用於檢查可寫性
- ◆ errorfds : 檢查是否有錯誤
- ◆ timeout 時間過後沒有情況發生 select 會回覆
- ◆ 如果 timeout 參數是空指標也沒情況發生，函數將被擋置

# 實作 select，讀取鍵盤，逾時 2.5 秒

## ◆ 宣告，初始 input 管理鍵盤輸入

```
int main()
{
 char buffer[128];
 int result, nread;

 fd_set inputs, testfds;
 struct timeval timeout;

 FD_ZERO(&inputs);
 FD_SET(0, &inputs);
```

## ◆ 在 stdin 上等待輸入

```
while(1) {
 testfds = inputs;
 timeout.tv_sec = 2;
 timeout.tv_usec = 500000;

 result = select(FD_SETSIZE, &testfds, (fd_set *)0, (fd_set *)0, &timeout);
```

- ◆ 檢驗 result ，如果沒有輸入，程式會繼續迴圈，如果有錯誤程式會離開

```
while(1) {
 testfds = inputs;
 timeout.tv_sec = 2;
 timeout.tv_usec = 500000;

 result = select(FD_SETSIZE, &testfds, (fd_set *)0, (fd_set *)0, &timeout);
```

- ◆ 程式等待的過程中，如果有動作，就會從 stdin 讀取輸入，並印出，直到輸入，`ctrl+d`

```
default:
 if(FD_ISSET(0, &testfds)) {
 ioctl(0, FIONREAD, &nread);
 if(nread == 0) {
 printf("keyboard done\n");
 exit(0);
 }
 nread = read(0, buffer, nread);
 buffer[nread] = 0;
 printf("read %d from keyboard: %s", nread, buffer);
 }
 break;
}
```

```
root@cuda04:~/Frank/ex/socket# ./select
timeout
qwe timeout

read 4 from keyboard: qwe
timeout
qw
read 3 from keyboard: qw
timeout
qwe
read 4 from keyboard: qwe
```





# Thanks for your listening !!



High Performance  
Distributed Systems Lab

High

Performance

Distributed

System

## KUAS – High Performance Distributed System

Linux Programming – IPC (Interprocess Communication)

Reporter: Po-Sen Wang

# Semaphore Concept (1/3)

- 一筆資料如果同時被兩個程式寫入，會發生什麼事呢？也許是正確的結果，但更有可能是錯誤的結果。
- 一變數  $A = 10$ ，process1 要對變數 A 作遞增的動作，process2 要對變數 A 作遞減的動作。得到的結果，可能是 9、10、11 其中之一，然而正確的答案是 10。
- 所以，要避免一筆資料同時被寫入，我們需要 semaphore 來限制能進行寫入的程式。

## Semaphore Concept (2/3)

- 最簡單的 semaphore 是一個只能有 0 或 1 的變數，它利用兩個操作 P 和 V 來控制。P 與 V 的定義如下：
  - P : 若 semaphore 大於 0 則 semaphore 減少 1  
                  ，若 semaphore 為 0 則暫停程序。
  - V : 將 semaphore 增加 1，並恢復其他被暫停的程序。

# Semaphore Concept (3/3)

## ■ Semaphore 的虛擬碼：

```
semaphore sv = 1;

loop forever {
 P(sv);
 critical code section;
 V(sv);
 non-critical code section;
}
```

# Semaphore Function (1/4)

- #include <sys/sem.h>
- #include <sys/type.h>
- #include <sys/ipc.h>
  
- int semget(key\_t *key*, int *num\_sems*, int *sem\_flags*);  
    建立新的 **semaphore** 或取得現有的 **semaphore id** 。
- int semop(int *sem\_id*, struct sembuf \**sem\_ops*, size\_t *num\_sem\_ops*);  
    允許對 **semaphore** 資訊的直接存取。
- int semctl(int *sem\_id*, int *sem\_num*, int *command*, union semun *sem\_union*);  
    用來改變 **semaphore** 值。

# Semaphore Function (2/4)

- `int semget(key_t key, int num_sems, int sem_flags);`
  - *key*: 用來允許不相關的程序存取相同的 semaphore。
  - *num\_sems*: 需要的 semaphore 數，通常為 1。
  - *sem\_flags*: Semaphore 的權限。在建立新的 semaphore 時要加上 `IPC_CREATE`。

# Semaphore Function (3/4)

- `int semop(int sem_id, struct sembuf *sem_ops, size_t num_sem_ops);`
  - *sem\_id*: Semaphore id.
  - *sem\_ops*:  
`struct sembuf {`
    - `short sem_num; // 通常設為 0，除非是使用 semaphore 陣列。`
    - `short sem_op; // 設為 -1 代表 semaphore 的 P 動作；設為 +1 代表 semaphore 的 V 動作。`
    - `short sem_flg; // 一般設定為 SEM_UNDO 。`
  - *num\_sem\_ops*: *sem\_ops* 的數量。

# Semaphore Function (4/4)

- `int semctl(int sem_id, int sem_num, int command, union semun sem_union);`
  - *sem\_id*: Semaphore id.
  - *sem\_num*: 通常為 0。
  - *command*: SETVAL : 用來將 semaphore 起始化。  
IPC\_RMID : 用來刪除 semaphore。
  - union semun {  
  
    int val; /\* Value for SETVAL \*/  
  
    struct semid\_ds \*buf; /\* Buffer for IPC\_STAT, IPC\_SET \*/  
  
    unsigned short \*array; /\* Array for GETALL, SETALL \*/  
  
    struct seminfo \*\_\_buf; /\* Buffer for IPC\_INFO \*/  
  
};

# Semaphore Example (1/6)

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

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "semun.h"

static int set_semvalue(void);
static void del_semvalue(void);
static int semaphore_p(void);
static int semaphore_v(void);

static int sem_id;
```

# Semaphore Example (2/6)

```
int main(int argc, char *argv[])
{
 int i;
 int pause_time;
 char *op_char_in = "process2 in";
 char *op_char_out = "process2 out";

 srand((unsigned int) getpid());

 sem_id = semget((key_t)888, 1, 0666 | IPC_CREAT);

 if (argc > 1) {
 if (!set_semvalue()) {
 fprintf(stderr, "Failed to initialize semaphore\n");
 exit(EXIT_FAILURE);
 }
 op_char_in = "process1 in";
 op_char_out = "process1 out";
 sleep(2);
 }
}
```

# Semaphore Example (3/6)

```
for(i = 0; i < 5; i++) {

 if (!semaphore_p()) exit(EXIT_FAILURE);
 printf("%s\t", op_char_in);fflush(stdout);
 pause_time = rand() % 3;
 sleep(pause_time);
 printf("%s\n", op_char_out);fflush(stdout);

 if (!semaphore_v()) exit(EXIT_FAILURE);

 pause_time = rand() % 2;
 sleep(pause_time);
}

printf("\n%d - finished\n", getpid());

if (argc > 1) {
 sleep(10);
 del_semvalue();
}

exit(EXIT_SUCCESS);
}
```

# Semaphore Example (4/6)

```
static int set_semvalue(void)
{
 union semun sem_union;

 sem_union.val = 1;
 if (semctl(sem_id, 0, SETVAL, sem_union) == -1) return(0);
 return(1);
}

static void del_semvalue(void)
{
 union semun sem_union;

 if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1)
 fprintf(stderr, "Failed to delete semaphore\n");
}
```

# Semaphore Example (5/6)

```
static int semaphore_p(void)
{
 struct sembuf sem_b;

 sem_b.sem_num = 0;
 sem_b.sem_op = -1; /* P() */
 sem_b.sem_flg = SEM_UNDO;
 if (semop(sem_id, &sem_b, 1) == -1) {
 fprintf(stderr, "semaphore_p failed\n");
 return(0);
 }
 return(1);
}
```

```
static int semaphore_v(void)
{
 struct sembuf sem_b;

 sem_b.sem_num = 0;
 sem_b.sem_op = 1; /* V() */
 sem_b.sem_flg = SEM_UNDO;
 if (semop(sem_id, &sem_b, 1) == -1) {
 fprintf(stderr, "semaphore_v failed\n");
 return(0);
 }
 return(1);
}
```

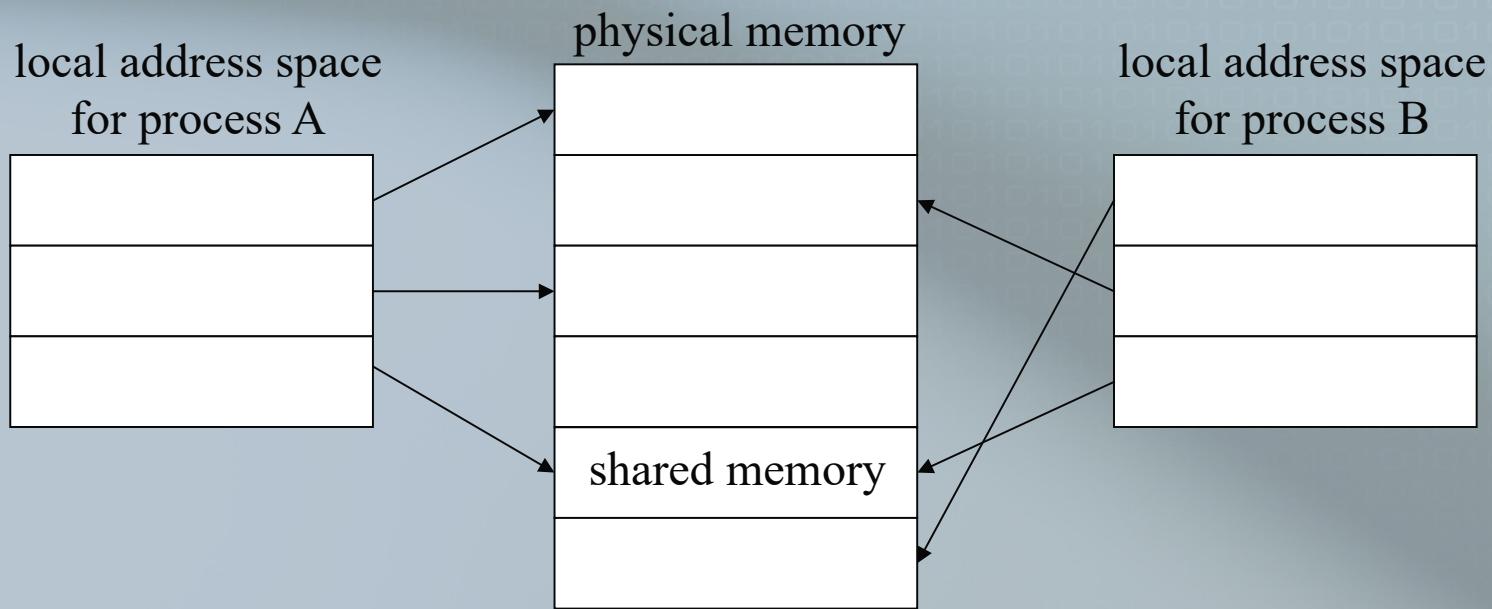
## Semaphore Example (6/6)

The screenshot shows a terminal window titled "kevin@localhost:~/linux\_program/ch13 [80x24]" with a blue header bar. The window contains a menu bar with Chinese options: 連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H). The main area of the terminal displays the following session:

```
[kevin@localhost ch13]$./sem1 1 &
[1] 21161
[kevin@localhost ch13]$./sem1
process2 in process2 out
process1 in process1 out
process2 in process2 out
process1 in
21162 - finished
[kevin@localhost ch13]$ process1 out
21161 - finished
```

# Shared Memory Concept

- 共享記憶體是由 IPC 為一程序所建立的特殊記憶體位址，其他的程序可以將此相同的 shared memory 區段納入自己的位址空間中，所有的程序皆可存取這些記憶體位址，就像是由自己定址一樣。



# Shared Memory Function (1/5)

- #include <sys/sem.h>
- #include <sys/type.h>
- #include <sys/ipc.h>
  
- int *shmget(key\_t key, size\_t size, int shmflg);*  
        建立 shared memory 。
- void \*shmat(int *shm\_id*, const void \**shm\_addr*, int *shmflg*);  
        允許程序對 shared memory 存取。
- int shmdt(const void \**shm\_addr*);  
        讓目前的程序從 shared memory 脫離出來。
- int shmctl(int *shm\_id*, int *cmd*, struct shmid\_ds \**buf*);  
        用來改變 shared memory 。

## Shared Memory Function (2/5)

- `int shmget(key_t key, size_t size, int shmflg);`
  - *key*: 用來為 shared memory 命名。
  - *size*: 需要的 shared memory 大小，以 byte 為單位。
  - *shmflg*: shared memory 的權限。在建立新的 shared memory 時要加上 `IPC_CREATE`。
    -

## Shared Memory Function (3/5)

- `void *shmat(int shm_id, const void *shm_addr, int shmflg);`
  - *shm\_id*: Shared memory id.
  - *shm\_addr*: Shared memory 加到目前程序中的位址  
，通常為一 null 指標。
  - *shmflg*: 一般設為 0 即可。

## Shared Memory Function (4/5)

- `int shmdt(const void *shm_addr);`
  - *shm\_addr*: `shmat` 傳回的位址。

# Shared Memory Function (5/5)

- `int shmctl(int shm_id, int cmd, struct shmid_ds *buf);`
  - *shm\_id*: Shared memory id.
  - *cmd*: IPC\_RMID: 刪除 shared memory 區段。
  - struct shmid\_ds {  
    uid\_t shm\_perm.uid;  
    uid\_t shm\_perm.gid;  
    mode\_t shm\_perm.mode;  
}

# Shared Memory Example (1/6)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "shm_com.h"

int main()
{
 int running = 1;
 void *shared_memory = (void *)0;
 struct shared_use_st *shared_stuff;
 int shmid;

 srand((unsigned int)getpid());

 shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

 if (shmid == -1) {
 fprintf(stderr, "shmget failed\n");
 exit(EXIT_FAILURE);
 }

 shared_memory = (void *)shmat(shmid, (void *)0, 0);
 if (shared_memory == (void *)-1)
 perror("shmat error");

 shared_stuff = (struct shared_use_st *)shared_memory;
 if (shared_stuff->written_by_you)
 printf("I see you wrote '%s'\n", shared_stuff->some_text);

 shared_stuff->written_by_you = 1;
 strcpy(shared_stuff->some_text, "I wrote this text\n");

 if (shmdt(shared_memory) == -1)
 perror("shmdt error");
}
```

shm\_com.h

```
#define TEXT_SZ 2048

struct shared_use_st {
 int written_by_you;
 char some_text[TEXT_SZ];
};
```

## Shared Memory Example (2/6)

```
shared_memory = shmat(shmid, (void *)0, 0);
if (shared_memory == (void *)-1) {
 fprintf(stderr, "shmat failed\n");
 exit(EXIT_FAILURE);
}

printf("Memory attached at %X\n", (int)shared_memory);

shared_stuff = (struct shared_use_st *)shared_memory;
shared_stuff->written_by_you = 0;
while(running) {
 if (shared_stuff->written_by_you) {
 printf("You wrote: %s", shared_stuff->some_text);
 sleep(rand() % 4); /* make the other process wait for us ! */
 shared_stuff->written_by_you = 0;
 if (strncmp(shared_stuff->some_text, "end", 3) == 0) {
 running = 0;
 }
 }
}
```

# Shared Memory Example (3/6)

```
if (shmfdt(shared_memory) == -1) {
 fprintf(stderr, "shmfdt failed\n");
 exit(EXIT_FAILURE);
}

if (shmctl(shmid, IPC_RMID, 0) == -1) {
 fprintf(stderr, "shmctl(IPC_RMID) failed\n");
 exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

# Shared Memory Example (4/6)

```
int main()
{
 int running = 1;
 void *shared_memory = (void *)0;
 struct shared_use_st *shared_stuff;
 char buffer[BUFSIZ];
 int shmid;

 shmid = shmget((key_t)1234, sizeof(struct shared_use_st), 0666 | IPC_CREAT);

 if (shmid == -1) {
 fprintf(stderr, "shmget failed\n");
 exit(EXIT_FAILURE);
 }

 shared_memory = shmat(shmid, (void *)0, 0);
 if (shared_memory == (void *)-1) {
 fprintf(stderr, "shmat failed\n");
 exit(EXIT_FAILURE);
 }

 printf("Memory attached at %X\n", (int)shared_memory);
```

# Shared Memory Example (5/6)

```
printf("Memory attached at %X\n", (int)shared_memory);

shared_stuff = (struct shared_use_st *)shared_memory;
while(running) {
 while(shared_stuff->written_by_you == 1) {
 sleep(1);
 printf("waiting for client...\n");
 }
 printf("Enter some text: ");
 fgets(buffer, BUFSIZ, stdin);

 strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
 shared_stuff->written_by_you = 1;

 if (strncmp(buffer, "end", 3) == 0) {
 running = 0;
 }
}
```

```
if (shmrdt(shared_memory) == -1) {
 fprintf(stderr, "shmrdt failed\n");
 exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);
}
```

# Shared Memory Example (6/6)

The screenshot shows a terminal window titled "kevin@localhost:~/linux\_program/ch13 [80x24]" with a blue header bar. The window contains the following text:

```
[kevin@localhost ch13]$./shm1 &
[1] 21231
[kevin@localhost ch13]$ Memory attached at F7035000
./shm2
Memory attached at F706E000
Enter some text: hello
You wrote: hello
waiting for client...
Enter some text: hi
You wrote: hi
waiting for client...
waiting for client...
Enter some text: good
You wrote: good
waiting for client...
waiting for client...
waiting for client...
Enter some text: bye
You wrote: bye
waiting for client...
waiting for client...
Enter some text: end
You wrote: end
[kevin@localhost ch13]$
```

The terminal window has a standard window frame with minimize, maximize, and close buttons in the top right corner. A vertical scroll bar is visible on the right side of the window.

# Message Queue Concept

- Message queue 就像 named pipe 一樣，但沒有開啟與關閉 pipe 的複雜性。不過使用 message queue 也有類似 named pipe 的問題，如 pipe 的阻礙模式。

# Message Queue Function (1/5)

- #include <sys/sem.h>
- #include <sys/type.h>
- #include <sys/ipc.h>
  
- int msgget(key\_t *key*, int *msgflg*);  
    建立與存取 message queue 。
- int msgsnd(int *msqid*, const void \**msg\_ptr*, size\_t *msg\_sz*, int *msgflg*);  
    加入 message 至 message queue 。
- int msgrcv(int *msqid*, void \**msg\_ptr*, size\_t *msg\_sz*, long int *msgtyp*, int *msqflg*);  
    從 message queue 擷取 message 。
- int msgctl(int *msqid*, int *cmd*, struct msqid\_ds \**buf*);  
    用來控制 message queue 。

# Message Queue Function (2/5)

- `int msgget(key_t key, int msgflg);`
  - *key*: 為 message queue 命名。
  - *msgflg*: message queue 的權限。在建立新的 message queue 時要加上 `IPC_CREATE`。

# Message Queue Function (3/5)

- `int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);`
  - *msqid*: Message queue id.
  - *msg\_ptr*: 指向被傳送訊息的指標，必需以為 long int 型態起

始。

```
struct my_message {
 long int my_msg_type; /* message type, must be > 0 */
 char some_text[MAX_TEXT]; /* message data */
};
```

- *msg\_sz*: *msg\_ptr* 的大小，不包括 long int 型態。
- *msgflg*: 控制 message queue 到達下限的反應動作。一般設

為 0 即可。

# Message Queue Function (4/5)

- `int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msqflg);`
  - *msqid*: Message queue id.
  - *msg\_ptr*: 指向接收訊息的指標，必需以為 long int 型態起始。
  - *msg\_sz*: *msg\_ptr* 的大小，不包括 long int 型態。
  - *msgtype*: 設定為 0 則從 queue 中取得第一個訊息；設定大於 0 則與 queue 中第一個訊息相同型態的訊息將全被取回；小於 0 則第一個與 *msgtype* 值相同或小於之訊息將被取回。
  - *msqflg*: 一般設為 0。

# Message Queue Function (5/5)

- `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`
  - *msqid*: Message queue id.
  - *cmd*: IPC\_RMID: 刪除 message queue。
  - struct msqid\_ds {  
    uid\_t msg\_perm.uid;  
    uid\_t msg\_perm.gid;  
    mode\_t msg\_perm.mode;  
}

# Message Queue Example (1/5)

```
int main()
{
 int running = 1;
 int msgid;
 struct my_msg_st some_data;
 long int msg_to_receive = 0;

 msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

 if (msgid == -1) {
 fprintf(stderr, "msgget failed with error: %d\n", errno);
 exit(EXIT_FAILURE);
 }
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct my_msg_st {
 long int my_msg_type;
 char some_text[BUFSIZ];
};
```

# Message Queue Example (2/5)

```
while(running) {
 if (msgrecv(msqid, (void *) &some_data, BUFSIZ,
 msg_to_receive, 0) == -1) {
 fprintf(stderr, "msgrecv failed with error: %d\n", errno);
 exit(EXIT_FAILURE);
 }
 printf("You wrote: %s", some_data.some_text);
 if (strncmp(some_data.some_text, "end", 3) == 0) {
 running = 0;
 }
}

if (msgctl(msqid, IPC_RMID, 0) == -1) {
 fprintf(stderr, "msgctl(IPC_RMID) failed\n");
 exit(EXIT_FAILURE);
}

exit(EXIT_SUCCESS);
}
```

# Message Queue Example (3/5)

```
int main()
{
 int running = 1;
 struct my_msg_st some_data;
 int msgid;
 char buffer[BUFSIZ];

 msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

 if (msgid == -1) {
 fprintf(stderr, "msgget failed with error: %d\n", errno);
 exit(EXIT_FAILURE);
 }
}
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512

struct my_msg_st {
 long int my_msg_type;
 char some_text[MAX_TEXT];
};
```

## Message Queue Example (4/5)

```
while(running) {
 printf("Enter some text: ");
 fgets(buffer, BUFSIZ, stdin);
 some_data.my_msg_type = 1;
 strcpy(some_data.some_text, buffer);

 if (msgsnd(msgid, (void *) &some_data, MAX_TEXT, 0) == -1) {
 fprintf(stderr, "msgsnd failed\n");
 exit(EXIT_FAILURE);
 }
 if (strncmp(buffer, "end", 3) == 0) {
 running = 0;
 }
}

exit(EXIT_SUCCESS);
}
```

## Message Queue Example (5/5)

The screenshot shows a terminal window titled "kevin@localhost:~/linux\_program/ch13 [80x24]" with a blue header bar. The window contains the following text:

```
連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)
[kevin@localhost ch13]$./msg2
Enter some text: hi
Enter some text: hello
Enter some text: good
Enter some text: end
[kevin@localhost ch13]$./msg1
You wrote: hi
You wrote: hello
You wrote: good
You wrote: end
[kevin@localhost ch13]$
```

# Homework

- 請在 O.S 中的生產者與消費者問題加入同步機制。

