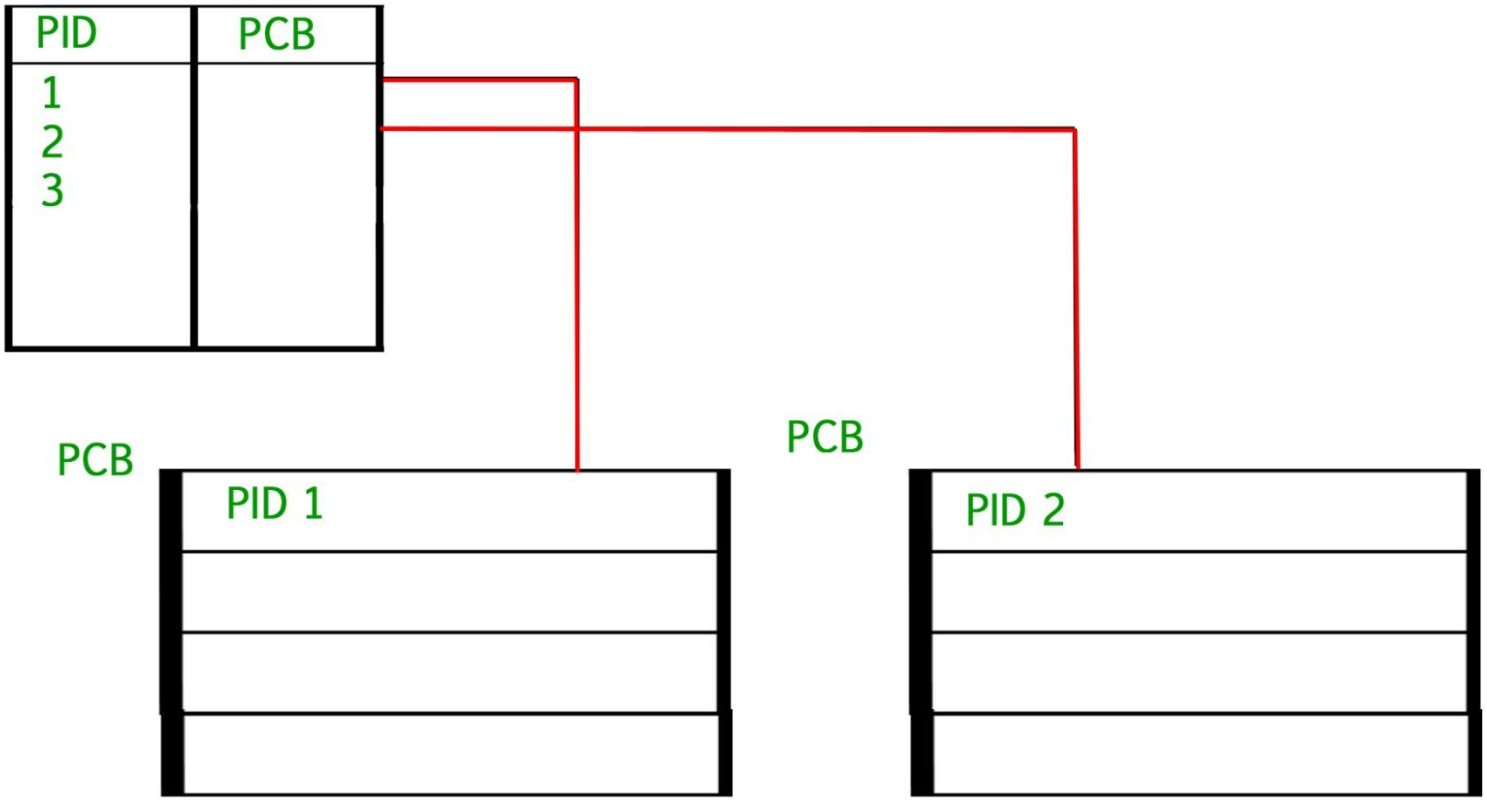
Interprocess Communication

Objectives

- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication
- To describe communication in client-server systems

Process Control Block



Process table and process control block

Process Control Block

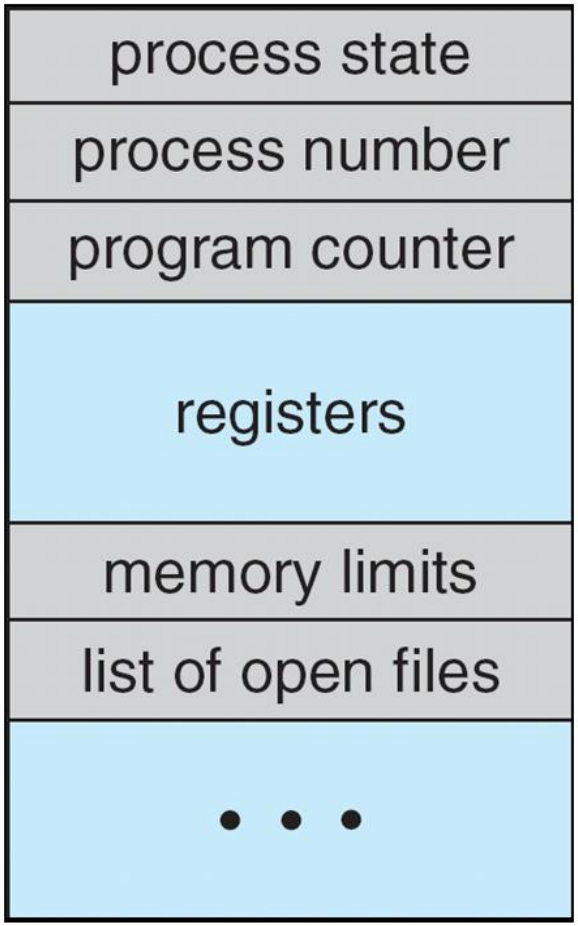
Process Control Block

- **Pointer** – It is a stack pointer which is required to be saved when the process is switched from one state to another to retain the current position of the process.
- **Process state** – It stores the respective state of the process.
- **Process number** – Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
- **Program counter** – It stores the counter which contains the address of the next instruction that is to be executed for the process.
- **Register** – These are the CPU registers which includes: accumulator, base, registers and general purpose registers.
- **Memory limits** – This field contains the information about memory management system used by operating system. This may include the page tables, segment tables etc.
- **Open files list** – This information includes the list of files opened for a process.

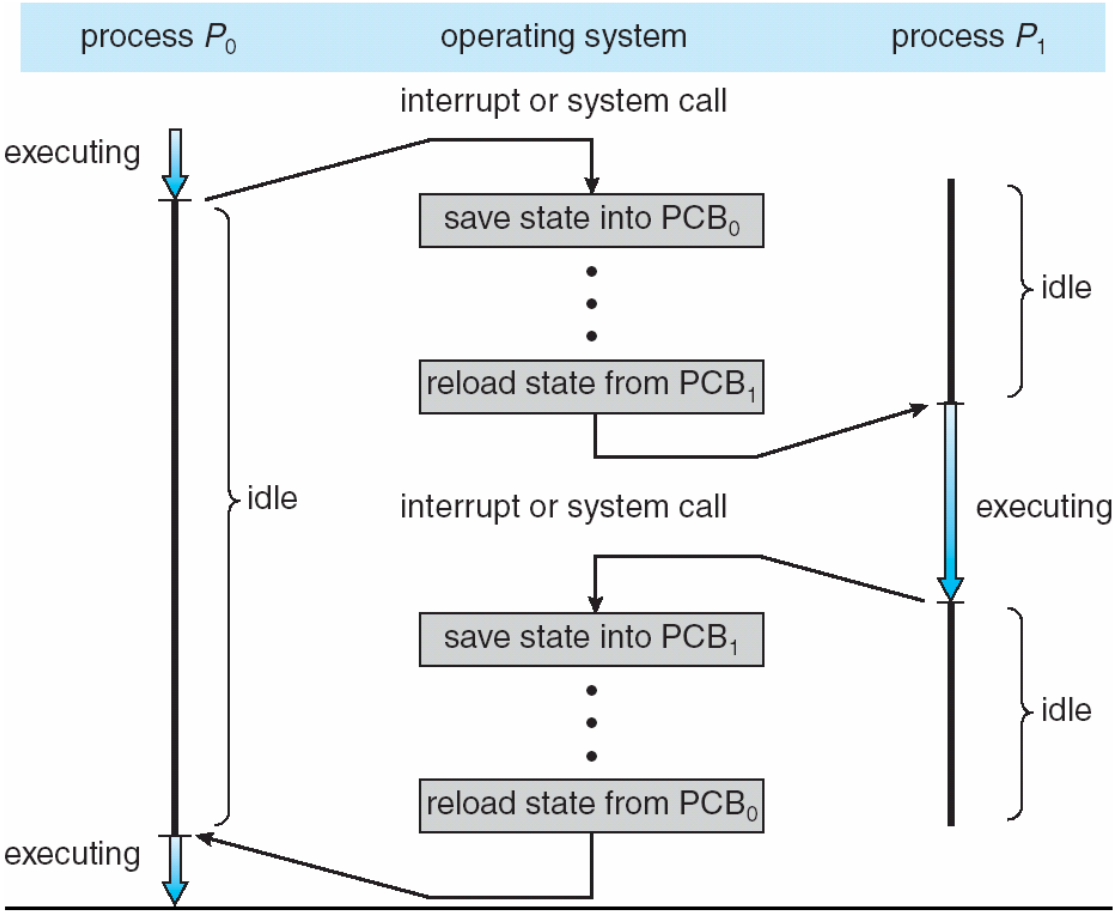
Process Control Block (PCB)

Information associated with each process

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



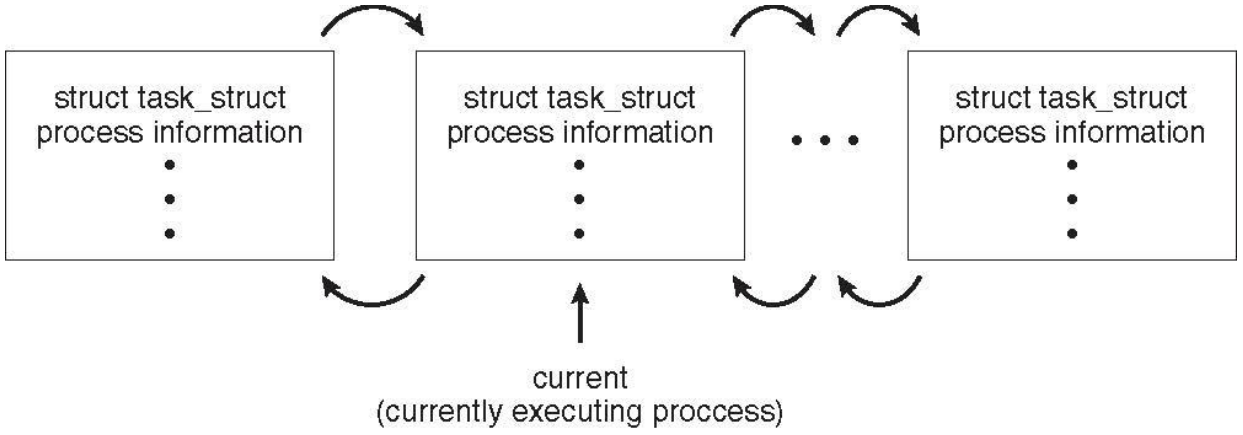
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Process Representation in Linux

Represented by the C structure `task_struct`

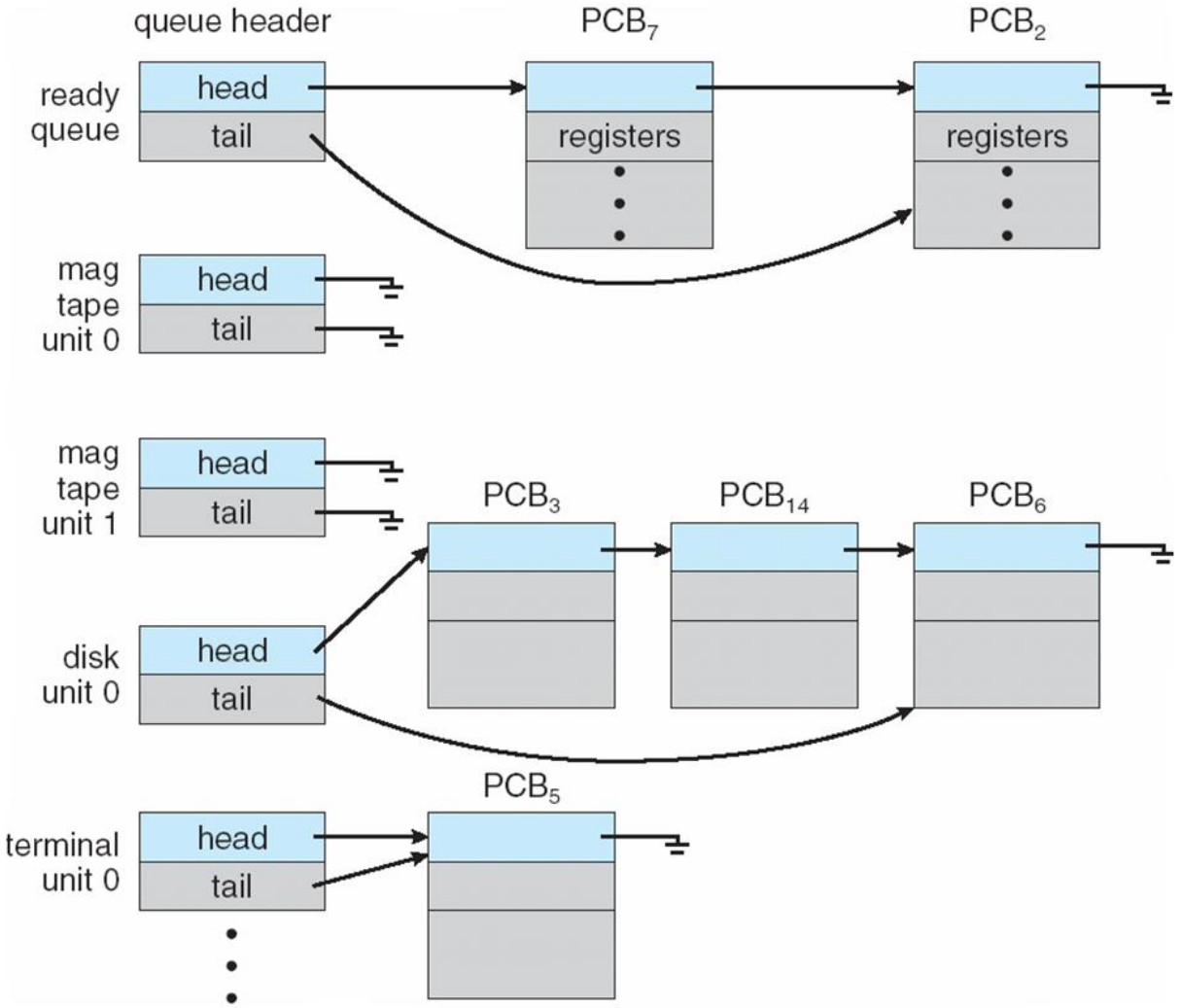
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

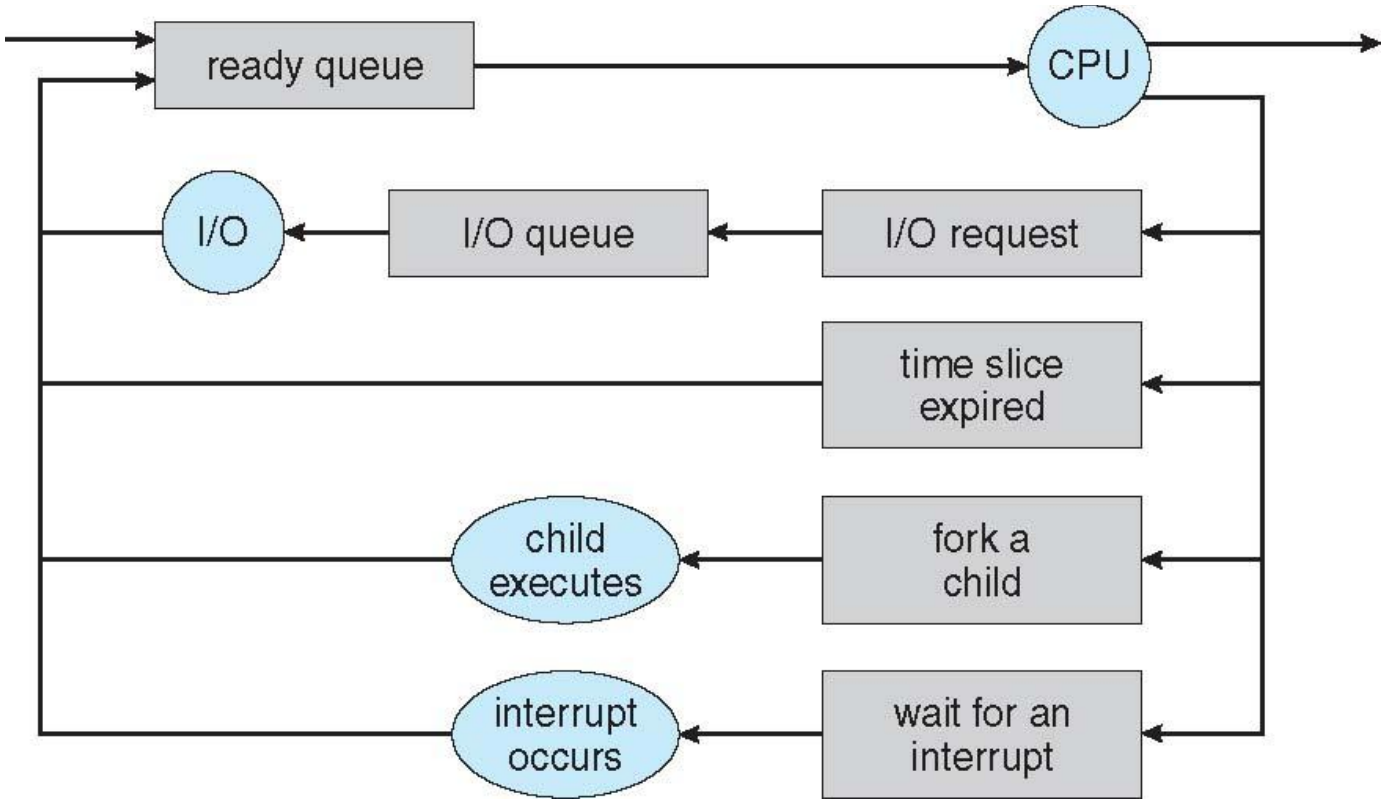
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing 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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



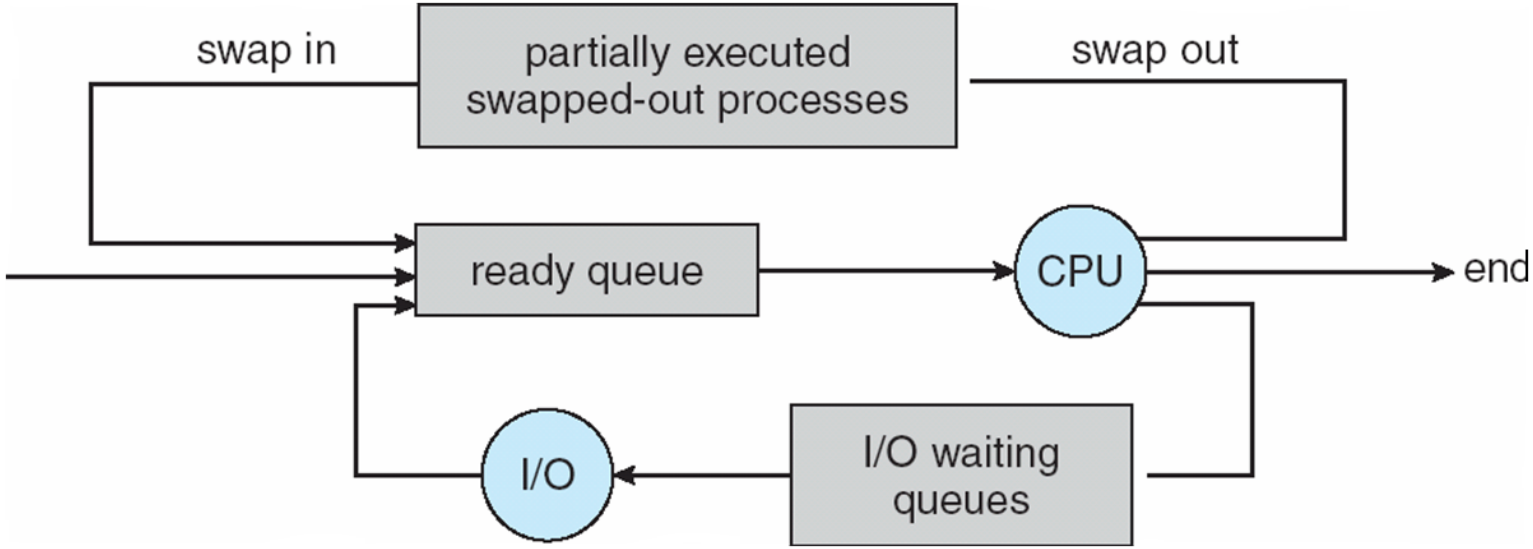
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

Medium-term scheduler can be added if degree of multiple programming needs to decrease

Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Process Swapping

Swap-in: Transfer from secondary storage back into RAM.

Swap-out: from RAM to secondary storage.

Advantages:

- Improved memory utilization
- Virtual memory creation
- Priority based scheduling

Disadvantages:

- Performance
- Data loss risk
- Increased page faults

Multitasking in Mobile Systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run, others suspended
- Due to screen real estate, and user interface limits iOS provides
  - Single **foreground** process- controlled via user interface
  - Multiple **background** processes– in memory, running, but not on the display, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback

Multitasking in Mobile Systems (cont.)

- Android runs foreground and background, with fewer limits
  - Background process uses a **service** to perform tasks
  - Service can keep running even if background process is suspended
  - Service has no user interface, small memory use

Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so, on as detailed next

Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

C Program Forking Separate Process

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.

Process Termination (cont.)

- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait`, process is an **orphan**

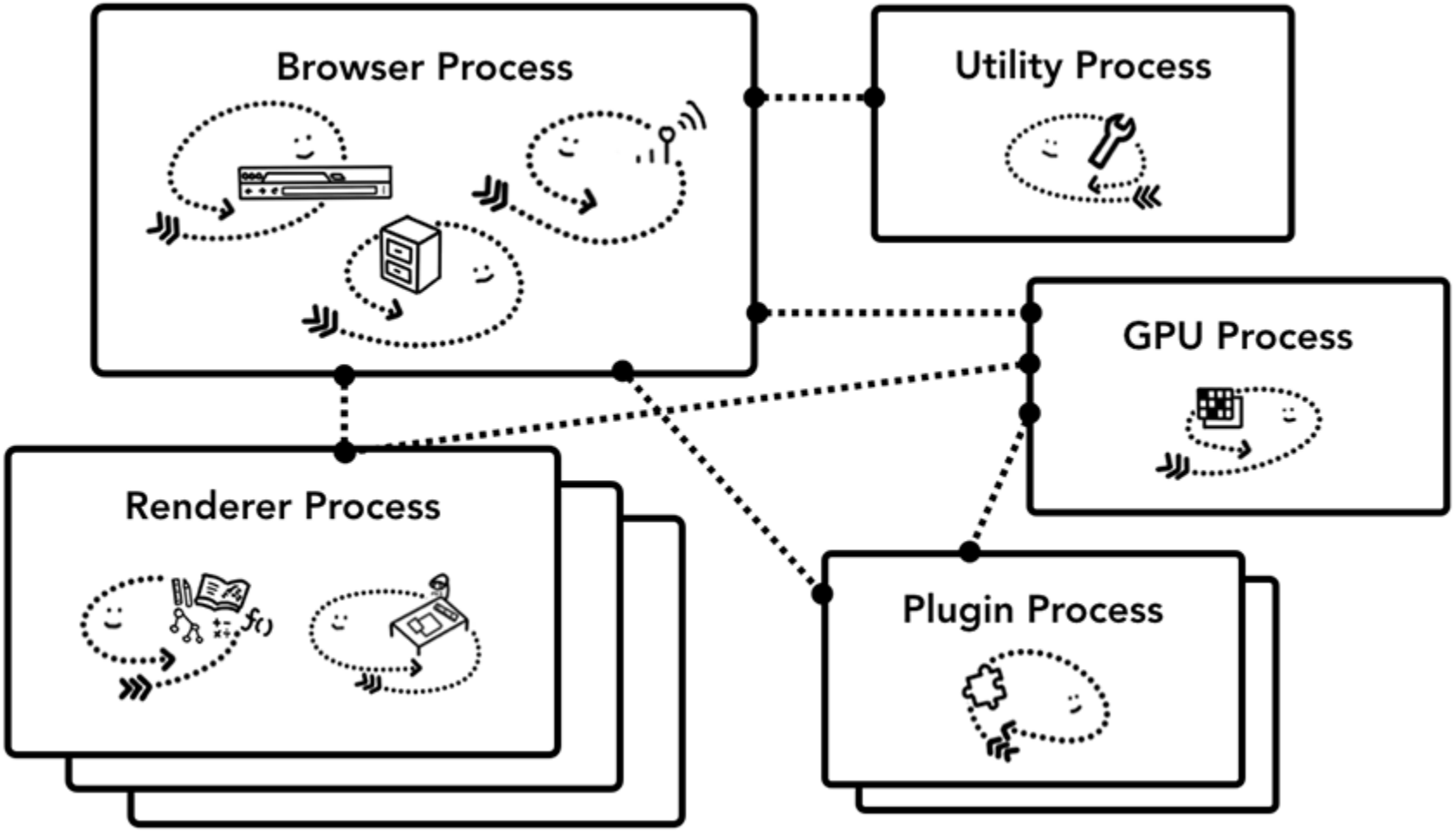
Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess.

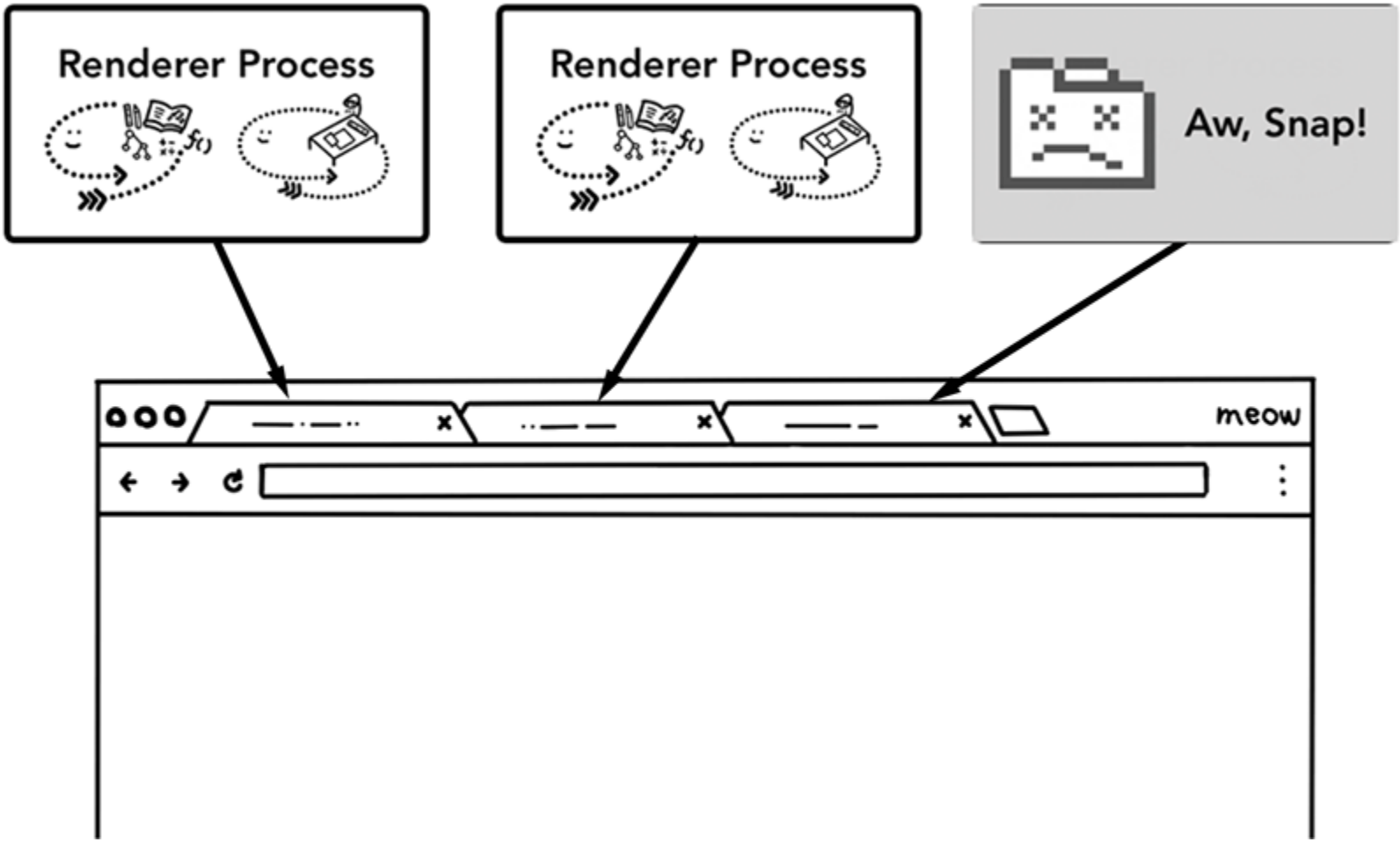
Multiprocess Architecture – Chrome Browser (cont.)

- Google Chrome Browser has 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

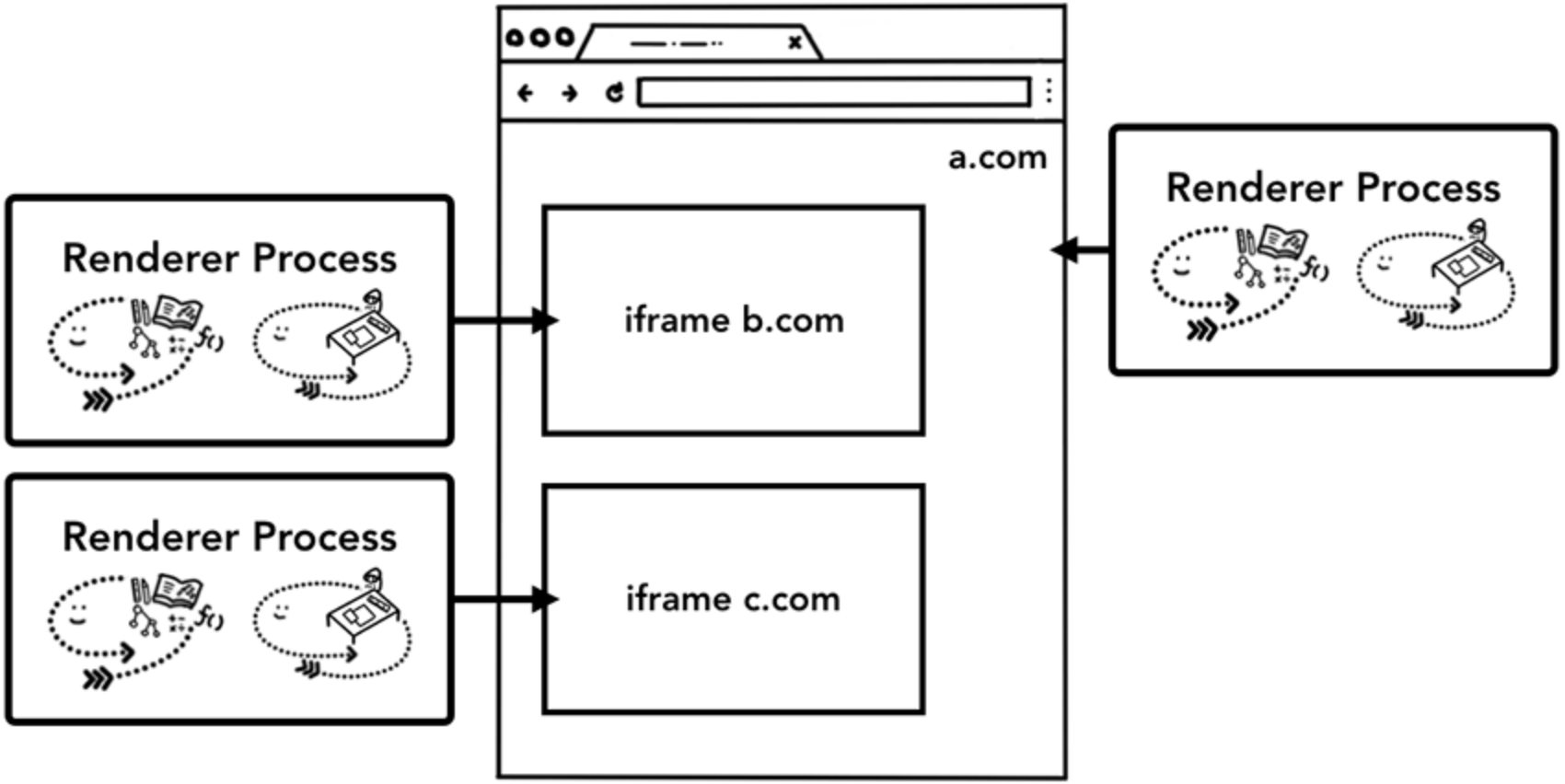
Multiprocess Architecture – Chrome Browser (cont.)



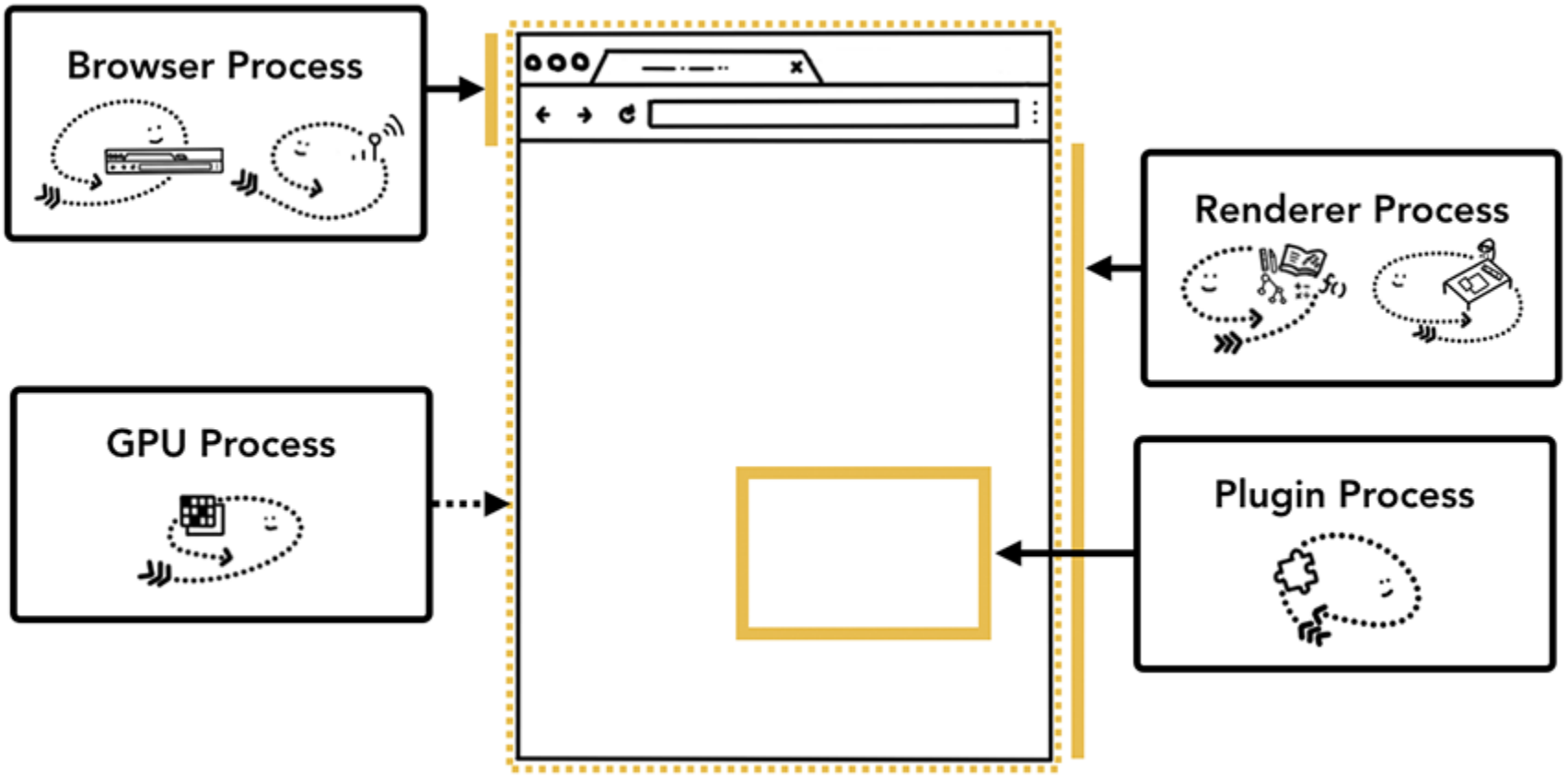
Multiprocess Architecture – Chrome Browser (cont.)



Multiprocess Architecture – Chrome Browser (cont.)
Multiple renderers



Multiprocess Architecture – Chrome Browser (cont.)
Different processes pointing to different parts of the bowser UI

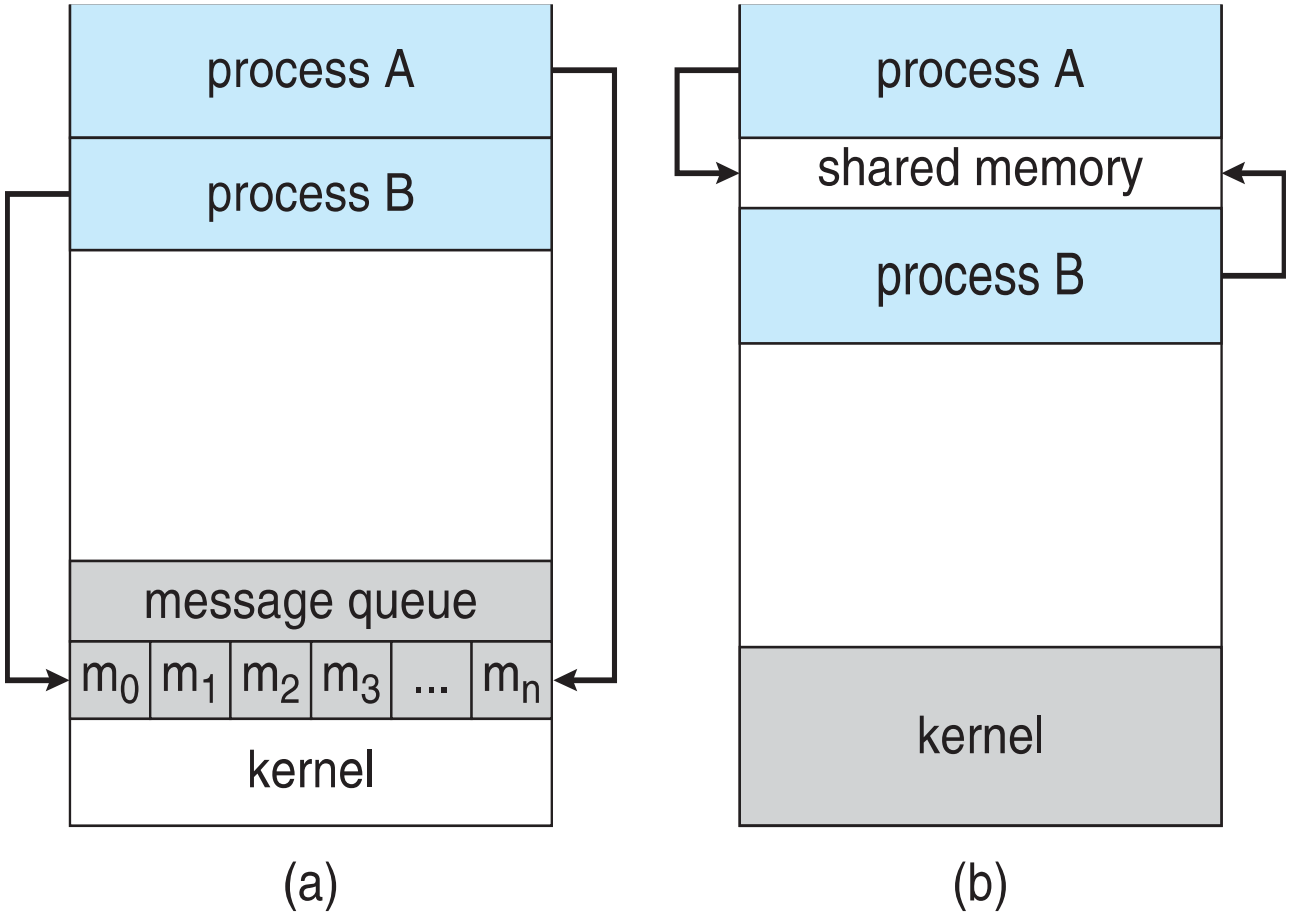


Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity

fork() and pipe()

- What if we want to support the following:
  - ls –la | sort
- One process should execute ls –la and another should execute sort
- By default, a command such as ls requires that its output goes to standard output. This means the terminal (stdout)
- The sort command requires a file be provided as a command line argument from standard input (stdin)

Possible Solution

```
int pid;
  pid = fork();
  if (pid<0) {
     perror("Problem forking");
     exit(1);
  } else if (pid>0) {
     execlp("ls","ls","-la", NULL);
     perror("exec problem");
     exit(1);
  } else {
     execlp("sort","sort",NULL);
     perror("exec problem");
     exit(1);
   }
   return(0);
}
```

Example (cont.)

- Why does it not work?
- The output of the ls –la goes to the terminal
- We wanted it to be the input to the sort

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?

Pipes (cont.)

- Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

Ordinary Pipes allow communication in standard producer-consumer style

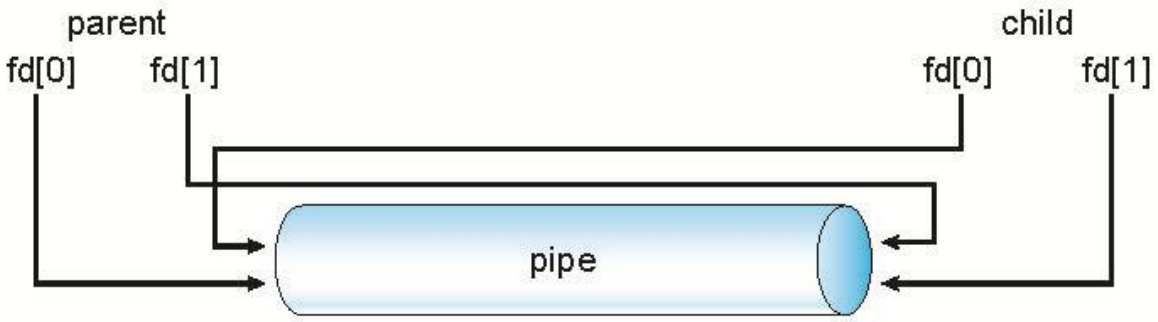
Producer writes to one end (the **write-end** of the pipe)

Consumer reads from the other end (the **read-end** of the pipe)

Ordinary Pipes(cont.)

Ordinary pipes are therefore unidirectional

Require parent-child relationship between communicating processes



Windows calls these **anonymous pipes**

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)

Interprocess Communication

- Pipes
  - Used to allow processes on the same machine to communicate with each other
  - One process spawns the other
- Sockets
  - Processes can be on different machine

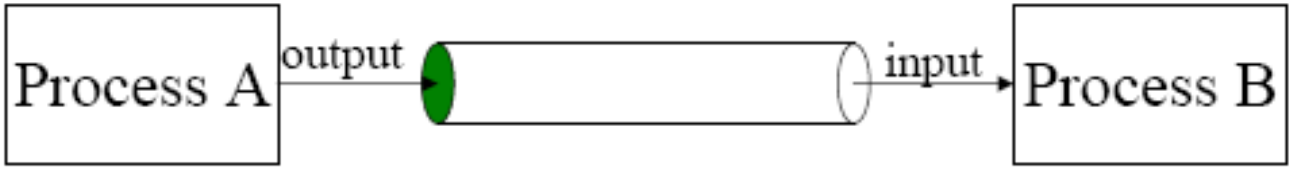
IPC Mechanisms

- Pipes
  - Processes on the same machine
  - Allows parent process to communicate with child process
  - Allows two "sibling" processes to communicate
  - Used mostly for a pipeline of filters
- Sockets
  - Processes on any machines
  - Processes created independently
  - Used for client/server communication (e.g., Web)

Both provide abstraction of an "ordered stream of bytes"

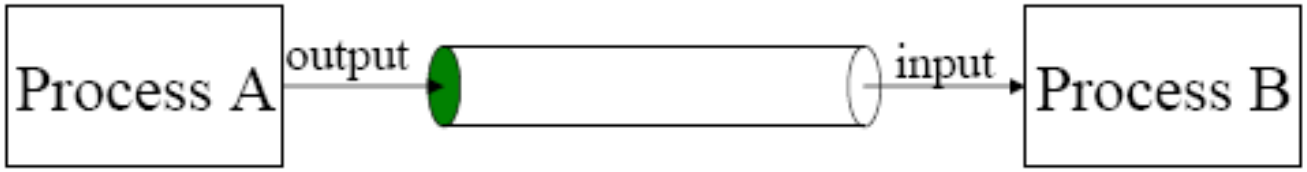
Pipes

- Provides an interprocess communication channel



- A filter is a process that reads from `stdin` and writes to `stdout`

Creating a Pipe



- Pipe is a communication channel abstraction
  - Process A can write to one end using "write" system call
  - Process B can read from the other end using "read" system call
- System call

```
int pipe( int fd[2] );
return 0 upon success –1 upon failure
fd[0] is open for reading
fd[1] is open for writing
```

- Two coordinated processes created by **`fork`** can pass data to each other using a pipe.

Redirection

- Unix allows redirection of stdin, stdout, or stderr
- How?
  - Use **`open()`**, **`creat()`**, and **`close()`** system calls
    - Described in **I/O Management** lecture
  - Use **`dup()`** system call…

```
int dup(int oldfd);
```

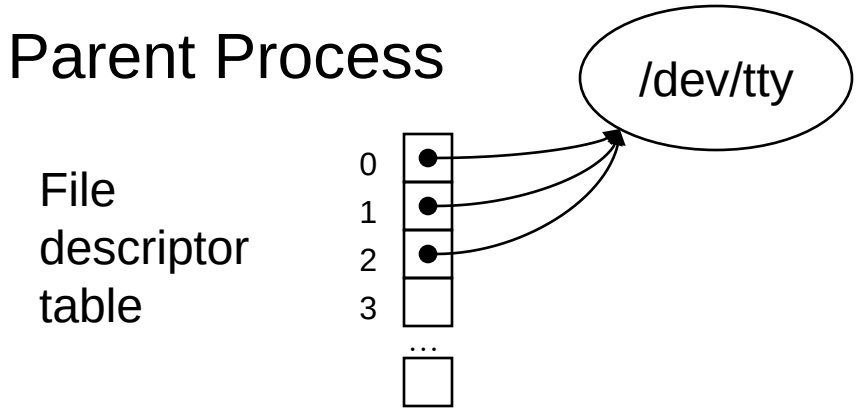
- Create a copy of the file descriptor oldfd. After a successful return from dup() or dup2(), the old and new file descriptors may be used interchangeably. They refer to the same open file description and thus share file offset and file status flags. Uses the lowest-numbered unused descriptor for the new descriptor. Return the new descriptor, or -1 if an error occurred.

Redirection Example

How does shell implement “somepgm > somefile”?

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

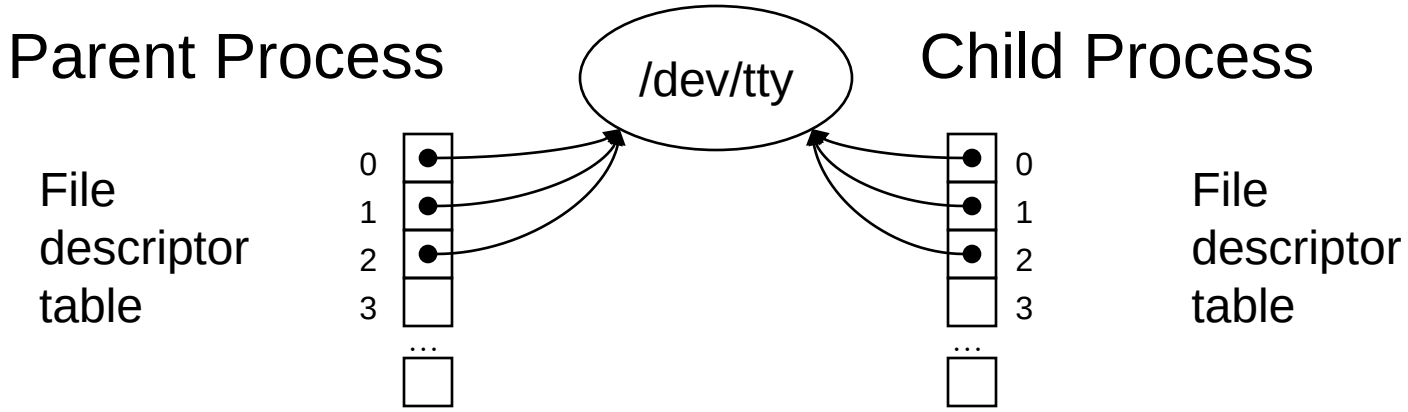
Redirection Example Trace (1)




```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent has file descriptor table; first three point to “terminal”

Redirection Example Trace (2)

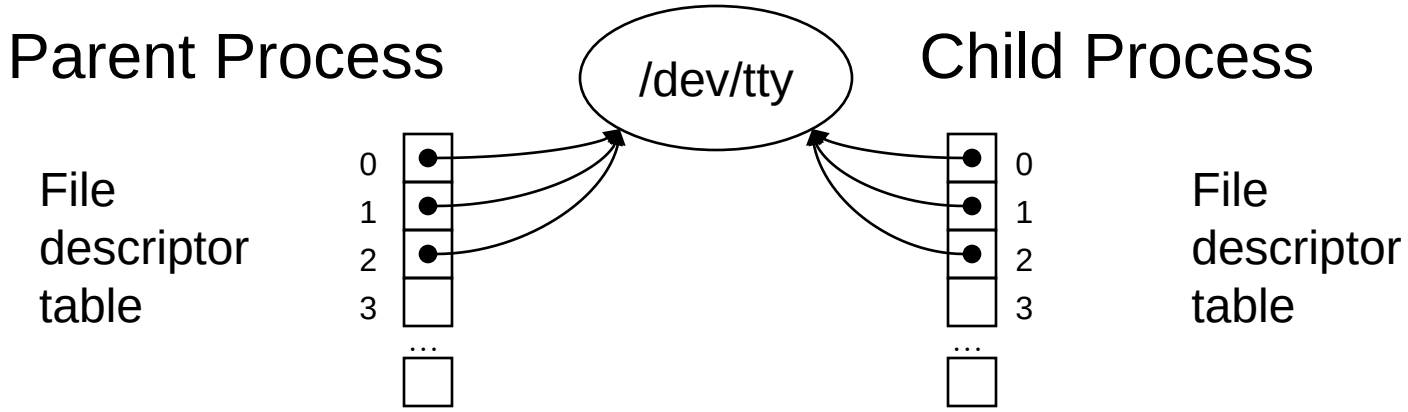



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Parent forks child; child has identical file descriptor table

Redirection Example Trace (3)

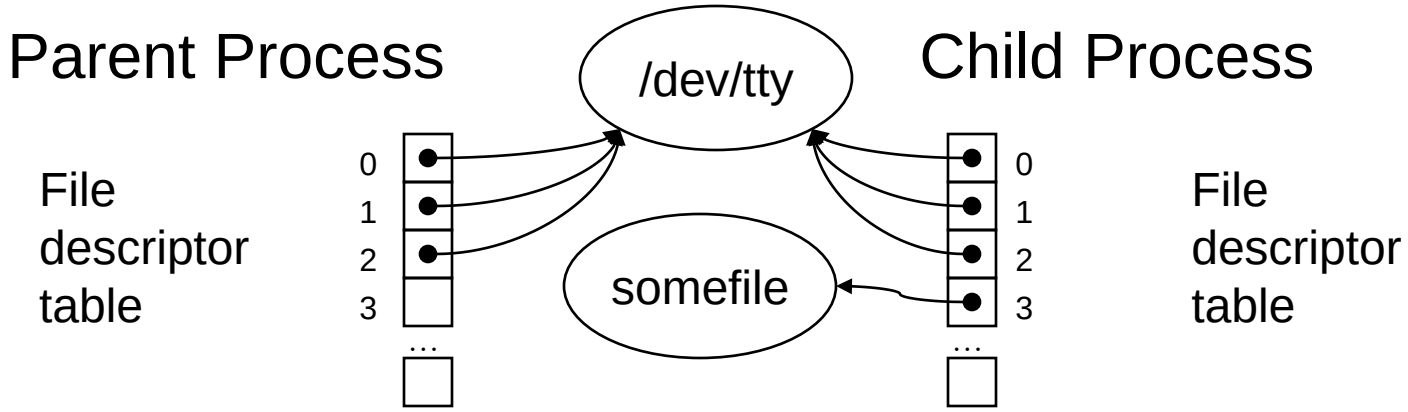



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Let's say parent gets CPU first; parent waits

Redirection Example Trace (4)



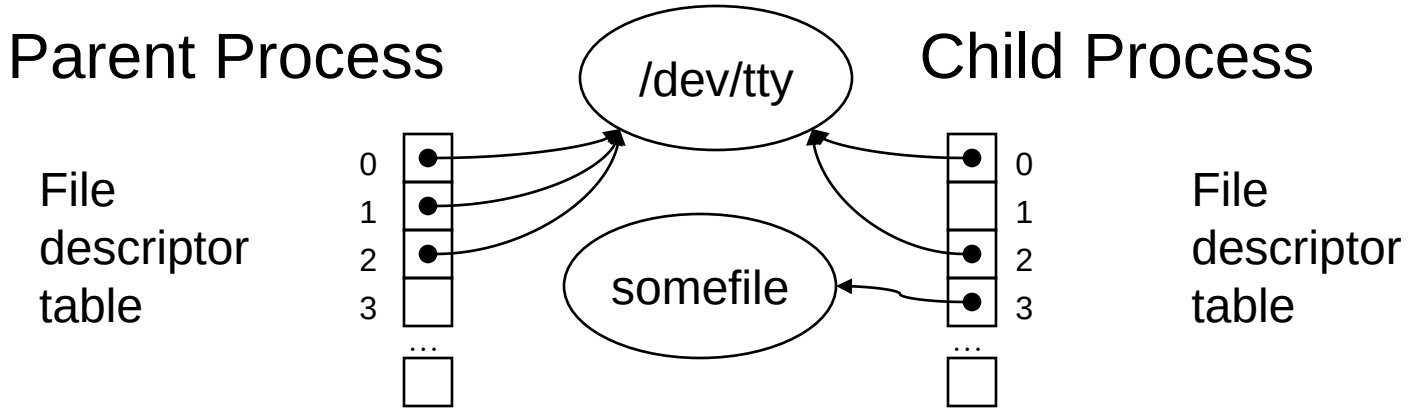

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child gets CPU; child creates somefile

Redirection Example Trace (5)



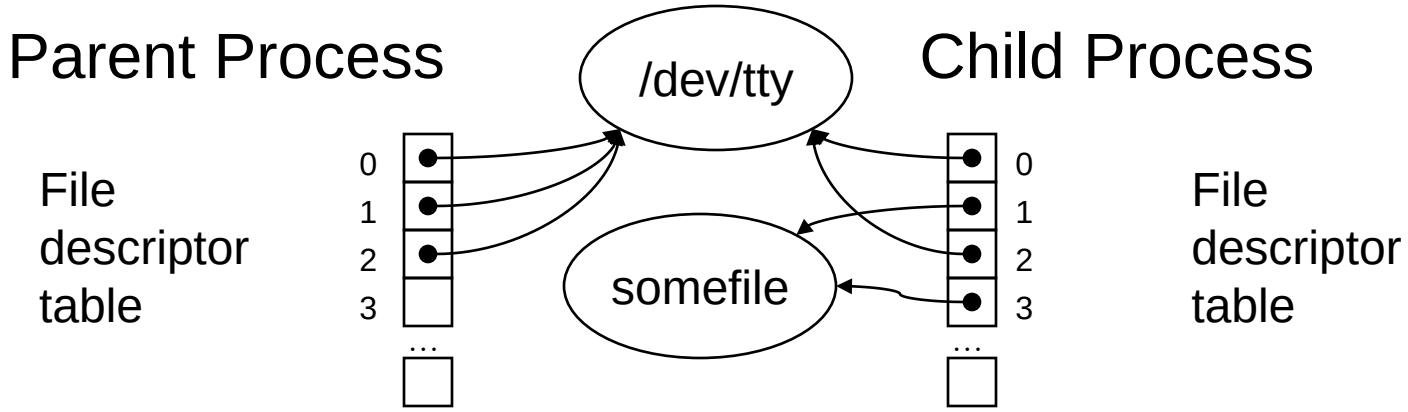

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

1

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 1 (stdout)

Redirection Example Trace (6)



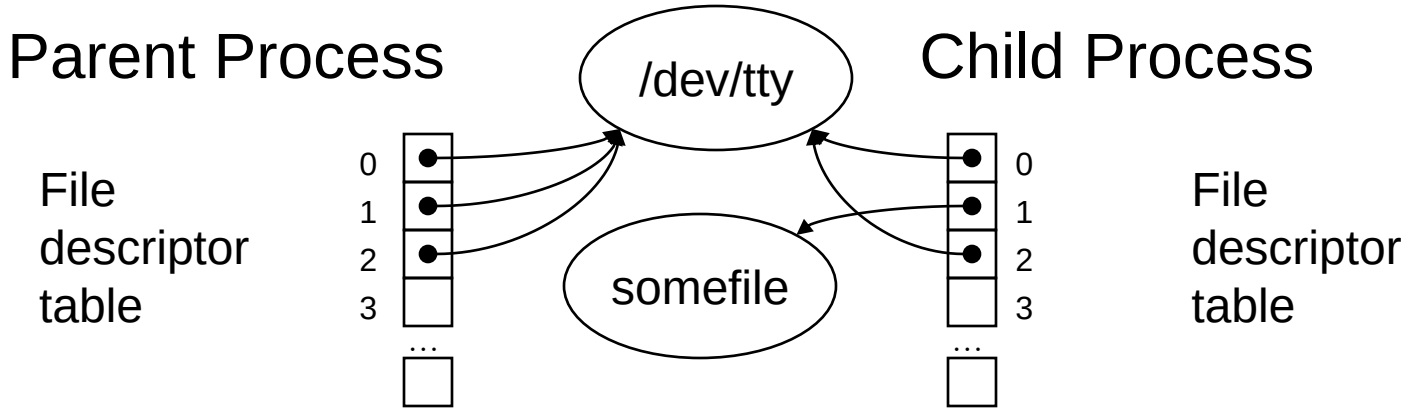

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3, 1

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child duplicates file descriptor 3 into first unused spot

Redirection Example Trace (7)



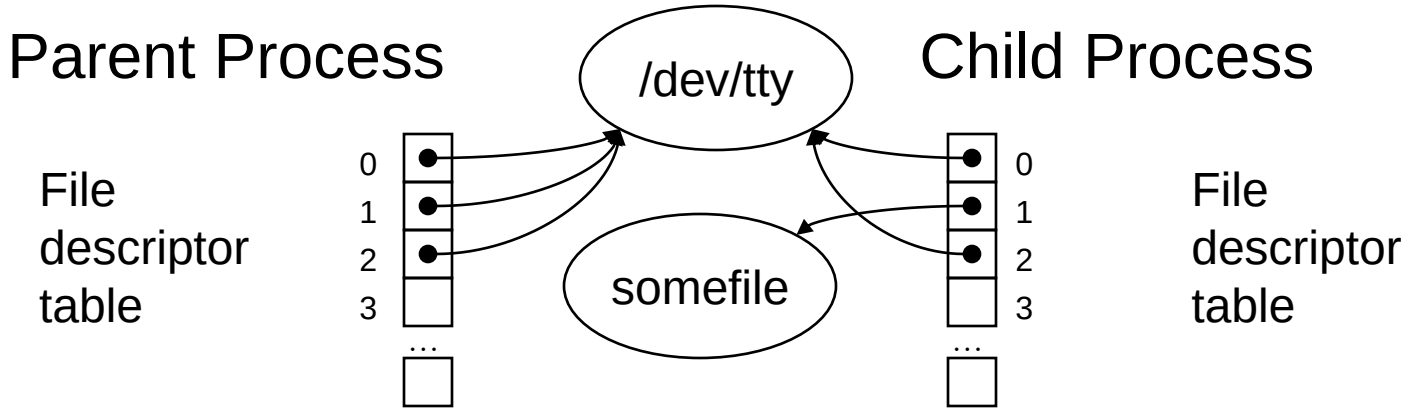

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

3

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child closes file descriptor 3

Redirection Example Trace (8)

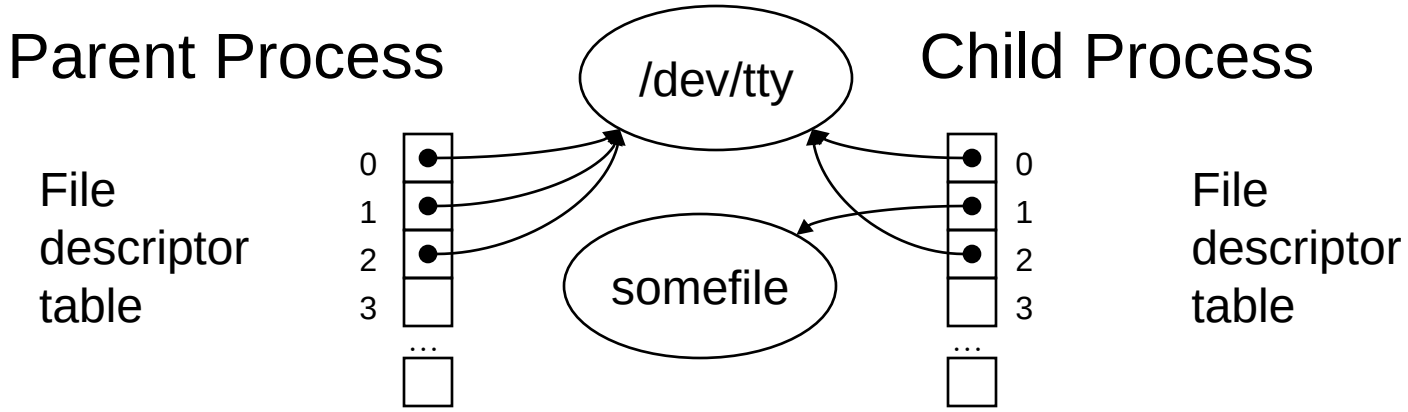



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

Child calls execvp()

Redirection Example Trace (9)

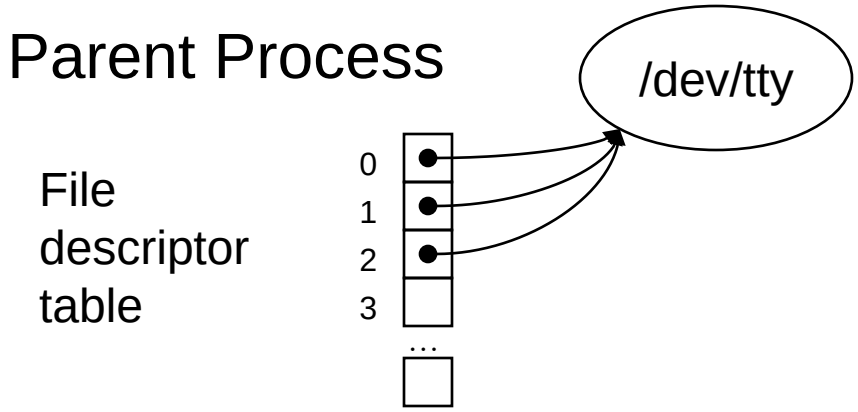



```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somepfm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

somepgm

Somepgm executes with stdout redirected to somefile

Redirection Example Trace (10)




```
pid = fork();
if (pid == 0) {
   /* in child */
   fd = creat("somefile", 0640);
   close(1);
   dup(fd);
   close(fd);
   execvp(somefile, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
pid = wait(&status);
```

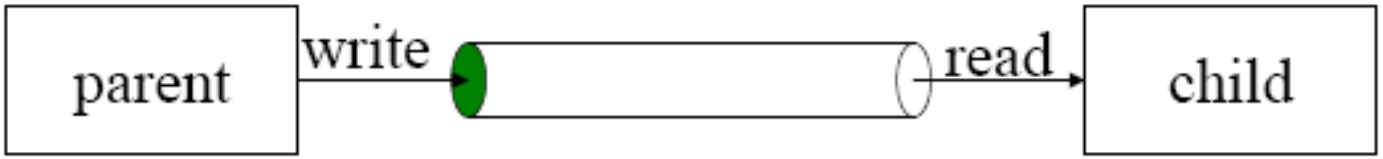
Somepgm exits; parent returns from **`wait()`** and proceeds

Pipe Example

```
int pid, p[2];
...
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    ... read using p[0] as fd until EOF ...
}
else {
    close(p[0]);
    ... write using p[1] as fd ...
    close(p[1]); /* sends EOF to reader */
    wait(&status);
}
```

child

parent

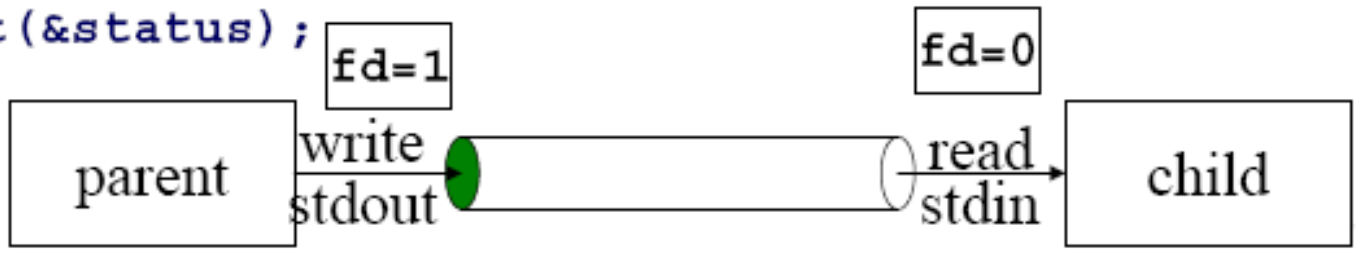


Pipes and Stdio

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    ... read from stdin ...
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child makes stdin (0)
the read side of the pipe

parent makes stdout (1)
the write side of the pipe

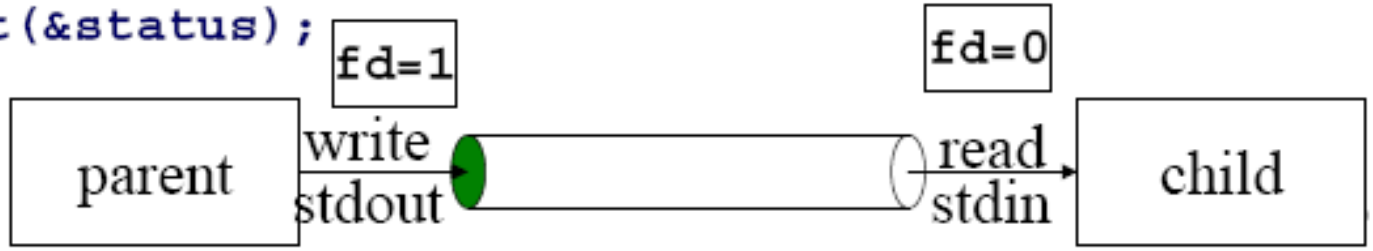


Pipes and Exec

```
int pid, p[2];
if (pipe(p) == -1)
    exit(1);
pid = fork();
if (pid == 0) {
    close(p[1]);
    dup2(p[0],0);
    close(p[0]);
    execl(...);
}
else {
    close(p[0]);
    dup2(p[1],1);
    close(p[1]);
    ... write to stdout ...
    wait(&status);
}
```

child process

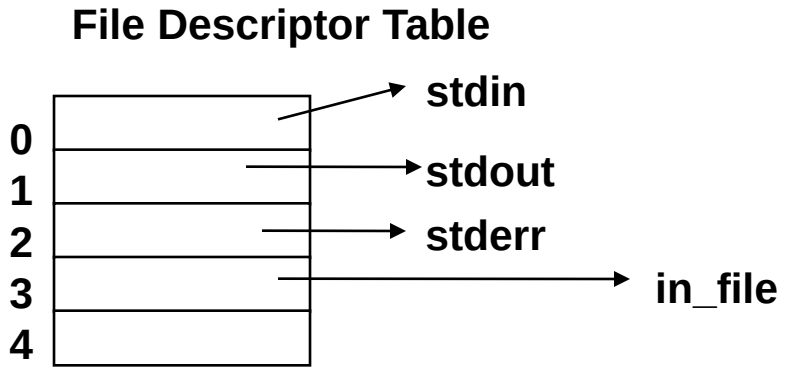
invokes a new program



Open Files Representation

- Assume that there was something like this in program

```
FILE *in_file;
   // open a file
in_file = fopen("list.txt", "r");
```



- in_file is a file descriptor

Open Files Representation

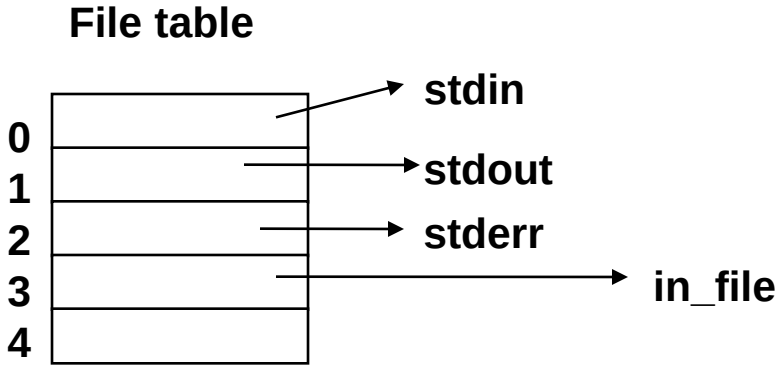
- Open Each Unix process should have the following three file descriptors:

Integer value	Name	<unistd.h> symbolic constant	<stdio.h> file stream
0	Standard input	STDIN_FILENO	stdin
1	Standard output	STDOUT_FILENO	stdout
2	Standard error	STDERR_FILENO	stderr

Open Files Representation

Assume that there was something like this program

```
FILE *in_file;
   // open a file
in_file = fopen("list.txt", "r");
```

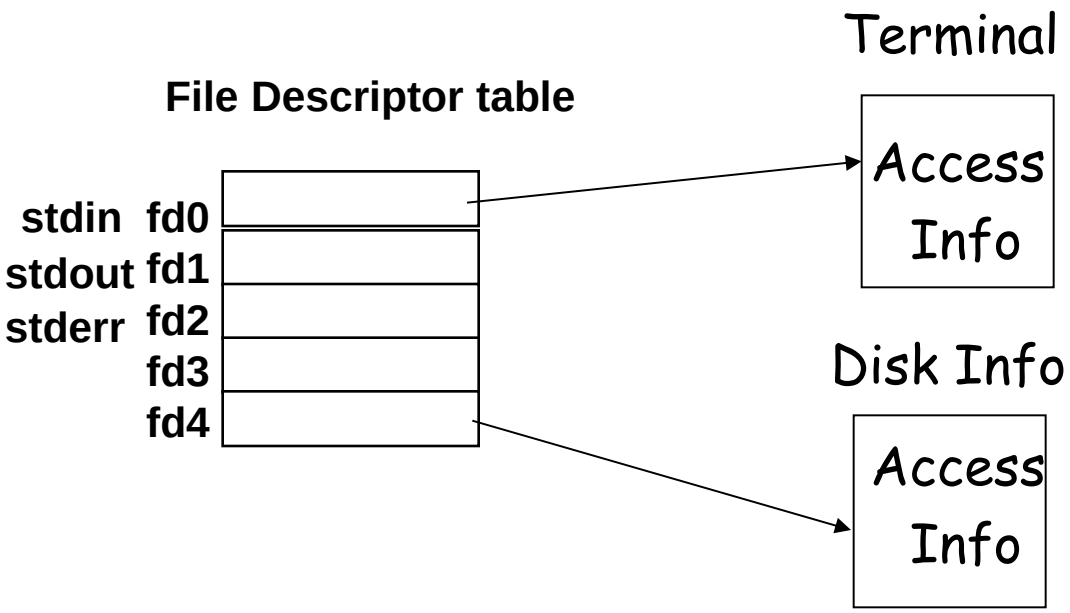


In a fork what gets copied is the file table;

- Each entry points to an area of memory with information about a file

Thus the parent and child point to the same open files

Open Files Representation



How do processes share information?

Pipes

We can think of pipes as a special file that can store a limited amount of data in a first-in-first-out (FIFO) manner

One process will write to the pipe (as if it were a file), while another process will read from the pipe

The system provides the synchronization between writing and reading processes

Pipes

Pipes can be used between processes that have a common ancestor

Typical use:

- Pipe created by a process
- Process calls fork()
- Pipe used between parent and child
- Allows for communication between processes

Pipes

By default, if a writing process attempts to *write* to a full pipe, the system will automatically block the process until the pipe is able to receive the data

Likewise, if a *read* is attempted on an empty pipe, the process will block until data is available

In addition, the process will block if a specified pipe has been opened for reading, but another process has not opened the pipe for writing

Creating a Pipe

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

Returns 0 if ok, -1 on error

Returns two file descriptors

- filedes[0] is open for reading
- filedes[1] is open for writing
- Output of filedes[1] is input to filedes[0]

Example

```
#include <unistd.h>
#include <stdio.h>
int main(void){

  int n;                // track of num bytes read
  int fd[2];            //  hold fds of both ends of pipe
  pid_t pid;            // pid of child process
  char line[80];        // hold text read/written
```

Continued …

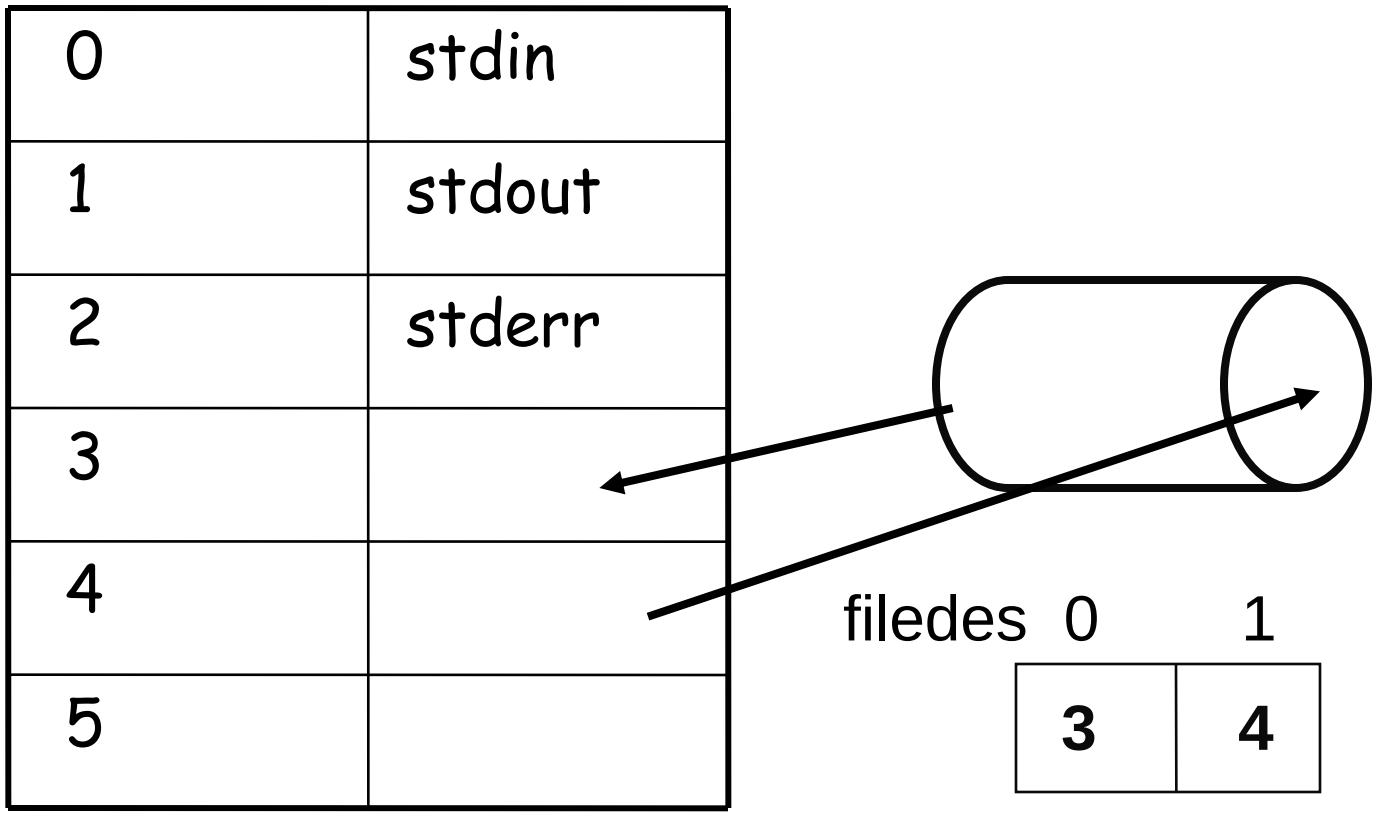
```
if (pipe(fd) < 0)                          // create the pipe
        perror("pipe error");

if ((pid = fork()) < 0) {                  // fork off a child
        perror("fork error");
  } else if (pid > 0) {                    // parent process
        close(fd[0]);                      // close  read end
        write(fd[1], "hello world\n", 12); // write to  it
}…
```

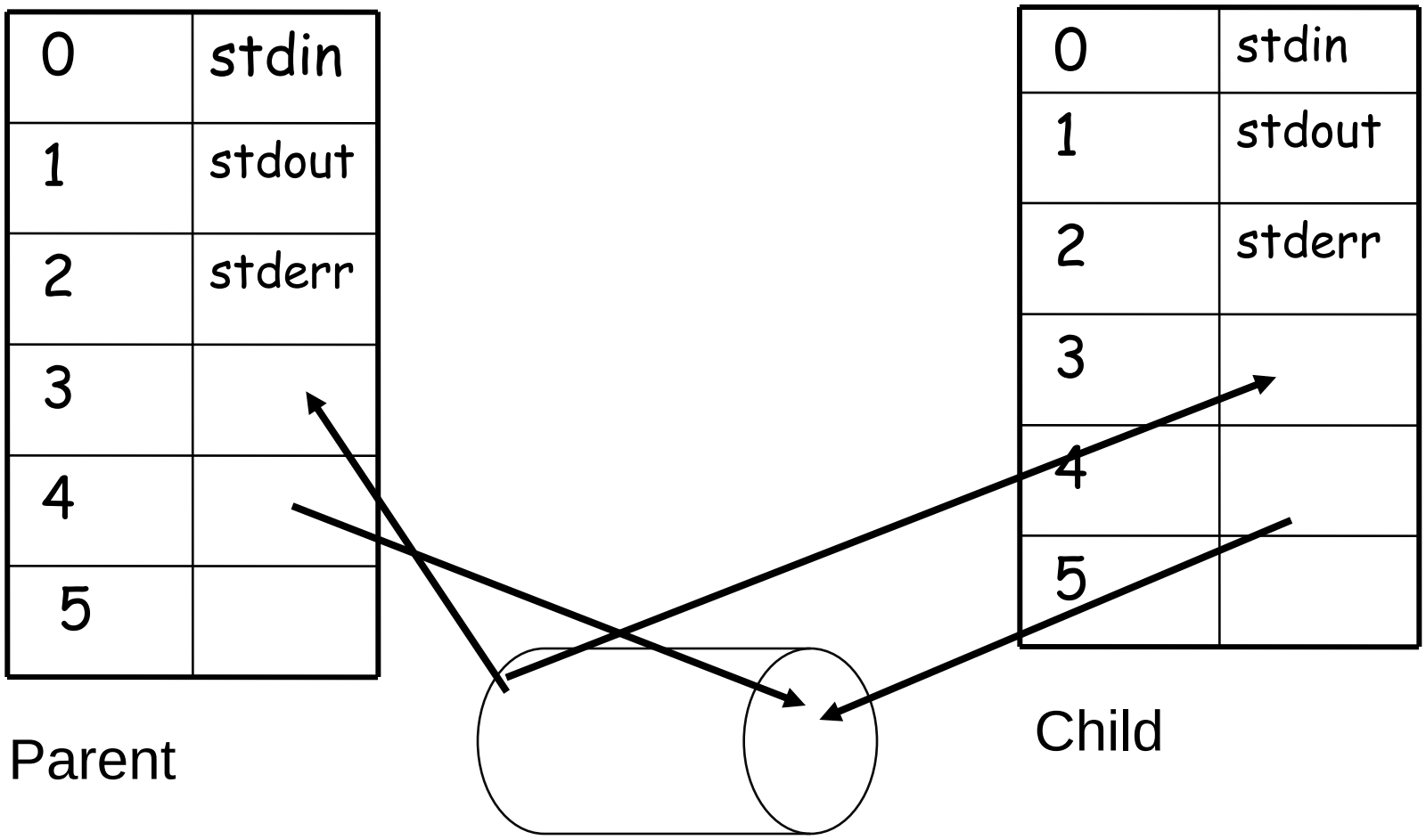
continued

```
   else {                                  // child process
        close(fd[1]);                      // close write end
        n = read(fd[0], line, 80);         // read from  pipe
        write(1, line, n);                 // echo  to  screen
  }
  exit(0);
}
```

After the pipe() call



The Process Calls fork()

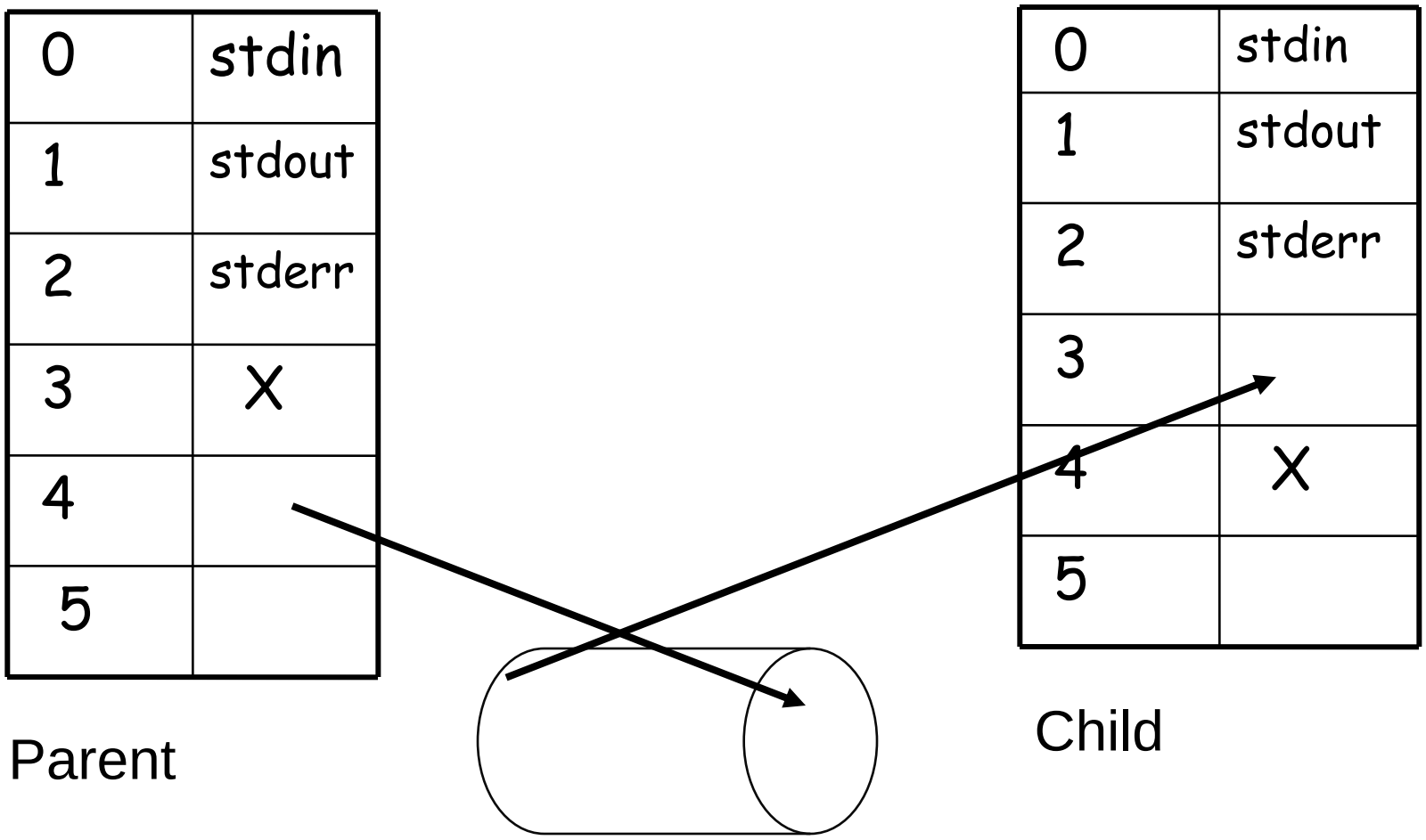


After Calling Fork,

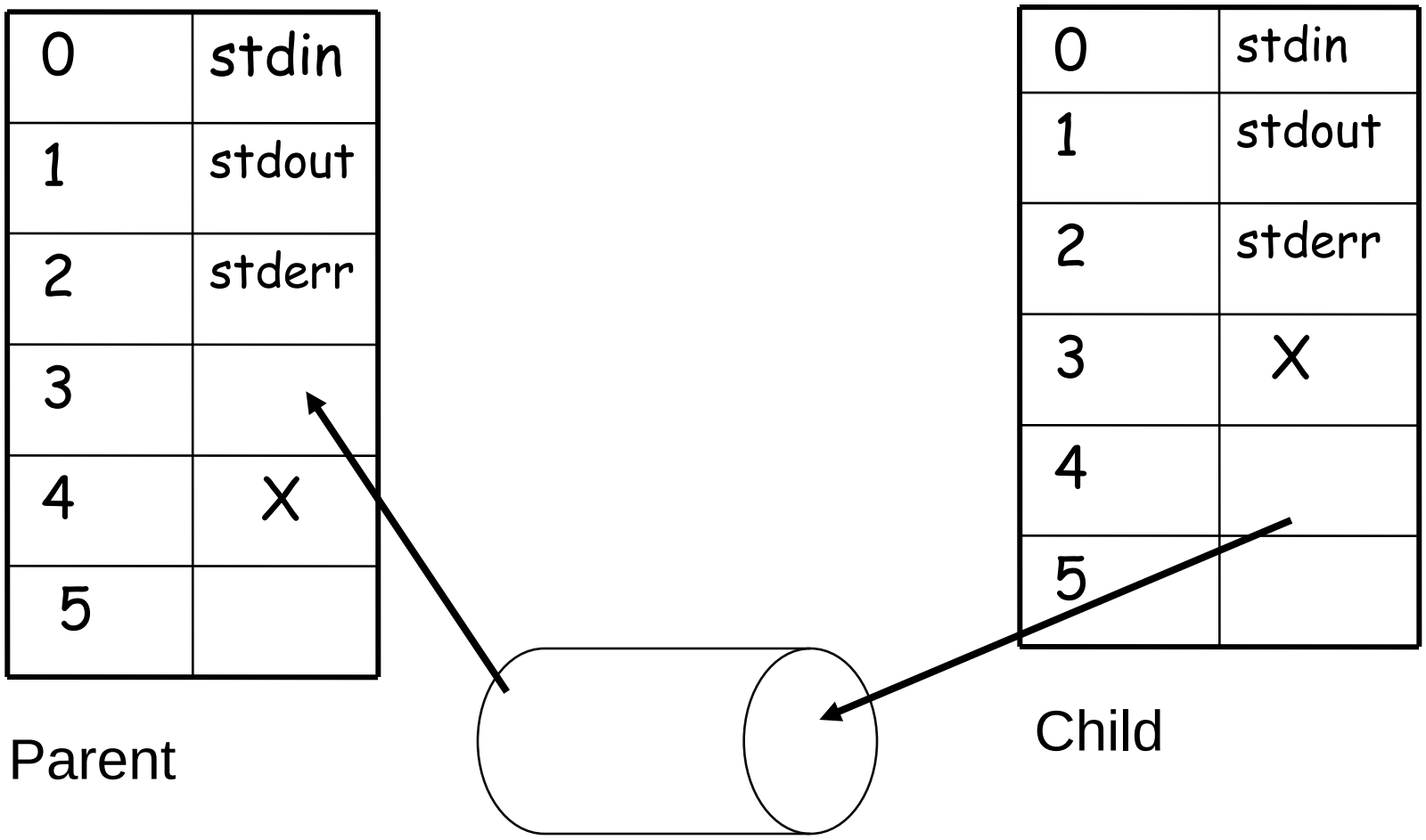
Close the read end in one process

Close the write end in the other process

Parent Writing to Child



Child Writing to Parent



After One End of the Pipe is closed …

Reading from a empty pipe whose write end has been closed returns 0 (indicating EOF)

Writing to a pipe whose read end has been closed generates a SIGPIPE signal

- If we ignore the signal or catch and return, handler returns -1, and errno set to EPIPE

Pipe Capacity

The OS has a limit on the buffer space used by the pipe

- If you hit the limit, write will block

dup() and dup2

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Both will duplicate an existing file descriptor

dup() returns lowest available file descriptor, now referring to whatever filedes refers to

dup2() - filedes2 (if open) will be closed and then set to refer to whatever filedes refers to

Implementing I/O Redirection

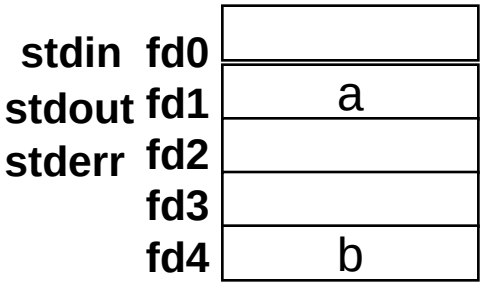
How does a shell implement I/O redirection?

ls > foo.txt

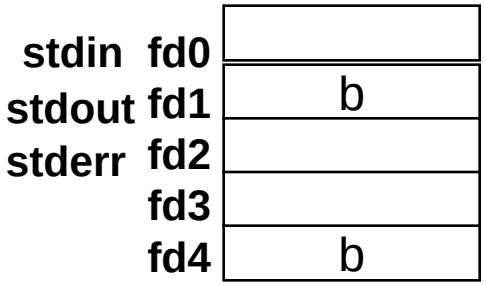
By calling the dup2(oldfd,newfd) function

- Copies descriptor table entry oldfd to entry newfd

File Descriptor table before dup2(4,1)



File Descriptor table after dup2(4,1)



Example

```
int main(int argc, char **argv) {
int fds[2];
int pid;

/* attempt to create a pipe  */
if (pipe(fds)<0) {
perror("Fatal Error");
exit(1);
}
```

Example

```
/* create another process */
  pid = fork();
  if (pid<0) {
     perror("Problem forking");
     exit(1);
  } else if (pid>0) {
     /* parent process */
     close(fds[0]);

     /* close stdout, reconnect to the writing end of the pipe */
     if ( dup2(fds[1],STDOUT_FILENO)<0) {
       perror("can't dup");
       exit(1);
     }

     execlp("ps","ps","-le", NULL);
     perror("exec problem");
     exit(1);
```

Example

```
} else {
    /* child process */

    close(fds[1]);
    if (dup2(fds[0],STDIN_FILENO) < 0) {
      perror("can't dup");
      exit(1);
    }
    execlp("sort","sort",NULL);
    perror("exec problem");
    exit(1);
  }

  return(0);
}
```

A Shell

A shell will perform the following tasks:

1. Display a prompt
2. Read a command line from the terminal
3. Start a background process to execute the command
4. Display another prompt and go back to step 1

A Shell (cont.)

- A shell will perform the following tasks:

```
int main()
{
   void background( char *command );
   for( ; ; )
    {
        printf(" your shell prompt\n");
        readLineOfInput();
        parseInput();
        background( command );
    }
    return;
```

Summary

- The fork, wait, exec, pipe, dup2 gives you all the functionality for creating a shell
- Not just shells
- Example: Apache servers forks processes to handle requests