# 系統程式設計
# Systems Programming


鄭卜壬教授
臺灣大學資訊工程系




Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Contents




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Process Control

- **Objective**
  - Process Control
    - process creation and termination, program execution, etc.
  - Process Properties and Accounting
    - e.g., ID's.
- **Process Identifiers**
  - Process ID – a nonnegative unique integer
    - tmpnam



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Booting

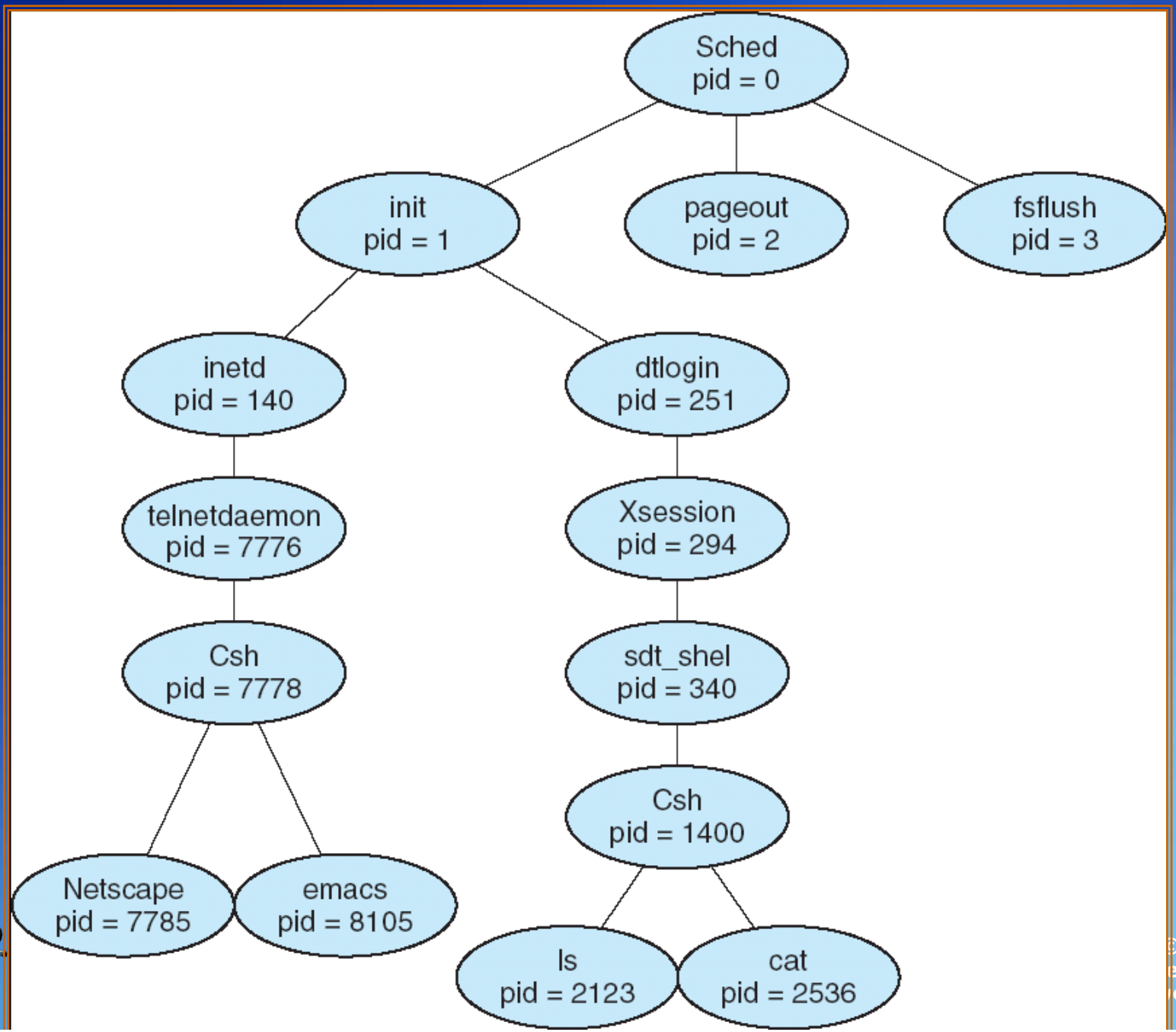
- **Bootstrapping**
  - The process of starting up a computer from a powered-down condition (cold boot)
  - Initializing all aspects of the system, e.g., CPU registers, device controllers, memory, etc.
  - The memory-resident code reads the boot program from the boot device
  - Boot program reads in the kernel and passes control to it.
  - Kernel identifies and configures the devices.
  - Initializes the system and starts the system processes.
  - Brings up the system in single-user mode (if necessary).
  - Runs the appropriate startup scripts.
  - Brings up the system for multi-user operation.

# Process Control

- **Special Processes**
  - PID 0 – *Swapper* (or scheduler)
    - Kernel process
    - No program on disks correspond to this process
  - PID 1 – *init* responsible for bringing up a Unix system after the kernel has been bootstrapped.
    - User process with superuser privileges
    - Initializing system processes, e.g., various daemons, login processes, etc.
    - /etc/rc* or /sbin/rc*
  - PID 2 - pagedaemon responsible for paging
    - Kernel process


Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

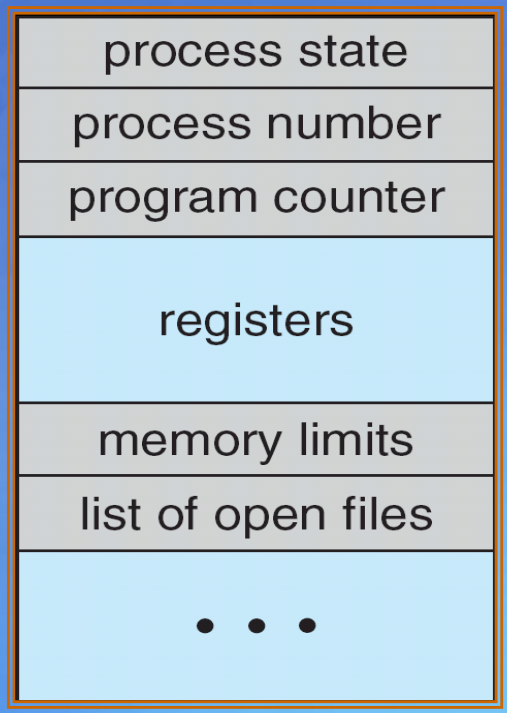
# A Tree of Processes on a Typical Solaris



# Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

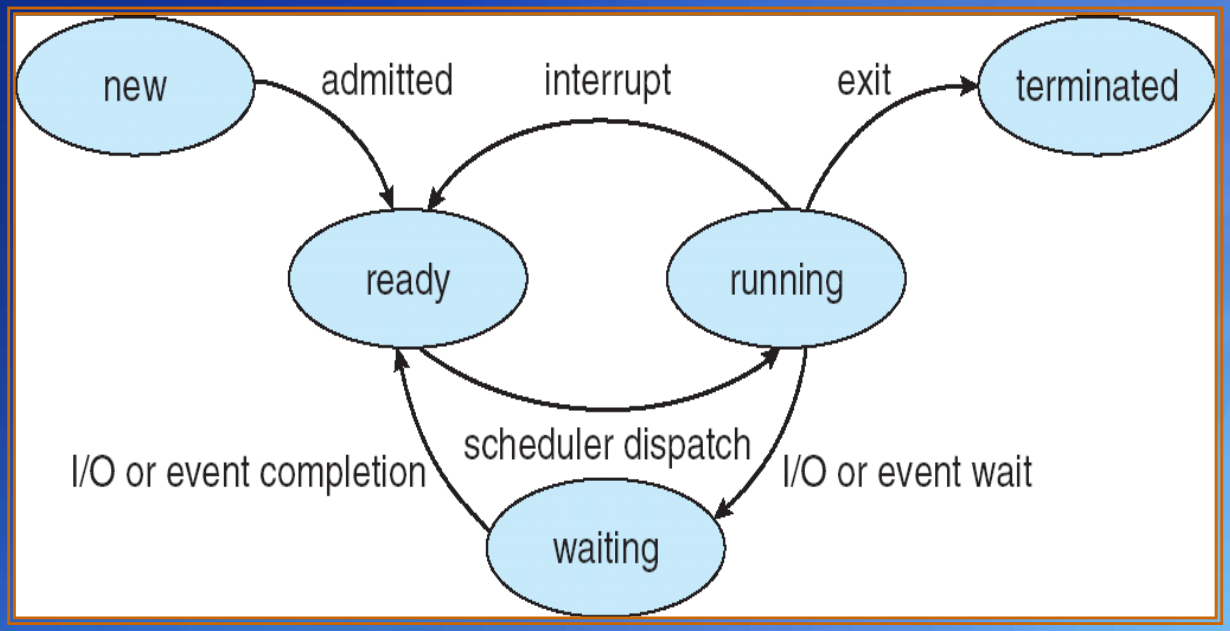
- sys/proc.h
- Some of the fields of the *proc* structure

| proc structure field | | Description |
|---|---|---|
| char | p_stat | The status of the process (running, sleeping, etc.). |
| char | p_pri | The priority of the process. |
| unsigned int | p_flag | Flags used in conjunction with the process status (p_stat) to determine process specific actions or actions which may affect the process. |
| unsigned short | p_uid | The user ID of the process. |
| unsigned short | p_suid | The effective user ID of the process. |
| int | p_sid | The process session ID. |
| short | p_pgrp | ID of the process group leader. |
| short | p_pid | The process ID. |
| short | p_ppid | The parent's process ID. |
| sigset_t | p_sig | The current signal which the kernel is processing. |
| unsigned int | p_size | Process size in pages. |
| time_t | p_utime | Time spent in user mode in seconds. |
| time_t | p_stime | Time spent in system mode in seconds. |
| caddr_t | p_ldt | A pointer to the process's LDT. |
| struct pregion | *p_region | List of per process memory regions. |
| short | p_xstat | Exit status to be returned to the parent. |
| unsigned int | p_uptbl[] | An array of page table entries for the u_area. |

# Process States

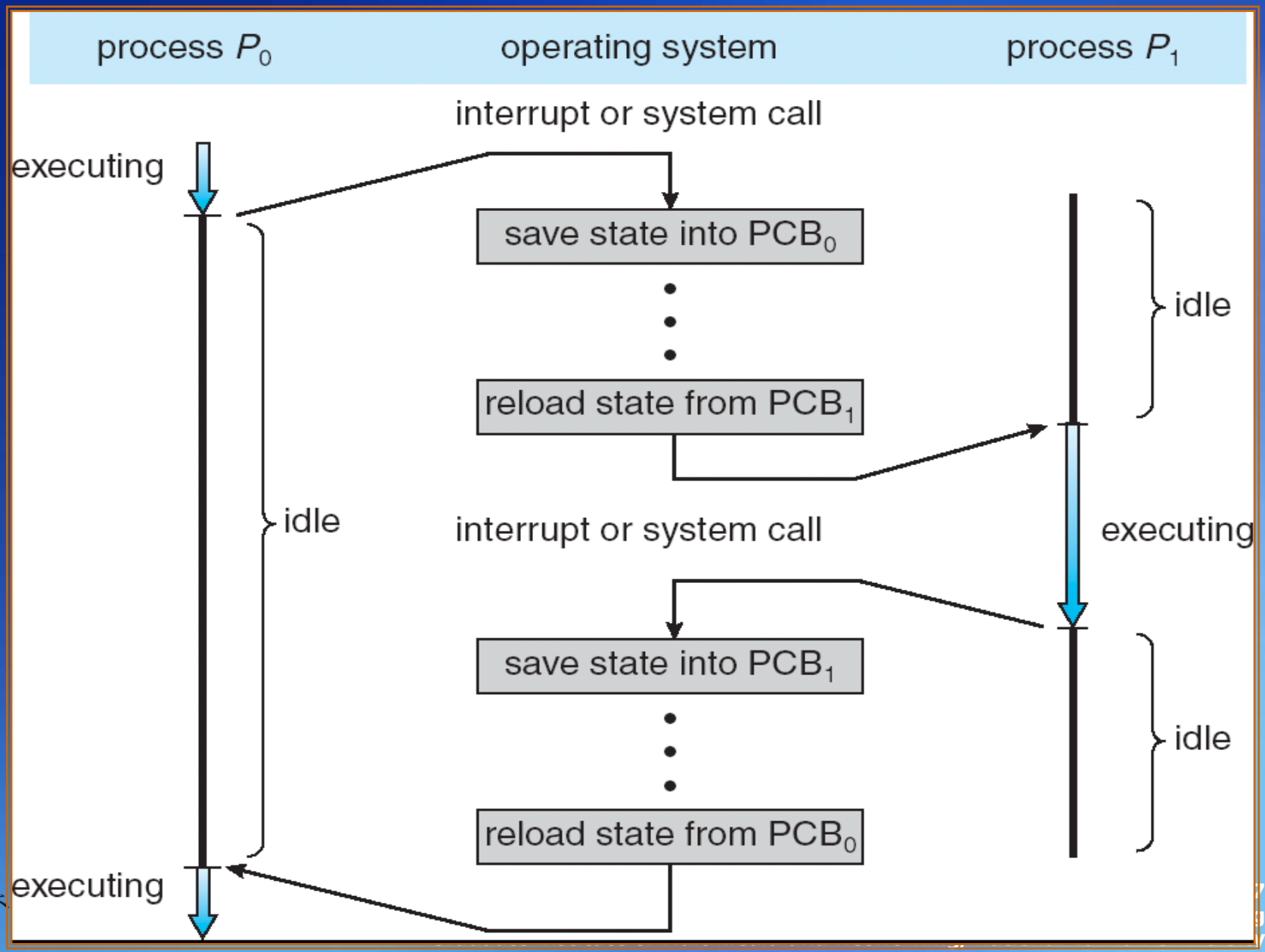
- New
  - The process is being created
- Running
  - Instructions are being executed
- Waiting
  - The process is waiting for some event to occur
- Ready
  - The process is waiting to be assigned to a processor
- Terminated
  - The process has finished execution

# Process States




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# CPU Switch between Processes



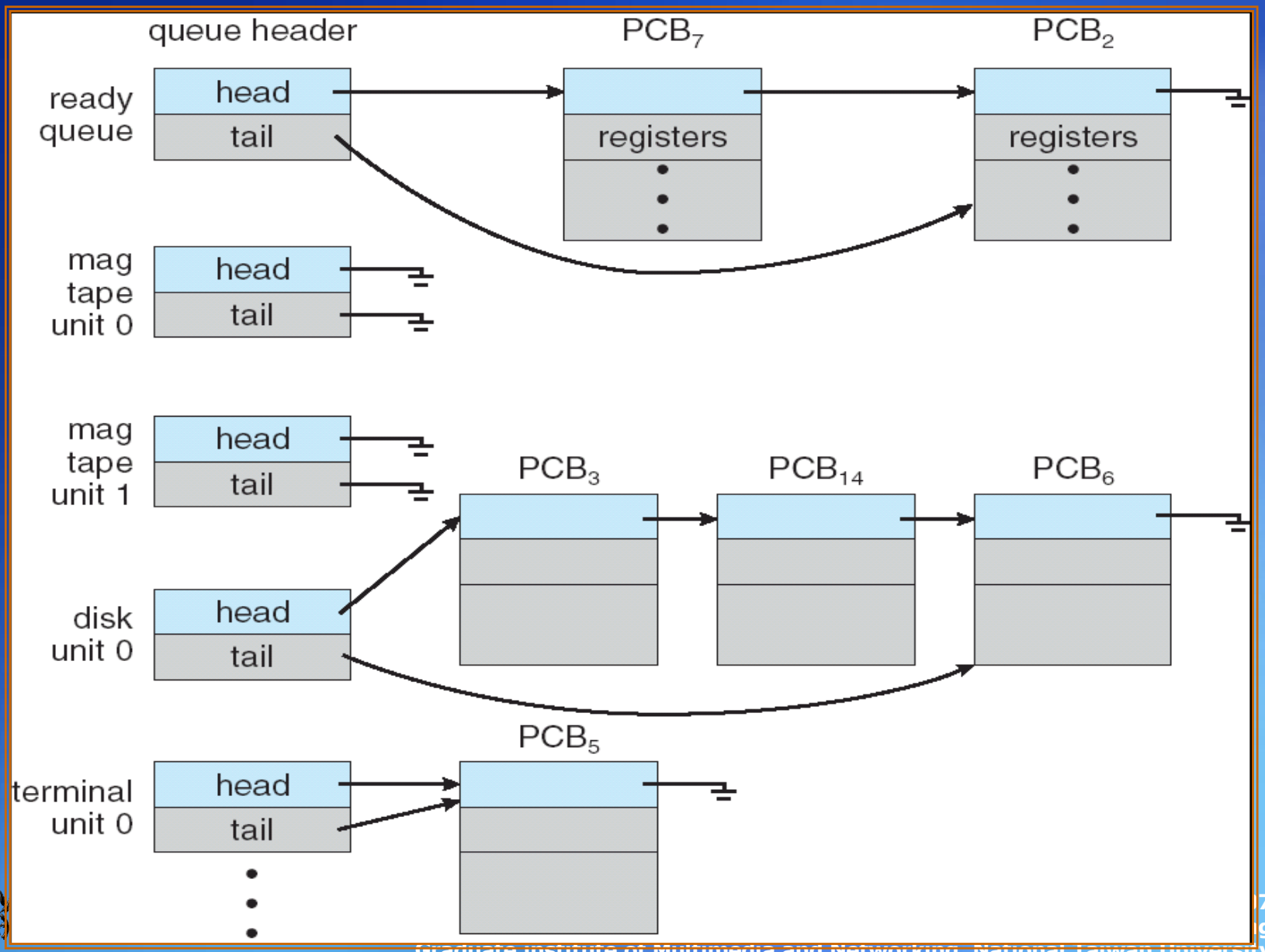
# Process Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue – set of all processes residing in main memory, ready and waiting to execute
- Device queues – set of processes waiting for an I/O device
- Processes migrate among the various queues

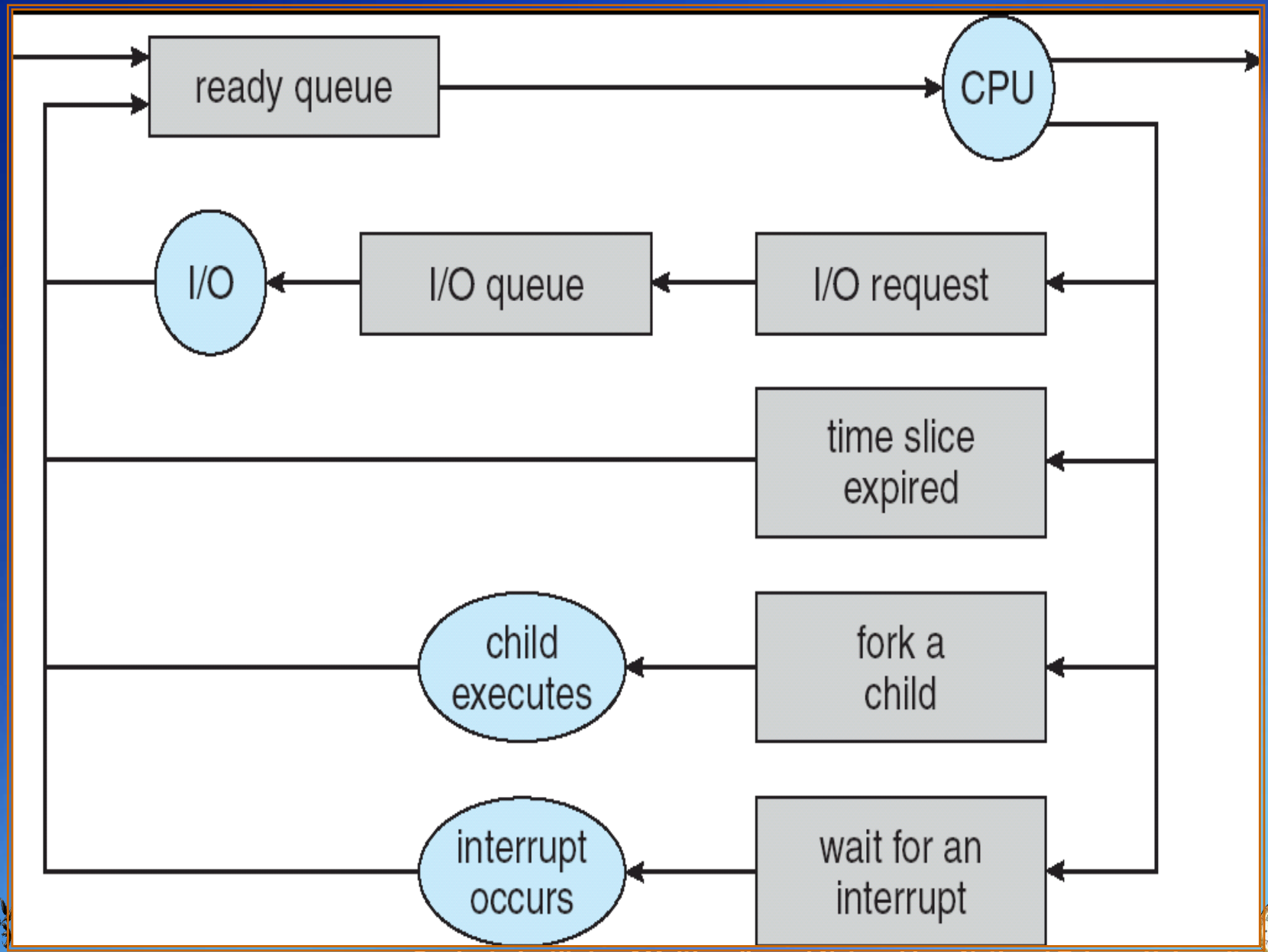


Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Ready Queue & I/O Queues



# Process Scheduling

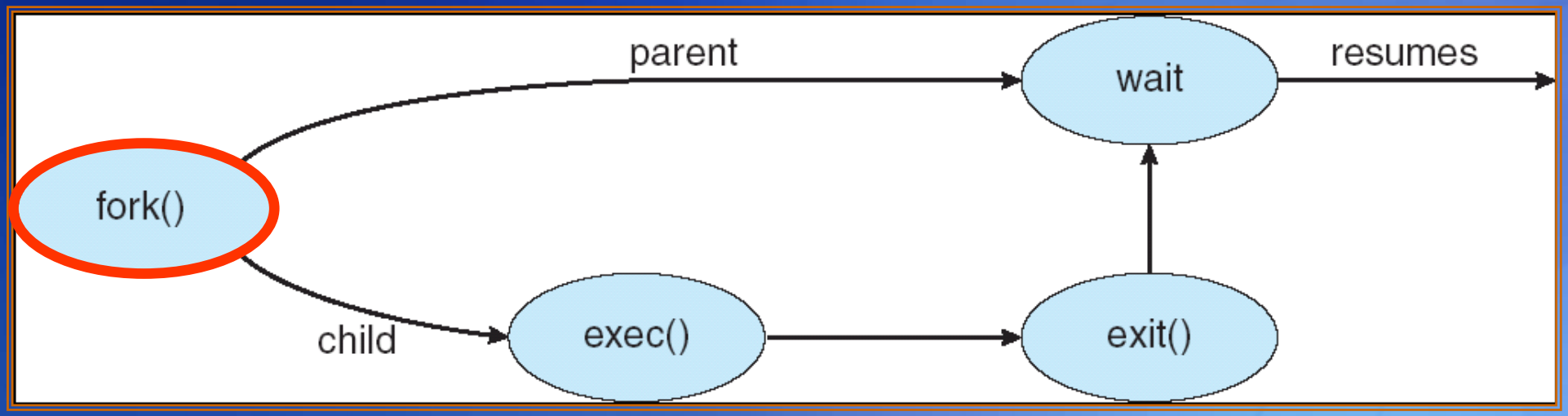


# Process Control

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void)
pid_t getppid(void)
uid_t getuid(void)
uid_t geteuid(void)
gid_t getgid(void)
gid_t getegid(void)
```

- None of them has an error return.

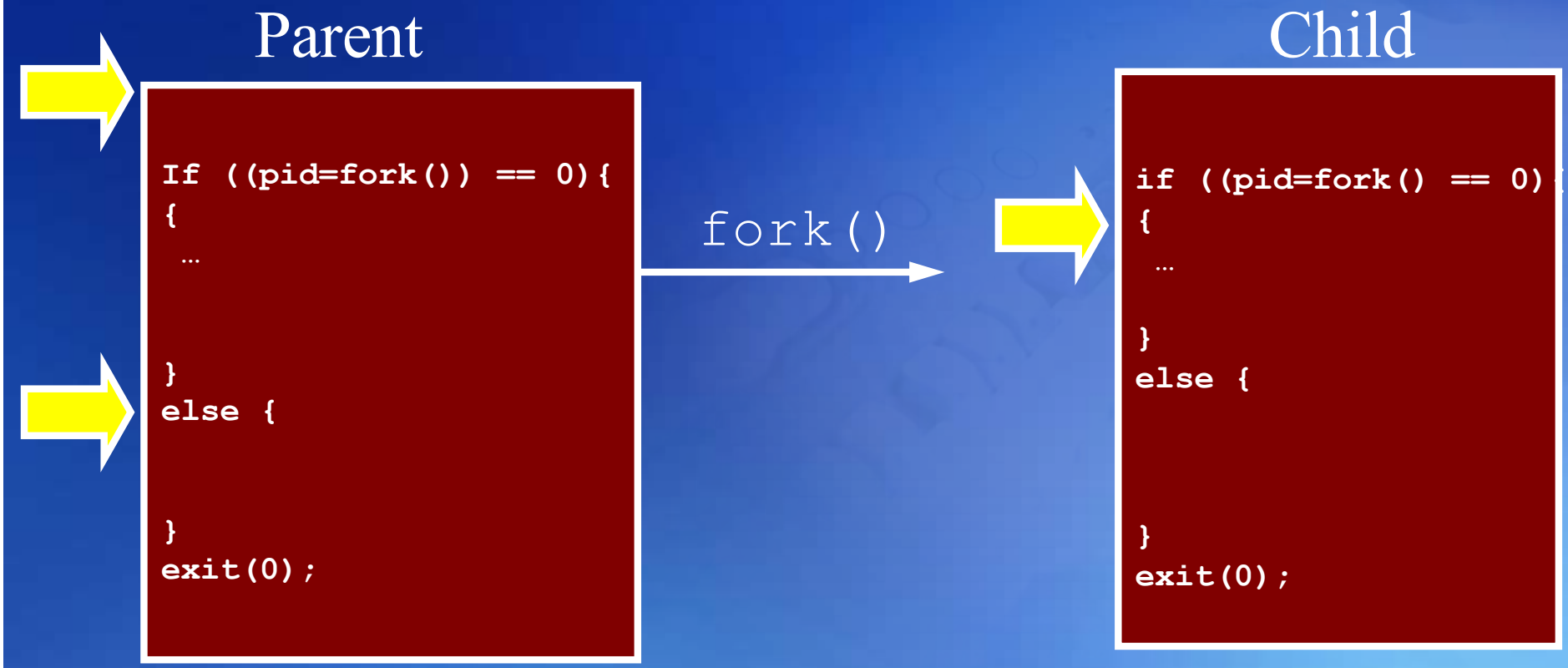
# Process Creation & Termination




The way to have multiple processes

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# What's a `fork()`




- Child is an exact copy of the parent process.
- They have their own memory space.

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

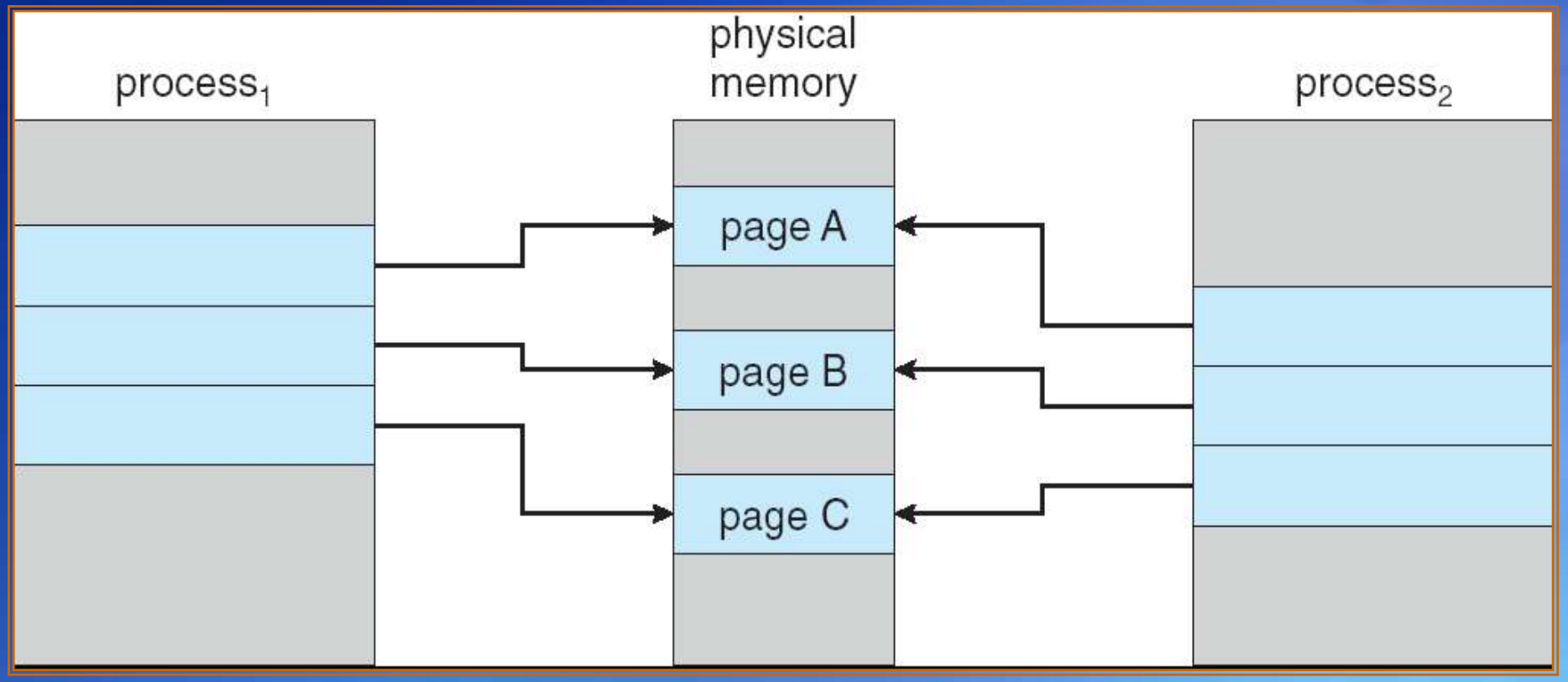
# fork

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

## pid_t fork(void)

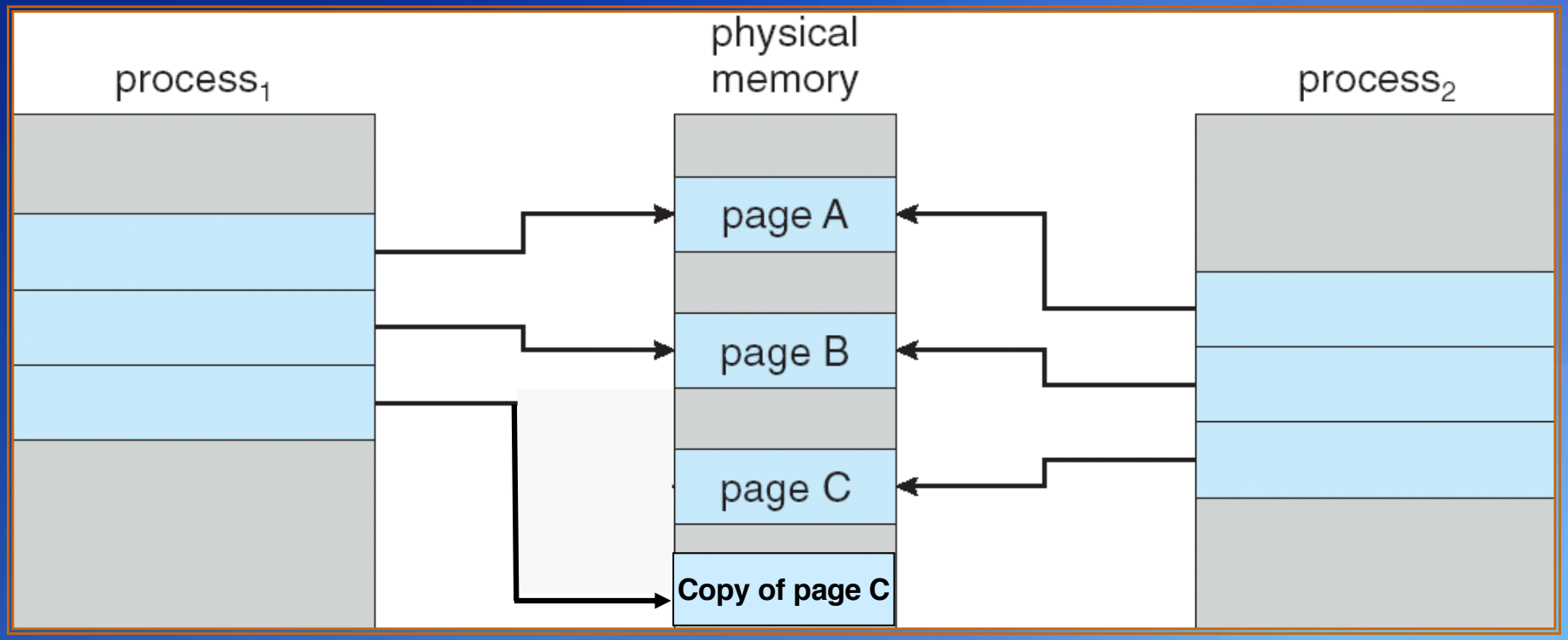
- The only way beside the bootstrap process to create a new process.
- Call once but return twice
  - 0 for the child process (getppid)
  - Child pid for the parent (1:n)
- Cannot predict which process runs first
  - Process scheduling
- Copies of almost everything but no sharing of memory, except text
  - Copy-on-write

# Before Process 1 Modifies Page C




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# After Process 1 Modifies Page C

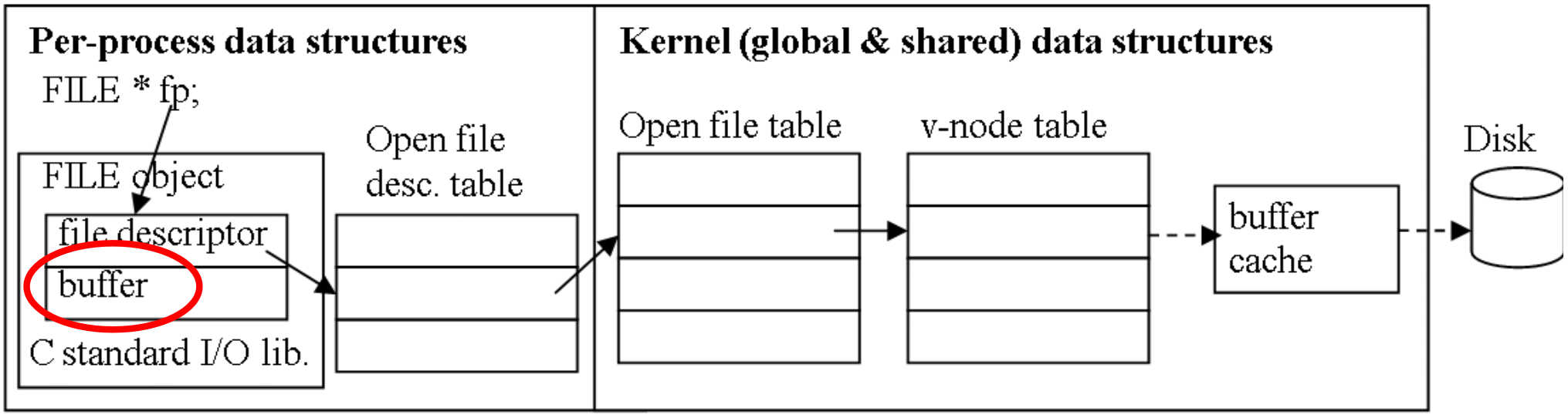



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Example of fork()

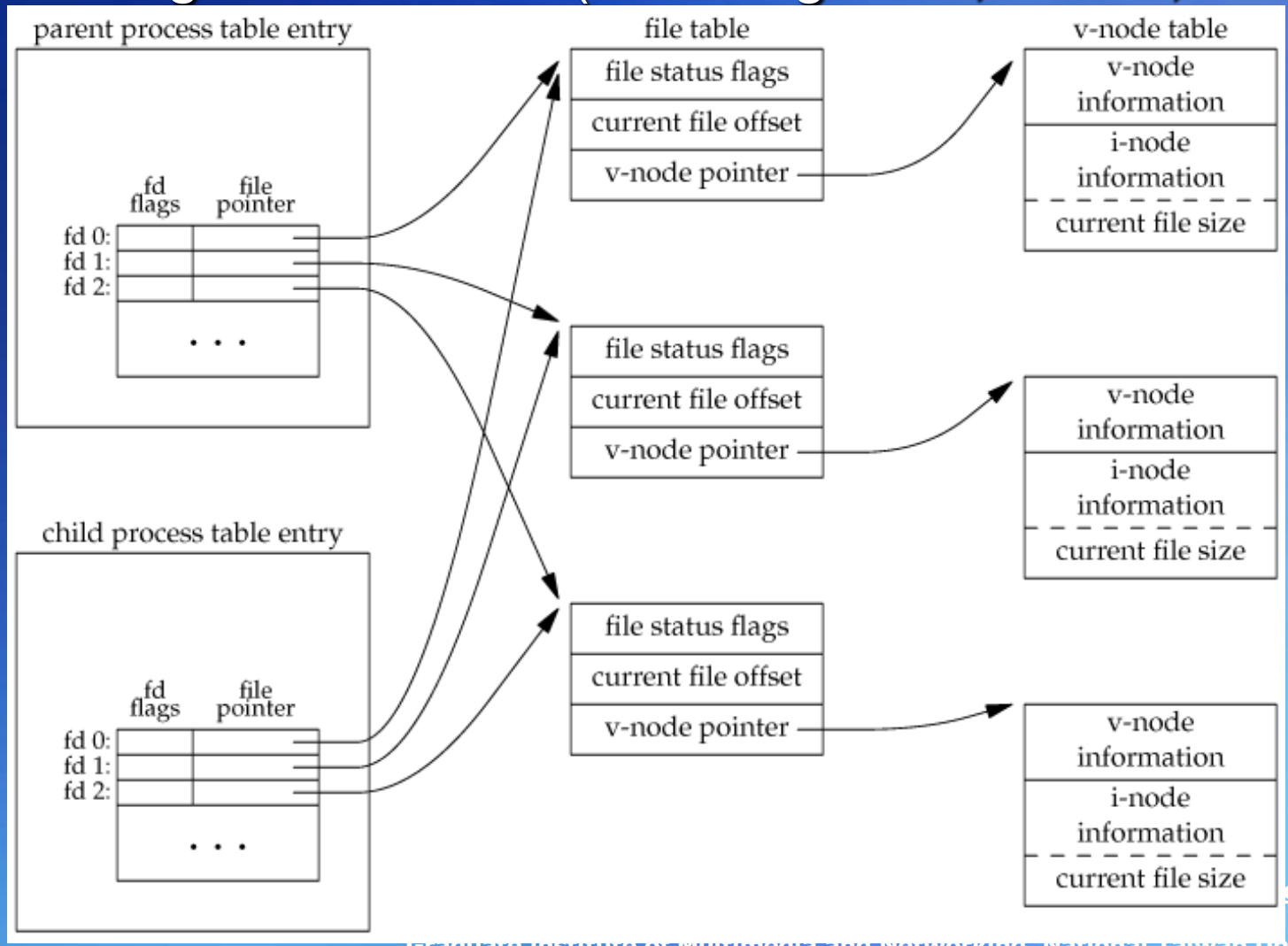
```
int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int         var;        /* automatic variable on the stack */
    pid_t       pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");     /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {       /* child */
        glob++;                  /* modify variables */
        var++;
    } else {
        sleep(2);                /* parent */
    }

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

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89        child's variables were changed
pid = 429, glob = 6, var = 88        parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```



# fork

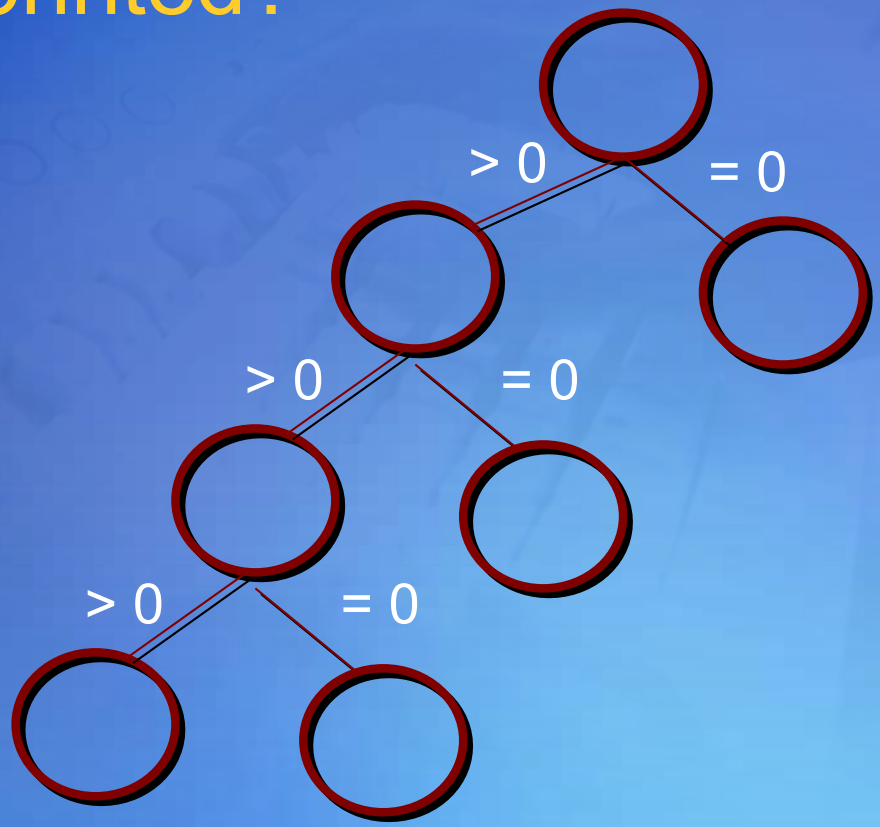
- File sharing
  - Sharing of file offsets (including stdin, stdout, stderr)



# Practice 1

- How many '*' will be printed?

```
void forkfunc() {
    if (fork () && fork ())
        fork ();
    write (1, "*\n", 2);
}
```




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Practice 2

- How many ‘*’ will be printed?

```
void forkfunc() {
    if (fork () || fork ())
        fork ();
    write (1, “*\n”, 2);
}
```

# fork

- **Normal cases in fork:**
  - The parent waits for the child to complete.
  - The parent and child each go their own way (e.g., network servers).
- **Inherited properties:**
  - Real uid, effective uid, real gid, effective gid, supplementary gid, process group id, session id, controlling terminal, set[ug]id flag, current working dir, root dir, file-mode creation mask, signal mask & dispositions, close-on-exec flag, environment, attached shared memory segments, resource limits
- **Differences on properties:**
  - Returned value from fork, process id, parent pid, tms_[us]time, tms_c[us]time, file locks, pending alarms, pending signals

# fork

- **Reasons for fork to fail**
  - Too many processes in the system
  - The total number of processes for the real uid exceeds the limit
    - CHILD_MAX
- **Usages of fork**
  - Duplicate a process to run different sections of code
    - Network servers
  - Want to run a different program
    - shells (spawn = fork+exec)

# vfork

- **Design Objective**
  - An optimization on performance
    - No fully copying of the parent's address space into the child.
    - Sharing of address space
  - Execute exec() right after returns from fork().
- **Mechanism – SVR4 & 4.3+BSD**
  - Since 4BSD
    - <vfork.h> in some systems

# vfork

- **vfork() is as the same as fork() except**
  - The child runs in the address space of its parent.
  - The parent waits until the child calls exit() or exec().
    - Child process always executes first.
    - A possibility of deadlock if the child waits for the parent to do something before calling exec().

# Example of vfork()

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

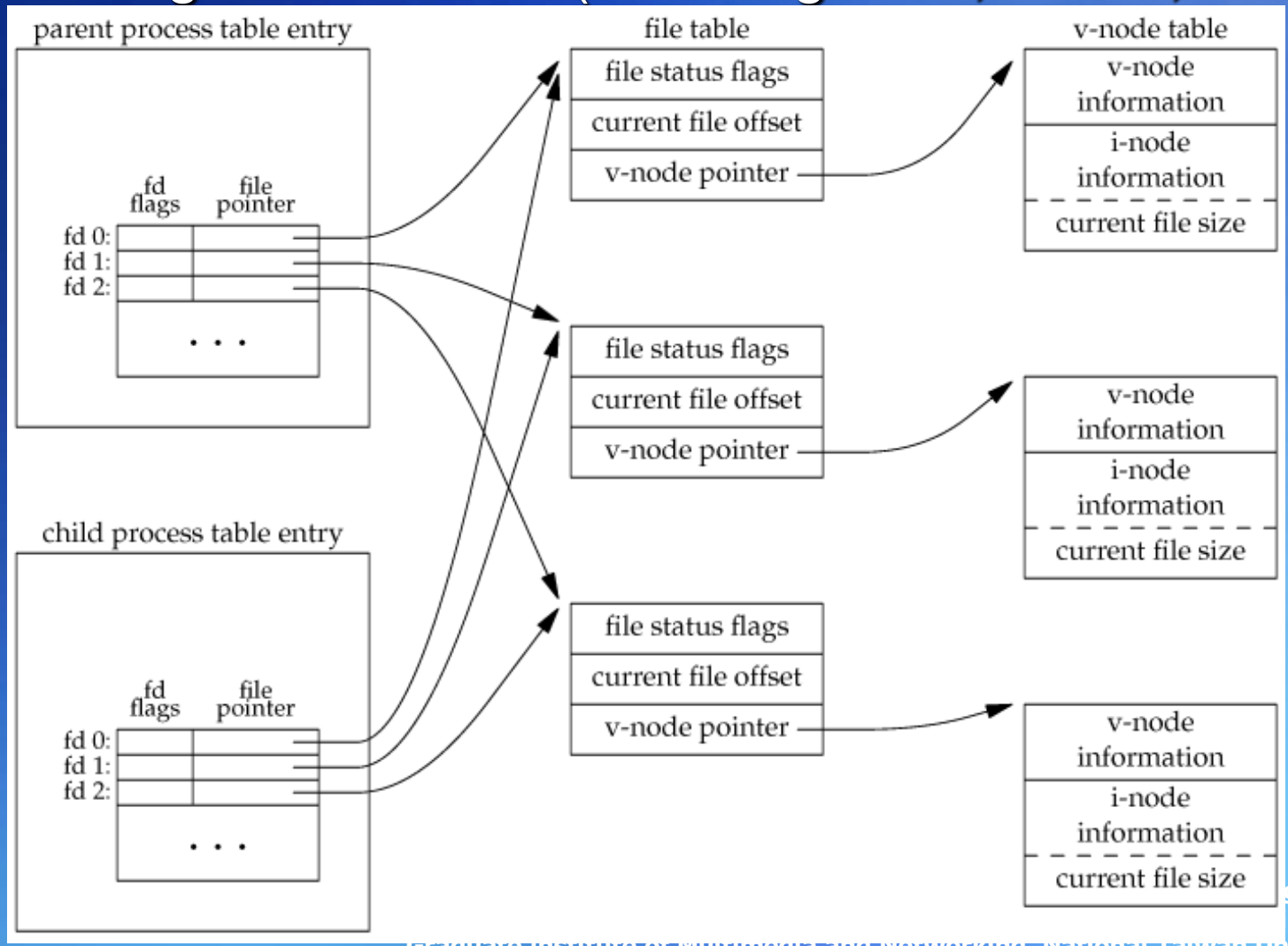
```
int     glob = 6;       /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify parent's variables */
        var++;
        _exit(0);               /* child terminates */
    }
    /*
     * Parent continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

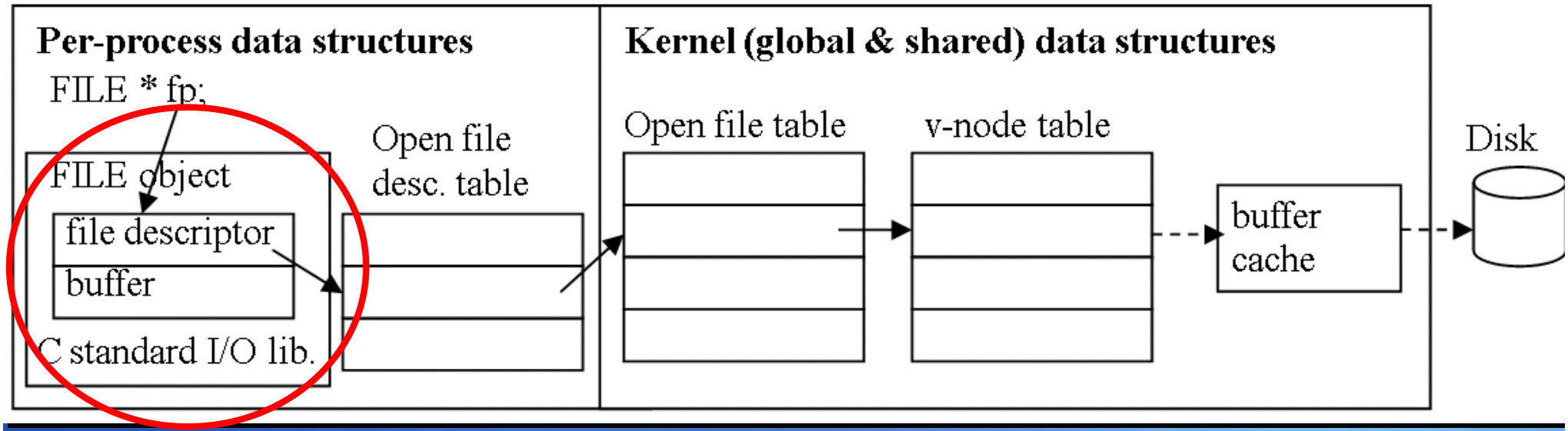
exit() may close the file descriptors, causing printf() to fail.

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# fork

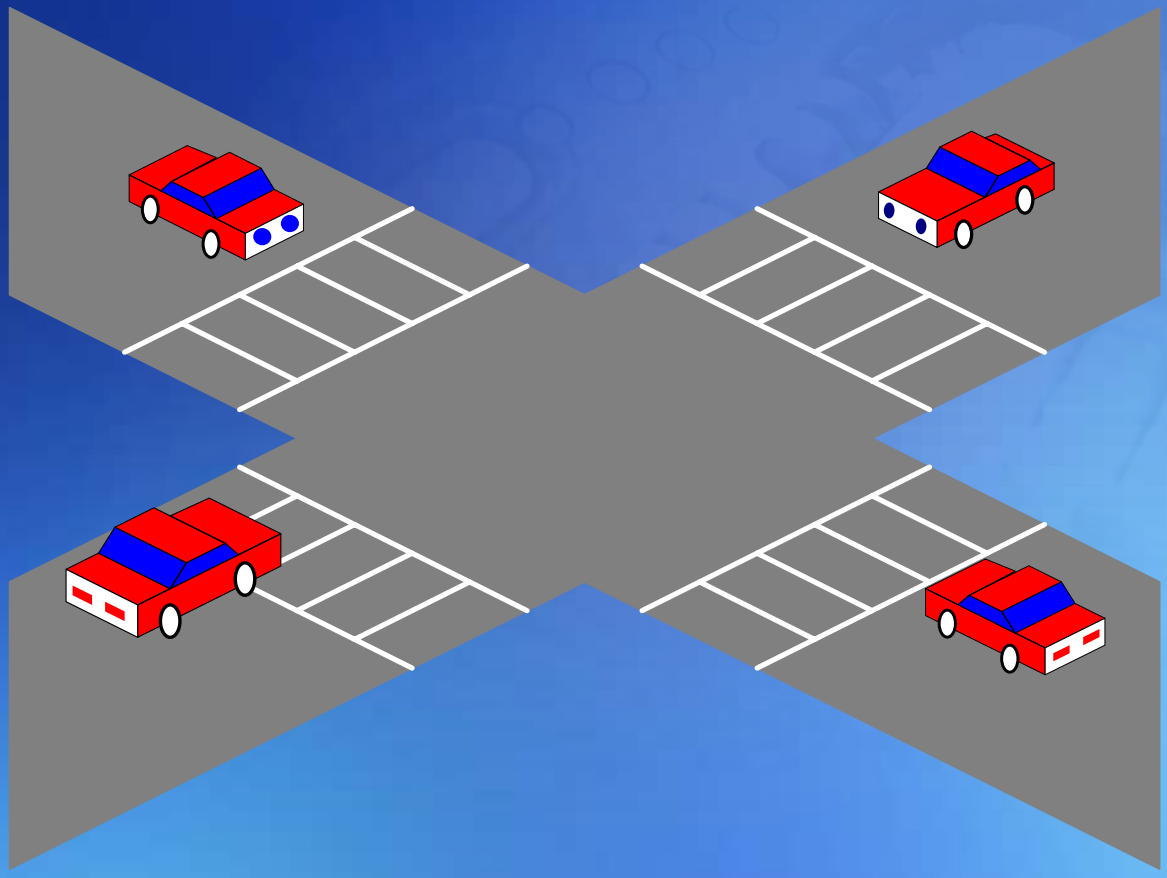
- File sharing
  - Sharing of file offsets (including stdin, stdout, stderr)




Per-process data structures
FILE * fp;
FILE object
file descriptor
buffer
C standard I/O lib.
Open file
desc. table
Kernel (global & shared) data structures
Open file table
v-node table
buffer
cache
Disk

# Deadlock

**Deadlock** is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.



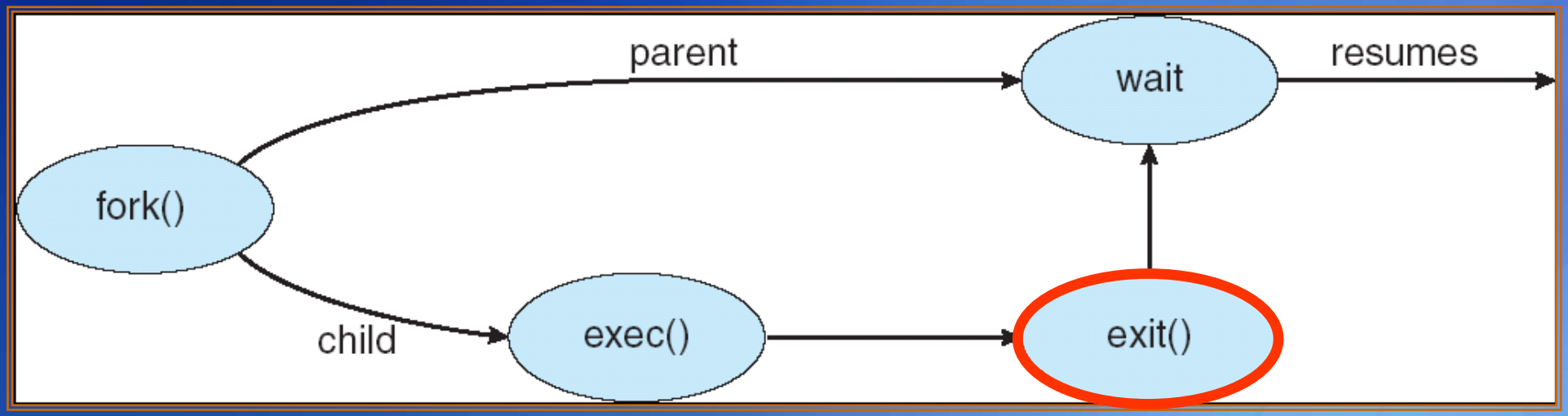
Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Deadlock – One Example

```
int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;
    printf("before vfork\n");   /* we don't flush stdio */
    if ((pid = vfork()) < 0) {
        err_sys("vfork error");
    } else if (pid == 0) {      /* child */
        while (var<100 );       /* parent's variables */
        _exit(0);               /* child terminates */
    }
    /* Parent continues here. */
    var = 100;
    exit(0);
}
```

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Process Creation & Termination

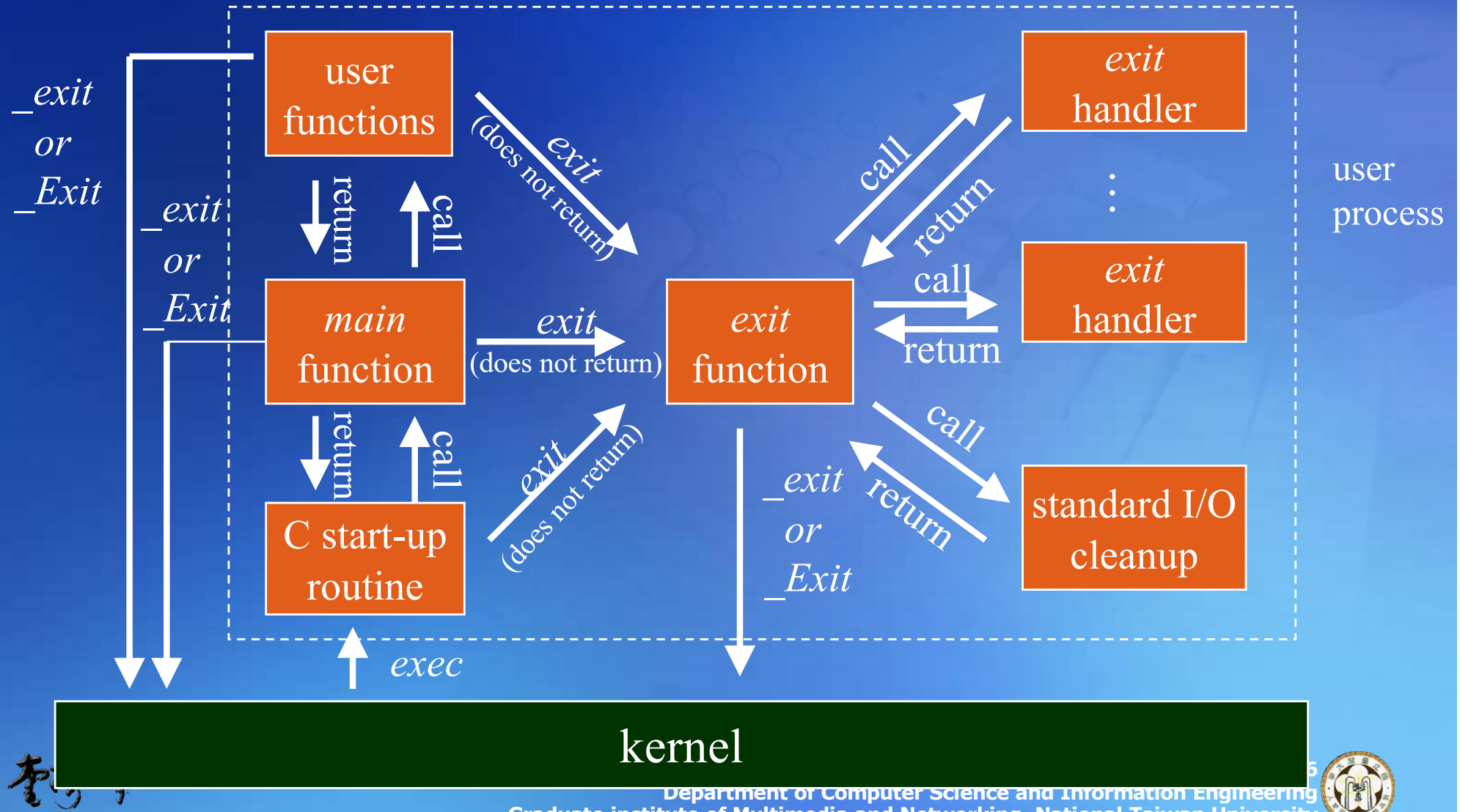



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Process Termination (Ch7.3)

- **Eight ways to terminate:**
  - Normal termination
    - Return from main()
      - exit(main(argc, argv));
    - Call `exit()`
    - Call `_exit()` or `_Exit()`
    - Return of the last thread from its start routine
    - Calling `pthread_exit` from the last thread.
  - Abnormal termination (Chapter 10)
    - Call abort() (generating the SIGABRT signal)
    - Be terminated by a signal
    - Response of the last thread to a cancellation request.

# How a C program is started and terminated.



# Process Termination

**#include <stdlib.h>**

**void exit(int status)**

- Perform cleanup processing
  - Call exit handlers
  - Close and flush I/O streams (fclose)
  - Called once, never returns
  - Puts the process into “zombie” status

**void _Exit(int status)**

- ANSI C

**#include <unistd.h>**

**void _exit(int status)**

- POSIX.1

```
#include <stdio.h>
main() {
   printf(“hello, world\n”);
}
```

```
$ cc -std=c99 hello.c          enable gcc's 1999 ISO C exte
hello.c:4: warning: return type defaults to 'int'
$ ./a.out
hello, world
$ echo $?
0
```

- **Exit status:**
  - Undefined exit status:
    - Exit/return without status.
    - main() is not declared to be an integer.
  - 0 as the status:
    - main() is declared to be an integer and main() falls off the end.
    - return(0) and exit(0).

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Process Termination

```
#include <stdlib.h>
int atexit(void (*func)(void))
```

- At least 32 functions called by *exit* – ANSI C, supported by SVR4&4.3+BSD
- The same exit functions can be registered for several times.
- Exit functions/handlers will be called in reverse order of their registration.
- Exit handlers registered will be cleared if exec() is called.

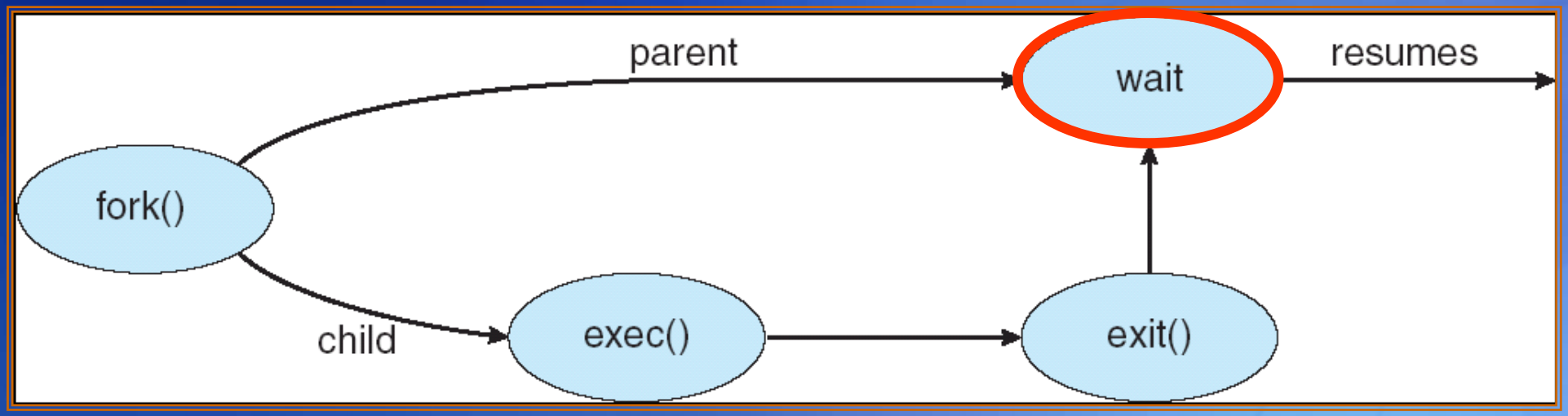

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Example of atexit()

```
static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
      if (atexit(my_exit2) != 0)
          err_sys("can't register my_exit2");

      if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");

      if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");

      printf("main is done\n");
      return(0);
}

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

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

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```

# Process Creation & Termination




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# wait & waitpid

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

**Called by the parent process to retrieve the termination status of its child process**

- wait will block until one child terminates or an error could be returned.
- waitpid could wait for a specific one and has an option not to be blocked.

- **SIGCHILD from the kernel if a child terminates**
  - Default action is ignoring.

# wait & waitpid

- **Three situations in calling wait/waitpid**
  - Block
  - Return with the termination status of a child
  - Return with an error.
- **Termination Status <sys/wait.h>**
  - Exit status (WIFEXITED, WEXITSTATUS)
  - Signal # (WIFSIGNALED, WTERMSIG)
  - Core dump (WCOREDUMP)
  - Others (WIFSTOPPED, WSTOPSIG)

| Macro | Description |
|---|---|
| WIFEXITED(*status*) | True if status was returned for a child that terminated normally. In this case, we can execute<br><br>WEXITSTATUS (*status*)<br><br>to fetch the low-order 8 bits of the argument that the child passed to exit, _exit,or _Exit. |
| WIFSIGNALED (*status*) | True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute<br><br>WTERMSIG (*status*)<br><br>to fetch the signal number that caused the termination.<br><br>Additionally, some implementations (but not the Single UNIX Specification) define the macro<br><br>WCOREDUMP (*status*)<br><br>that returns true if a core file of the terminated process was generated. |
| WIFSTOPPED (*status*) | True if status was returned for a child that is currently stopped. In this case, we can execute<br><br>WSTOPSIG (*status*)<br><br>to fetch the signal number that caused the child to stop. |
| WIFCONTINUED (*status*) | True if status was returned for a child that has been continued after a job control stop (XSI extension to POSIX.1; waitpid only). |

# wait & waitpid

## pid_t waitpid(pid_t pid, int *statloc, int op);

- pid
  - pid == -1 → wait for any child
  - pid > 0 → wait for the child with pid
  - pid == 0 → wait for any child with the same group id
  - pid < -1 → wait for any child with the group ID = |pid|
- pid of the child or an error is returned.

wait(&status) == waitpid(-1, &status, 0)

```c
#include "apue.h"
#include <sys/wait.h>

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

# wait & waitpid

- Errors
  - No such child or wrong parent
- Option for waitpid
  - WNOHANG, WUNTRACED
  - WNOWAIT, WCONTINUED (SVR4)

| Constant | Description |
|---|---|
| WCONTINUED | If the implementation supports job control, the status of any child specified by *pid* that has been continued after being stopped, but whose status has not yet been reported, is returned (XSI extension to POSIX.1). |
| WNOHANG | The waitpid function will not block if a child specified by *pid* is not immediately available. In this case, the return value is 0. |
| WUNTRACED | If the implementation supports job control, the status of any child specified by *pid* that has stopped, and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process. |

# Different exit status

```
int
main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        exit(7);

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        abort();                    /* generates SIGABRT */

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

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)              /* child */
        status /= 0;                /* divide by 0 generates SIGFPE */

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

    exit(0);
}
```

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

# Zombie & Orphan

- **Zombie**
  - The process has terminated, but its parent has not yet waited for it.
  - It keeps the minimal information (e.g., process ID, termination status, and CPU time taken) for the parent process.
- **What if the parent process terminates before the child process? (orphan)**
  - Inherited by init
    - When a parent terminates, it is done by the kernel.
    - Clean up of the zombies by the init – wait whenever needed!

# Discussion: why do we need zombie?

- **A parent can fetch its child's termination status**
- **Without zombie state, the relationship between parent and children may miss**
  - A parent process can end up with two different children that share the same PID.
  - A parent process can end up trying to wait for the return code of another process's child
  - …

# Troubles from Zombie

- **Although zombie processes do not occupy any memory space in the system, it consumes process IDs, which are limited resources too.**
  - You may not be able to fork new child processes when there are too many zombie processes in the systems.
- **How to fork a new child process but**
  - not asking the parent process to wait for the child AND
  - not generating zombie process?

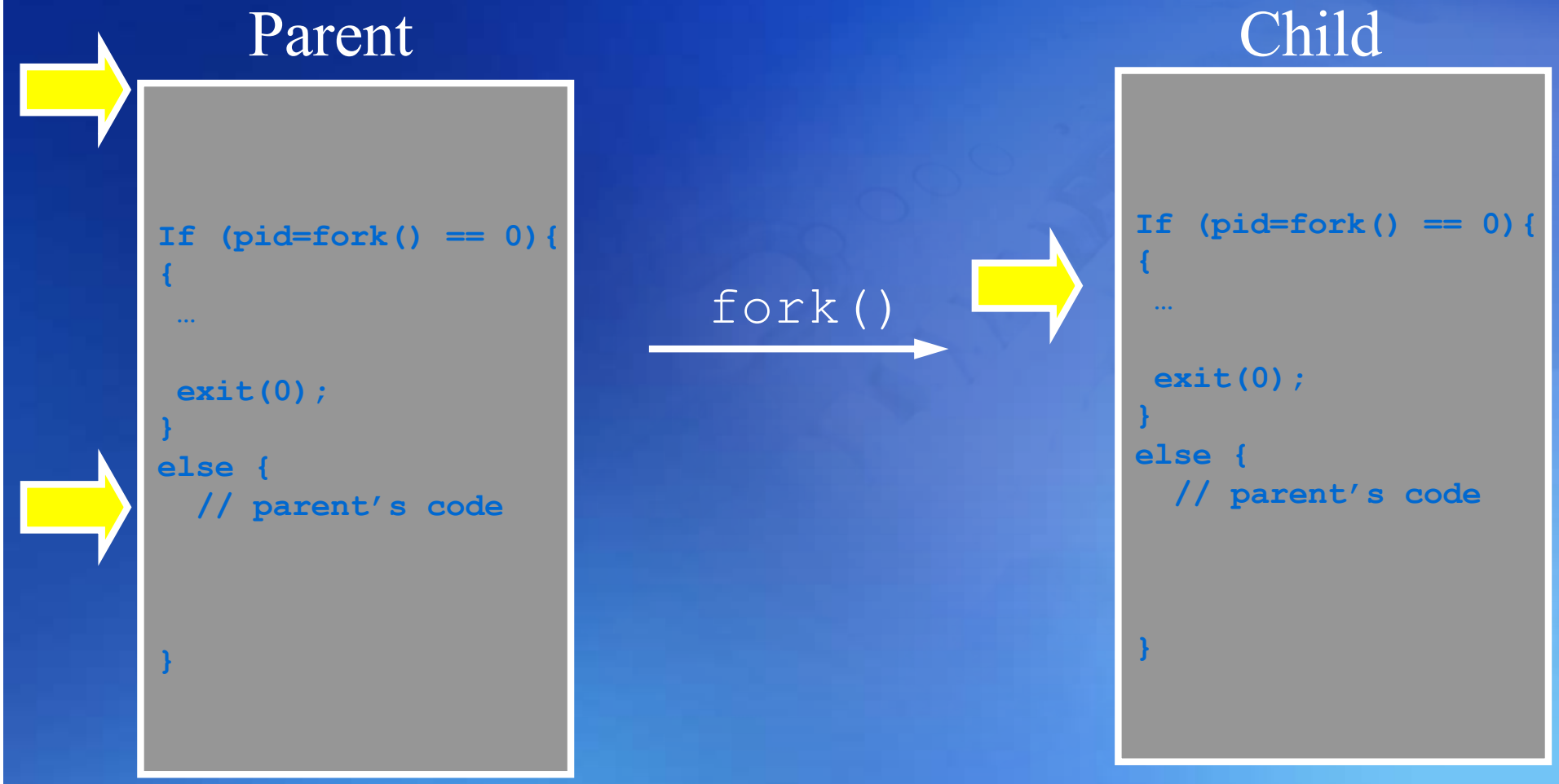
p

```
int
main(void)
{
    pid_t   pid;

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

    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

```
$ ./a.out
$ second child, parent pid = 1
```



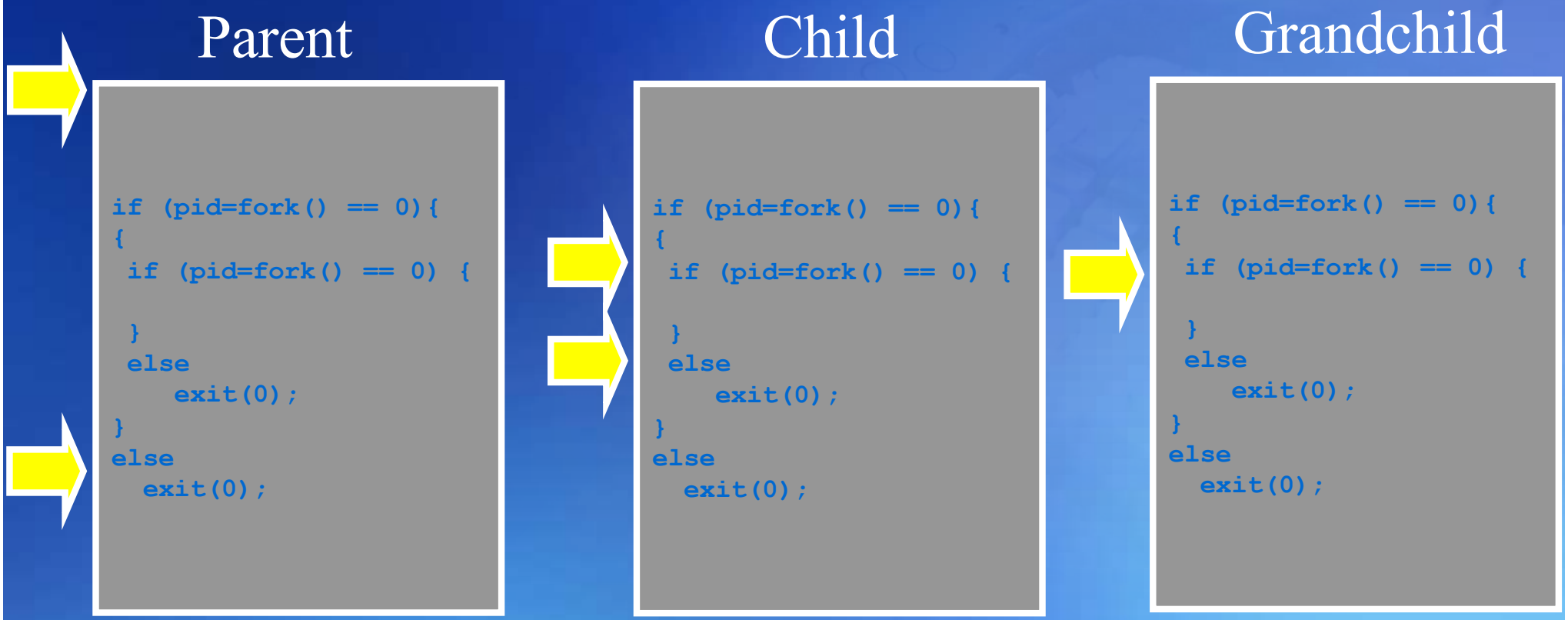
# How to create a background process without making a zombie?




We get a zombie!!

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# How to create a background process without making a zombie?




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# waitid

```
#include <sys/wait.h>
pid_t waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- Defined in XSI extension
- Similar to waitpid() but provide extra info.
- idtype:
  - P_PID: wait for a particular process
  - P_PGID: wait for a group of process
  - P_ALL: wait for any child process
- options:

| Constant | Description |
|---|---|
| WCONTINUED | Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported. |
| WEXITED | Wait for processes that have exited. |
| WNOHANG | Return immediately instead of blocking if there is no child exit status available. |
| WNOWAIT | Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to wait, waitid,or waitpid. |
| WSTOPPED | Wait for a process that has stopped and whose status has not yet been reported. |

# wait3 & wait4

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

- User/system CPU time, # of page faults, # of signals received, the like.



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Race Conditions

- **Definition**
  - When multiple processes are trying to do something with shared data, the final outcome depends on the order in which the processes run.


```
static void charatatime(char *);

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
    }
    exit(0);
}

static void
charatatime(char *str)
{
    char    *ptr;
    int     c;

    setbuf(stdout, NULL);           /* set unbuffered */
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}
```

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

# Who is the parent of the 2nd child?

```
int
main(void)
{
    pid_t   pid;

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

    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
        err_sys("waitpid error");

    /*
     * We're the parent (the original process); we continue executing,
     * knowing that we're not the parent of the second child.
     */
    exit(0);
}
```

```
$ ./a.out
$ second child, parent pid = 1
```

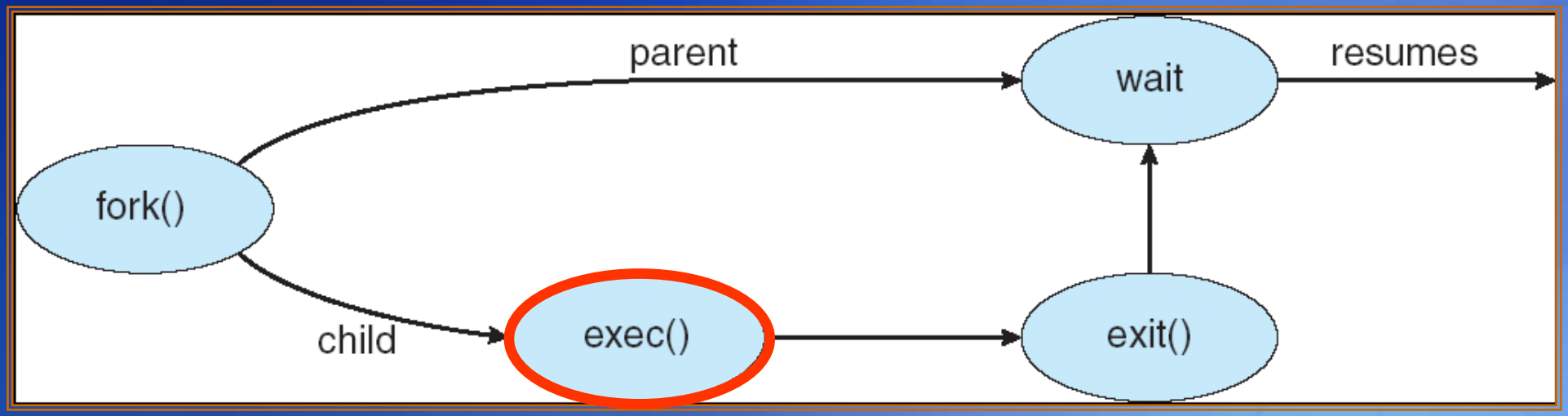
# Race Conditions

- **How to synchronize parent and child processes?**
  - Waiting loops?

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

  - Inter-Process Communication facility, such as pipe (Chapter 15.2), fifo, semaphore, shared memory, etc.

# Process Creation & Termination



# exec

- Replace the text (i.e., instructions) and data (i.e., global variables, heap, and stack) segments of a process with a program!

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, … /* (char *) 0 */);

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, … /* (char *) 0, char
    *const envp[] */);

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- *l*, *v*, and *e* stands for list, vector, and environment, respectively.

# exec

```
#include <unistd.h>
int execlp(const char *filename, const char
    *arg0, … /* (char *) 0 */);
int execvp(const char *filename, , char *const
    argv[]);
```

- With *p*, a filename is specified unless it contains '/'.
  - PATH=/bin:/usr/bin:.
  - /bin/sh is invoked with "filename" if the file is not a machine executable.
  - Example usage: login, ARG_MAX (4096)

# Differences among the six exec functions

| Function | *pathname* | *filename* | Arg list | *argv*[] | environ | *envp*[] |
|---|---|---|---|---|---|---|
| execl | • | | • | | • | |
| execlp | | • | • | | • | |
| execle | • | | • | | | • |
| execv | • | | | • | • | |
| execvp | | • | | • | • | |
| execve | • | | | • | | • |
| (letter in name) | | p | l | v | | e |

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Environment Variables

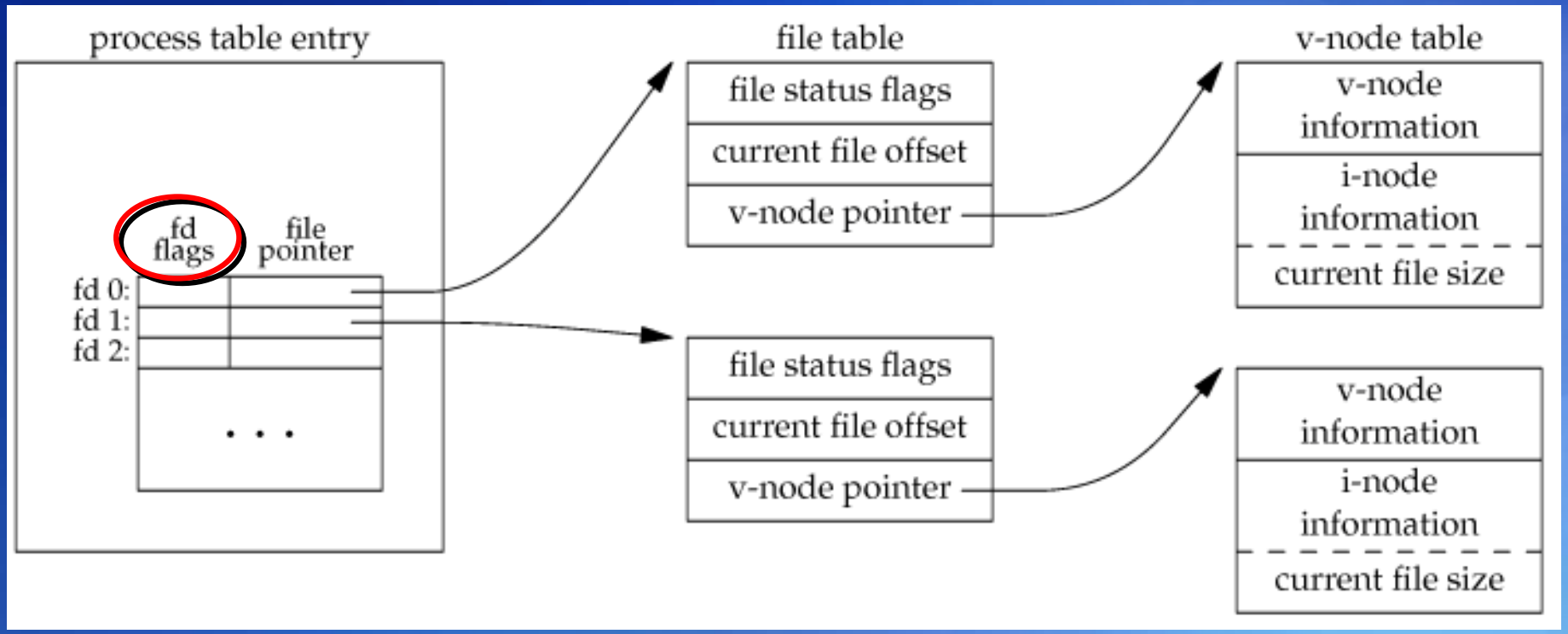
int main(int agrc, char **argv, char **envp);


Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2006
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# exec

- **Inherited from the calling process:**
  - pid, ppid, real [ug]id, supplementary gid, proc gid, session id, controlling terminal, time left until alarm clock, current working dir, root dir, file mode creation mask, file locks, proc signal mask, pending signals, resource limits, tms_[us]time, tms_cutime, tms_ustime
  - FD_CLOEXEC flag
- **Requirements & Changes**
  - Closing of open dir streams, effective user/group ID, etc.

# FD_CLOEXEC/close-on-exec Flag




Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

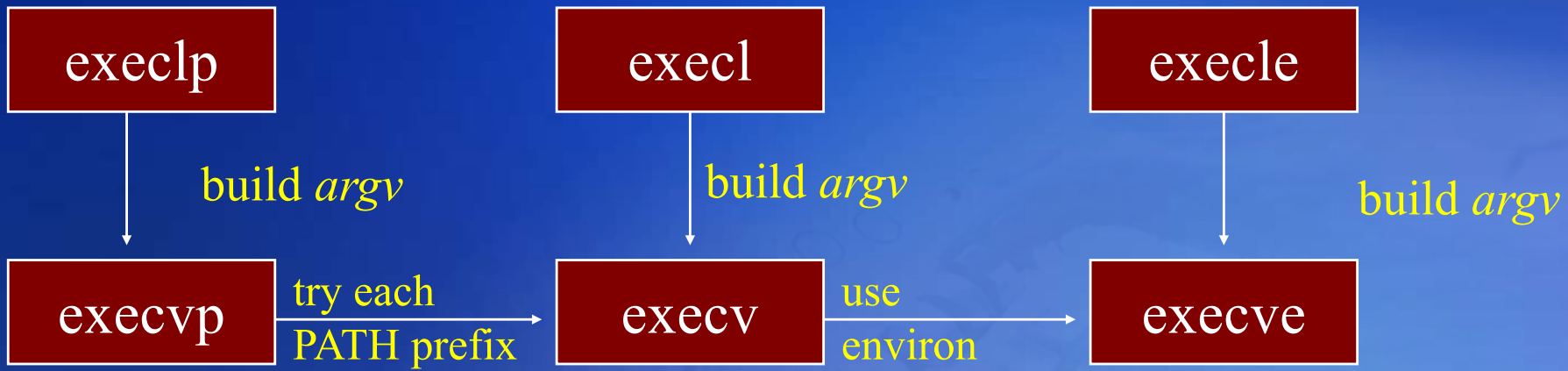
```
#include <fcntl.h>

int fcntl(int filedes, int cmd, ... /* int arg */ );
```

Returns: depends on *cmd* if OK (see following), 1 on error

| | |
|---|---|
| F_DUPFD | Duplicate the file descriptor *filedes*. The new file descriptor is returned as the value of the function. It is the lowest-numbered descriptor that is not already open, that is greater than or equal to the third argument (taken as an integer). The new descriptor shares the same file table entry as *filedes*. (Refer to Figure 3.8.) But the new descriptor has its own set of file descriptor flags, and its FD_CLOEXEC file descriptor flag is cleared. (This means that the descriptor is left open across an exec, which we discuss in Chapter 8.) |
| F_GETFD | Return the file descriptor flags for *filedes* as the value of the function. Currently, only one file descriptor flag is defined: the FD_CLOEXEC flag. |
| F_SETFD | Set the file descriptor flags for *filedes*. The new flag value is set from the third argument (taken as an integer). |

Be aware that some existing programs that deal with the file descriptor flags don't use the constant FD_CLOEXEC. Instead, the programs set the flag to either 0 (don't close-on-exec, the default) or 1 (do close-on-exec).

# exec




- **In many Unix implementations, execve is a system call.**

Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

```
char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {  /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
                "MY ARG2", (char *)0, env_init) < 0)
            err_sys("execle error");
    }

    if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

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

    exit(0);
}
```

```c
int
main(int argc, char *argv[])
{
    int         i;
    char        **ptr;
    extern char **environ;

    for (i = 0; i < argc; i++)      /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

    for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
        printf("%s\n", *ptr);

    exit(0);
}
```

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
                                    47 more lines that aren't shown
HOME=/home/sar
```

# Why keep fork() and exec() separate

- **In many client-server applications, the server may fork processes that continue to execute the same program**
  - multi-threading
- **Child ensures that the new program to be executed is in the desired state**

- Redirecting standard input, output, or error.
- Closing open files inherited from the parent that are not needed by the new program.
- Changing the UID or *process group*.
- Resetting signal handlers.

# Interprocess Communication

- Except for communicating via the file system, processes can communicate via various IPCs. (Ch.15)

| IPC type | POSIX.1 | XPG3 | V7 | SVR2 | SVR3.2 | SVR4 | 4.3BSD | 4.3+BSD |
|---|---|---|---|---|---|---|---|---|
| pipes (half duplex) | ● | ● | ● | ● | ● | ● | ● | ● |
| FIFOs | ● | ● | | ● | ● | ● | | |
| Stream pipes (full duplex) | | | | | ● | ● | ● | ● |
| named stream pipes | | | | | ● | ● | ● | ● |
| message queues | | ● | | ● | ● | ● | | |
| semaphores | | ● | | ● | ● | ● | | |
| shared memory | | ● | | ● | ● | ● | | |
| sockets | | | | | | ● | ● | ● |
| streams | | | | | ● | ● | | |

# Pipes (Ch15.2)

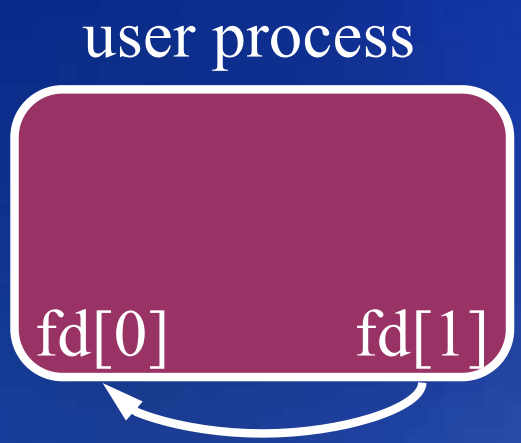
- **Pipes have two limitations:**
  - They are half-duplex.
  - They can be used between processes that have a common ancestor.
- **Stream pipes are duplex (Chap 17.2)**
- **Named pipes/FIFOs and named stream pipes (Chap 17.2) can be used between process not having the same ancestor.**
- **`pipe` function**

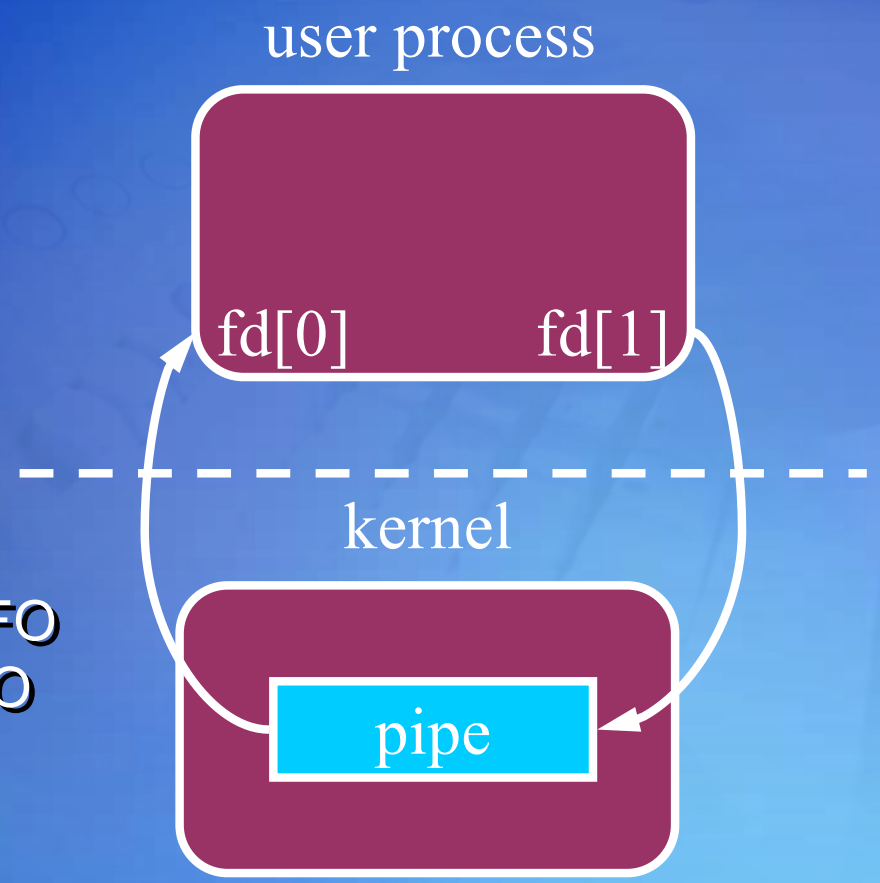
```
#include <unistd.h>
int pipe( int filedes[2] );
```



Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Unix Pipes

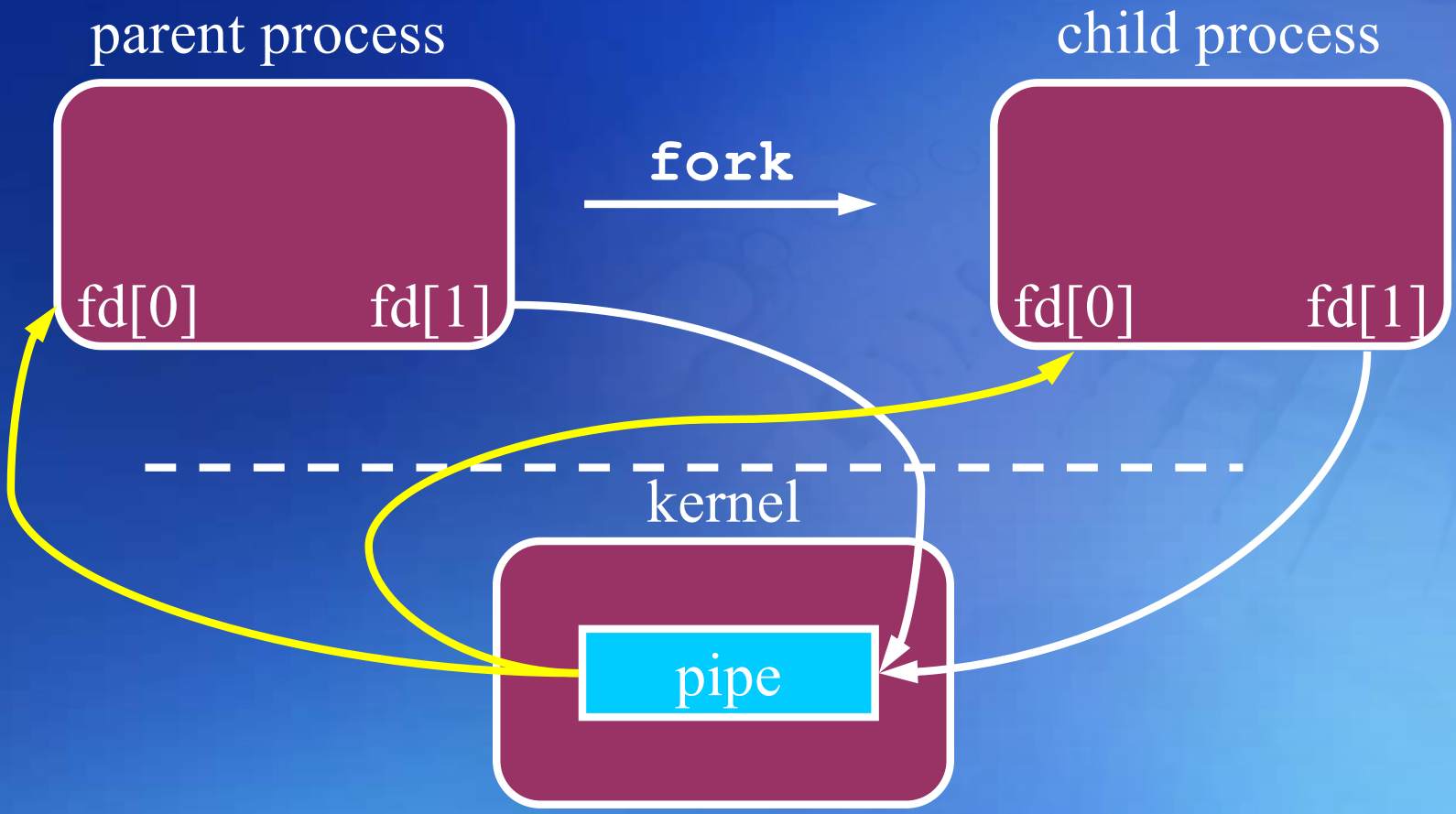



fd[0]: open for reading, file type is FIFO
fd[1]: open for writing, file type is FIFO

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

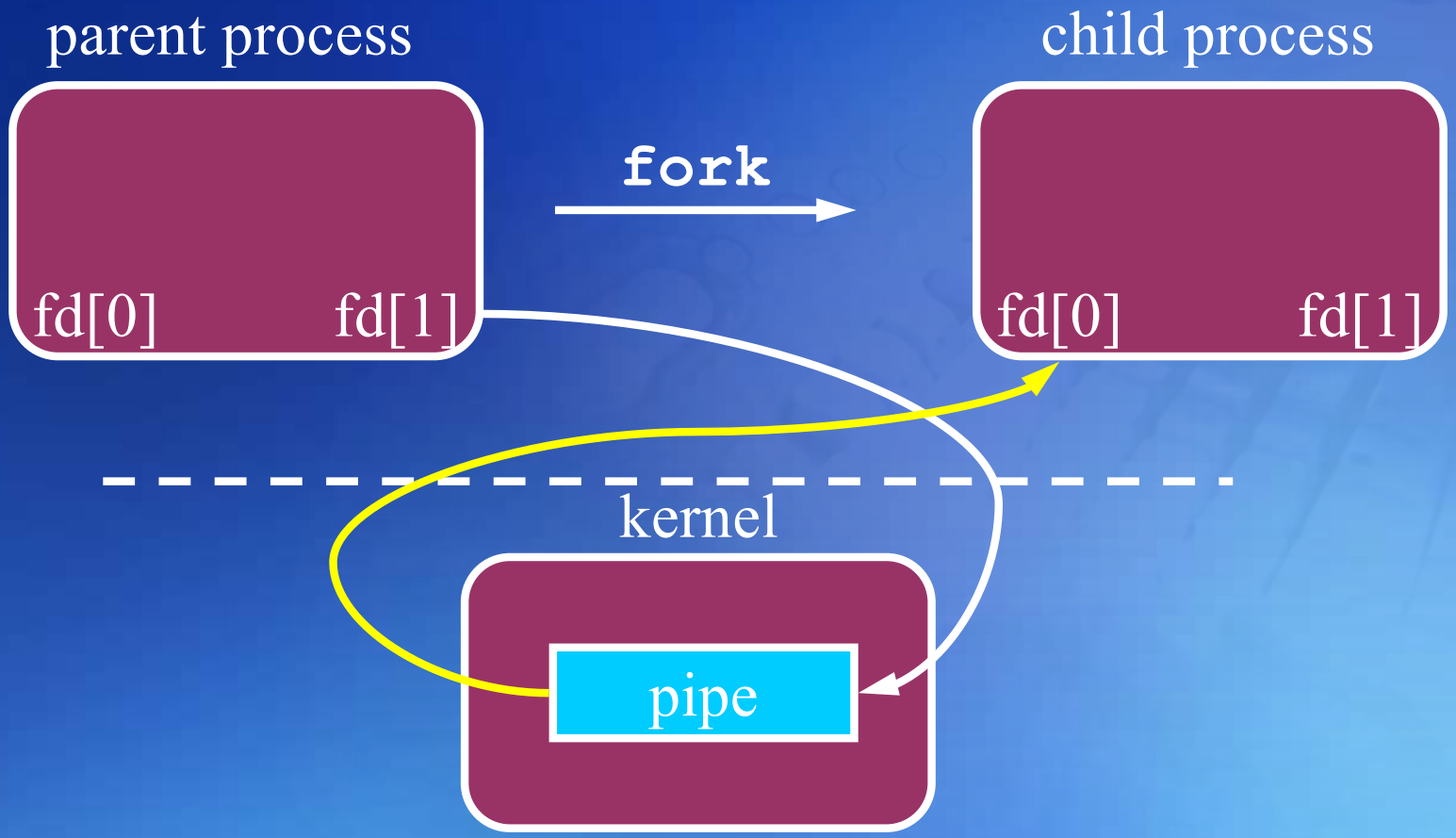
# Pipes between Processes




Half-duplex pipe after a **fork**

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Useful Pipe




Pipe from parent process to child process

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

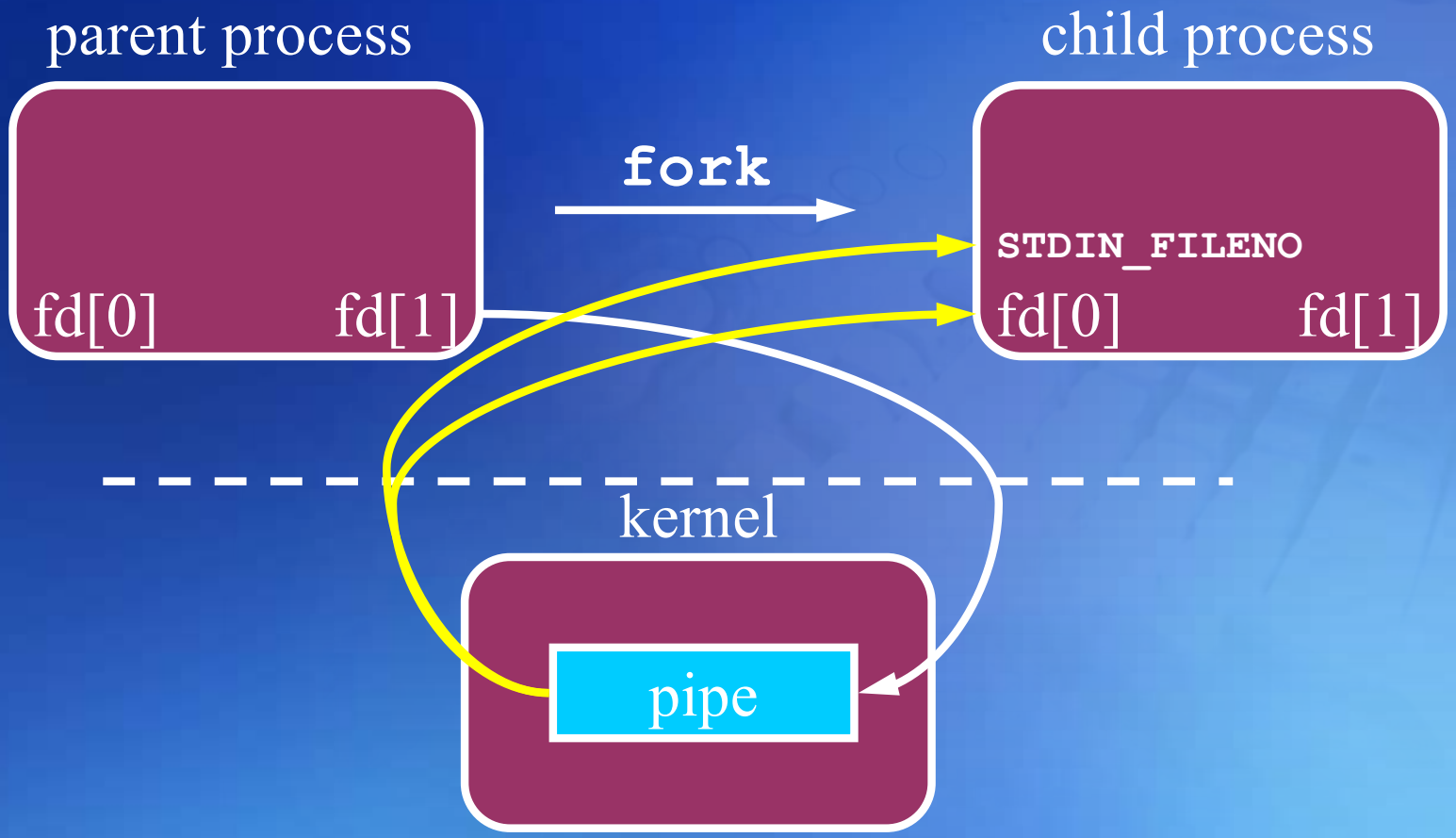
# Read from/Write to Pipes

- When the write end is closed, after all the data has been read, reading from a pipe returns 0 (end of file)
- When the read end of a pipe is closed, writing to a pipe triggers `SIGPIPE` signal.
- Kernel's pipe buffer size:
  - Constant PIPE_BUF specifies the pipe buffer size.
  - Writing PIPE_BUF or less will not be interleaved.

# Creating a pipe from the parent to the child and send data down the pipe

```
int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {        /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {                     /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

# Connecting Pipes to Standard I/O



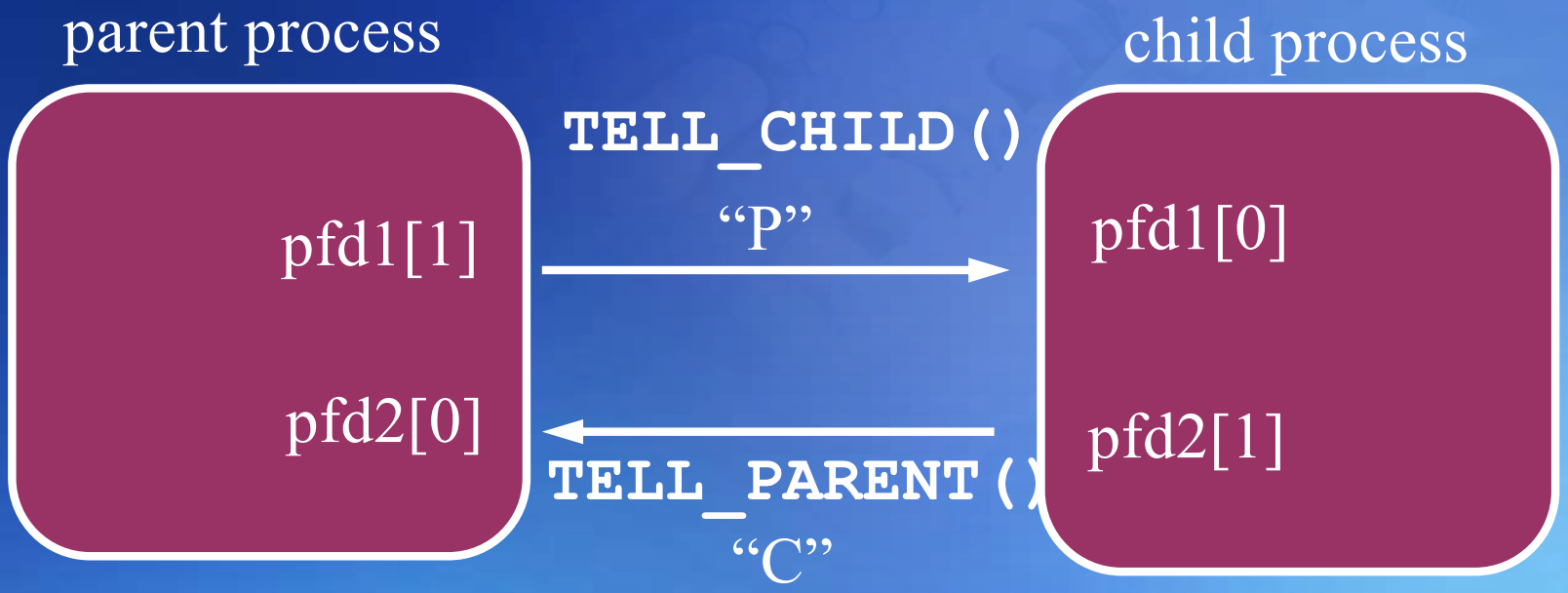

Pipe from parent process to child process

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

```c
#define DEF_PAGER   "/bin/more"     /* default pager program */

int
main(int argc, char *argv[])
{
    int    n;
    int    fd[2];
    pid_t  pid;
    char   *pager, *argv0;
    char   line[MAXLINE];
    FILE   *fp;

    if (argc != 2)
        err_quit("usage: a.out <pathname>");

    if ((fp = fopen(argv[1], "r")) == NULL)
        err_sys("can't open %s", argv[1]);
    if (pipe(fd) < 0)
        err_sys("pipe error");

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {                                   /* parent */
        close(fd[0]);       /* close read end */

        /* parent copies argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
            n = strlen(line);
            if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
            err_sys("fgets error");

        close(fd[1]);   /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");
        exit(0);
    } else {                                                /* child */
```

```
    } else {                                                /* child */
        close(fd[1]);   /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                err_sys("dup2 error to stdin");
            close(fd[0]);   /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0)
            err_sys("execl error for %s", pager);
    }
    exit(0);
}
```

# Using Pipes for Parent-child Synchronization




Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# Avoid Race Condition (Chap 8)

```
    static void charatatime(char *);

    int
    main(void)
    {
        pid_t   pid;

+       TELL_WAIT();
+
        if ((pid = fork()) < 0) {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();      /* parent goes first */
            charatatime("output from child\n");
        } else {
            charatatime("output from parent\n");
+           TELL_CHILD(pid);
        }
        exit(0);
    }
    static void
    charatatime(char *str)
    {
        char    *ptr;
        int     c;

        setbuf(stdout, NULL);           /* set unbuffered */
        for (ptr = str; (c = *ptr++) != 0; )
            putc(c, stdout);
    }
```

```
static int  pfd1[2], pfd2[2];

void
TELL_WAIT(void)
{
    if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
        err_sys("pipe error");
}

void
TELL_PARENT(pid_t pid)
{
    if (write(pfd2[1], "c", 1) != 1)
        err_sys("write error");
}

void
WAIT_PARENT(void)
{
    char    c;

    if (read(pfd1[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'p')
        err_quit("WAIT_PARENT: incorrect data");
}
```

```
void
TELL_CHILD(pid_t pid)
{
    if (write(pfd1[1], "p", 1) != 1)
        err_sys("write error");
}

void
WAIT_CHILD(void)
{
    char    c;

    if (read(pfd2[0], &c, 1) != 1)
        err_sys("read error");

    if (c != 'c')
        err_quit("WAIT_CHILD: incorrect data");
}
```

# FIFOs (Ch15.5)

- **FIFOs – named pipes for (un-)related processes to exchange data.**
  - A file type – st_mode (S_ISFIFO macro)
  - Data in a FIFO removed when the last referencing process terminates.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char *pathname,
           mode_t mode);
```

  - mode and usr/grp ownership = those of a file
  - Op's: open, close, read, write, unlink, etc.

# FIFOs

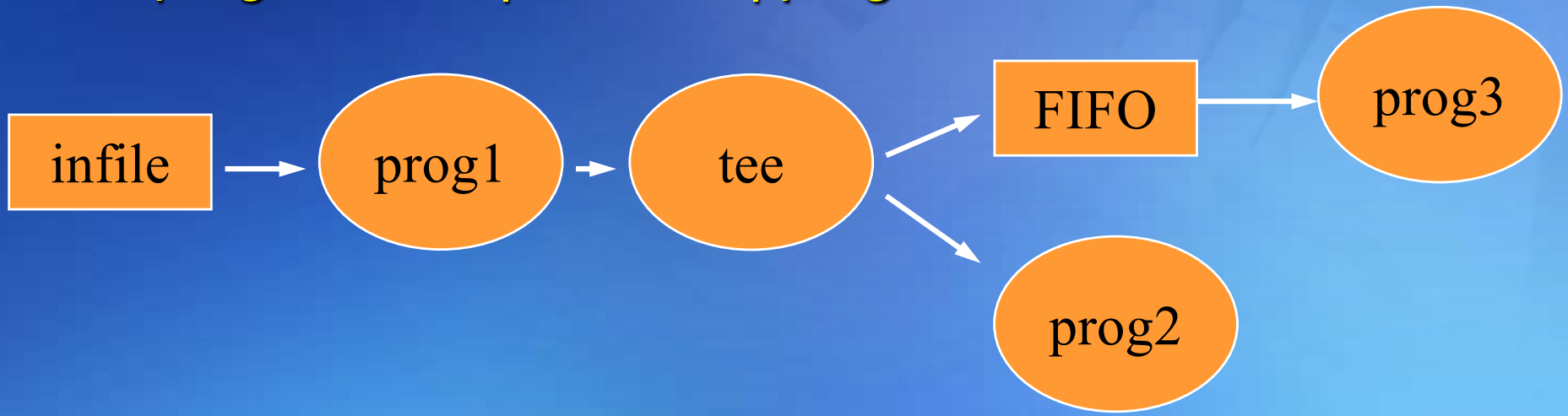
- **O_NONBLOCK**
  - NO → an open for read-only blocks until some other process opens the FIFO for writing (write-only as well).
  - Yes → an open for read-only always returns, while that for write-only returns with an error (errno=ENXIO) if there is no reader.
- **Write for a no-reader FIFO → SIGPIPE**
  - Closing of the last writer → EOF
- **Atomic writes**
  - PIPE_BUF

# FIFOs

**tee:** copies stdin to stdout, making a copy in files

- **How to pipe the standard output of a process to more than one file/process?**
- **Example – Stream Duplication**

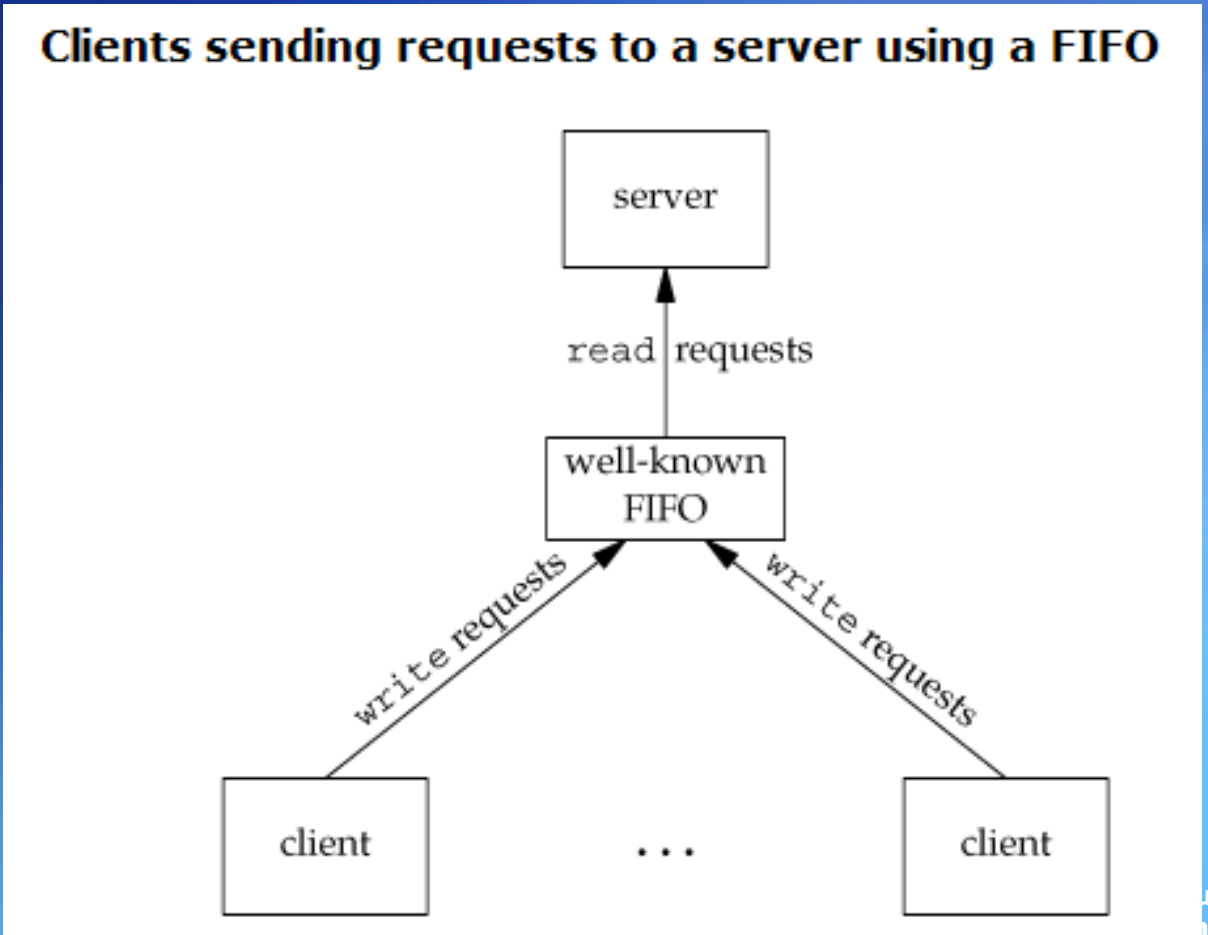
```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```



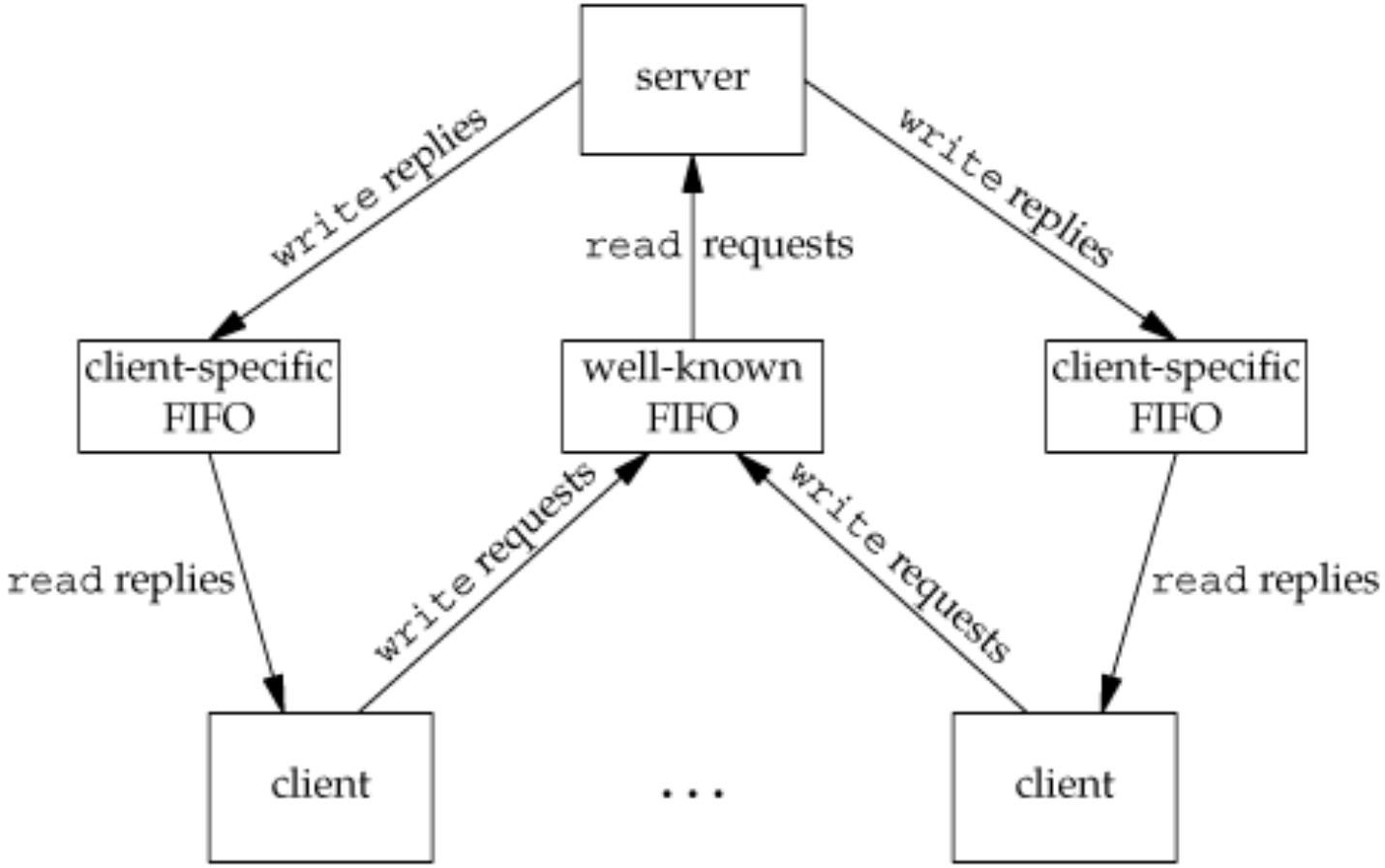

Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# FIFOs

- **Example – Client-Server Communication**
  - One server & multiple clients
    - Issues: replies, SIGPIPE, removing of all clients



## Clientserver communication using FIFOs




Tei-Wei Kuo, Chi-Sheng Shih and Hao-Hua Chu©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# What You Should Know

- **The Concept of Process**
  - Process Control Block, Process States, Context Switch
- **Process Control Primitives**
  - fork(), exec(), exit(), and wait()
- **Performance of fork() (vs. vfork())**
- **Inherited/Different properties affected by fork() and exec()**
- **Zombie vs Orphan**
- **Race Conditions**



Tei-Wei Kuo, Chi-Sheng Shih, Hao-Hua Chu, and Pu-Jen Cheng©2007
Department of Computer Science and Information Engineering
Graduate institute of Multimedia and Networking, National Taiwan University

# What You Should Know

- **Least-privilege Model**
  - saved set-user/group-id, security problem (+fork())
- **Interpreter Files**
  - exec(), advantage/disadvantage
- **Process Accounting and Times**
- **Pipe and FIFO**
  - limitations
  - I/O: read (EOF), write (SIGPIPE), blocking/nonblocking
  - intermixing problem