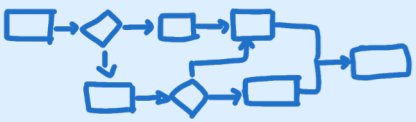
Imperial College London



Operating Systems
Processes

Course 211
Spring Term 2018-2019

http://www.imperial.ac.uk/computing/current-students/courses/211/calendar/

(Slides courtesy of Cristian Cadar)

Peter Pietzuch

prp@doc.ic.ac.uk
http://www.doc.ic.ac.uk/~prp

Introduction to Processes

One of the **oldest abstractions** in computing

- An instance of a program being executed, a running program

Allows a single processor to run multiple programs "simultaneously"

- Processes turn a single CPU into multiple virtual CPUs
- Each process runs on a virtual CPU

Why Have Processes?

Provide (the illusion of) concurrency

- Real vs. apparent concurrency

Provide isolation between programs

- Each process has its own address space

Simplicity of programming

- Firefox doesn't need to worry about gcc

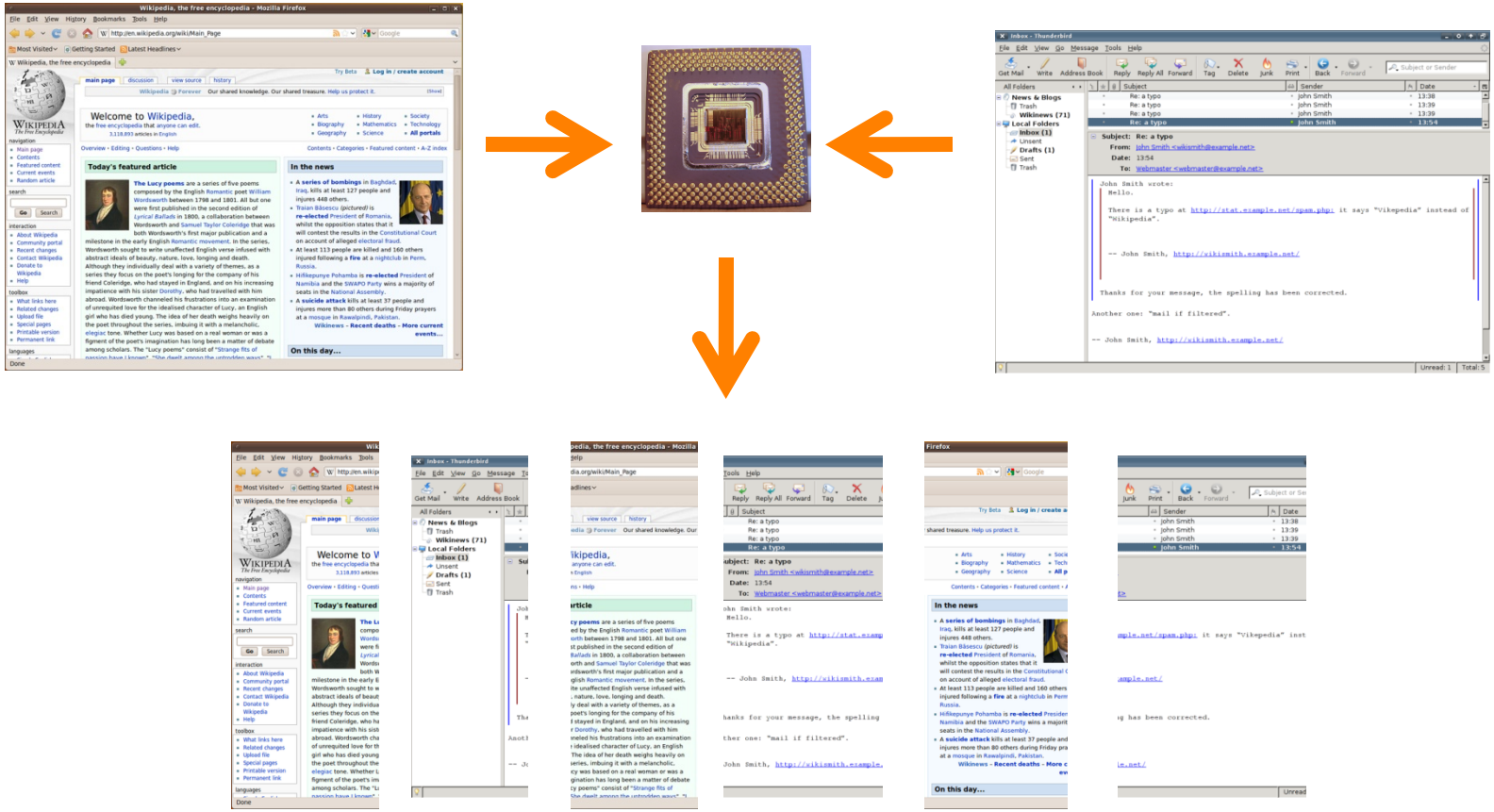
Allow better utilisation of resources

- Different processes require different resources at certain times

Time-Slicing For Concurrency

One example technique used is **time-slicing**

- OS switches applications running on physical CPU every 50ms

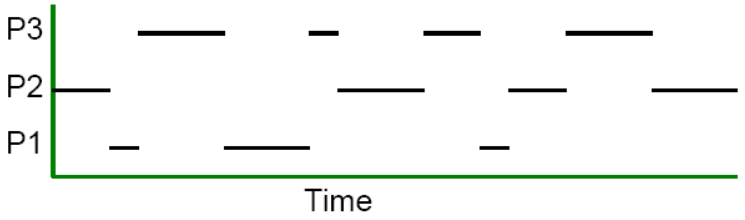


time

Types of Concurrency

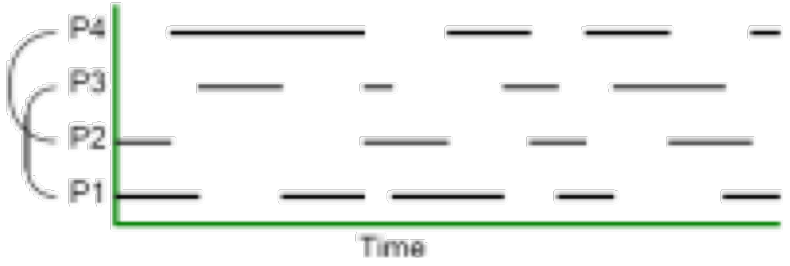
Pseudo concurrency

- Single physical processor is switched between processes by interleaving
- Over a period of time this gives the illusion of concurrent execution

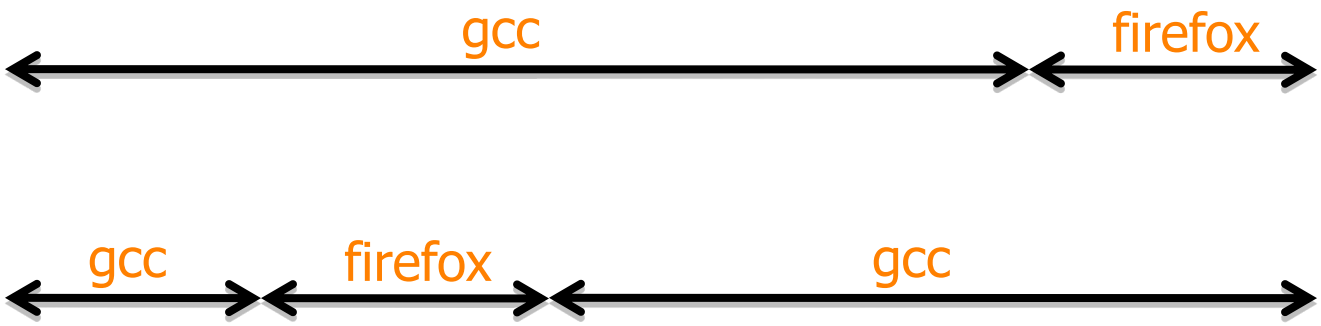


Real concurrency

- Utilises multiple physical processors (or CPU cores)
- Usually fewer processors than processes

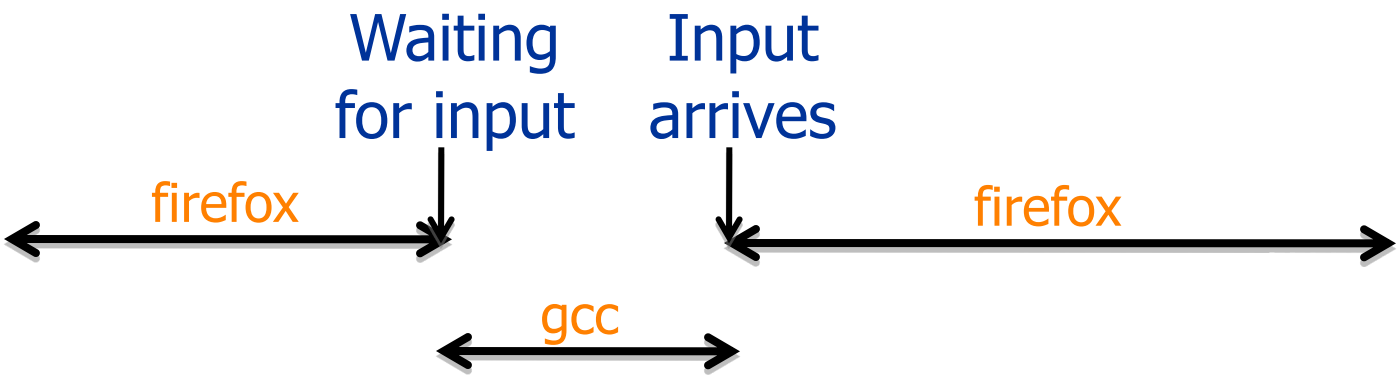


Fairness with Multiple Applications



How would users notice unfairness?

Better CPU Utilisation



Don’t occupy the CPU when waiting for I/O data

CPU Utilisation in Multiprogramming (Uniprocessor)

If on average a process computes 20% time, then with 5 processes we should have 100% CPU utilisation, right?

A: In the ideal case, if the five processes never wait for I/O at the same time

Better estimate

- **n** = total number of processes
- **p** = fraction of time a process is waiting for I/O

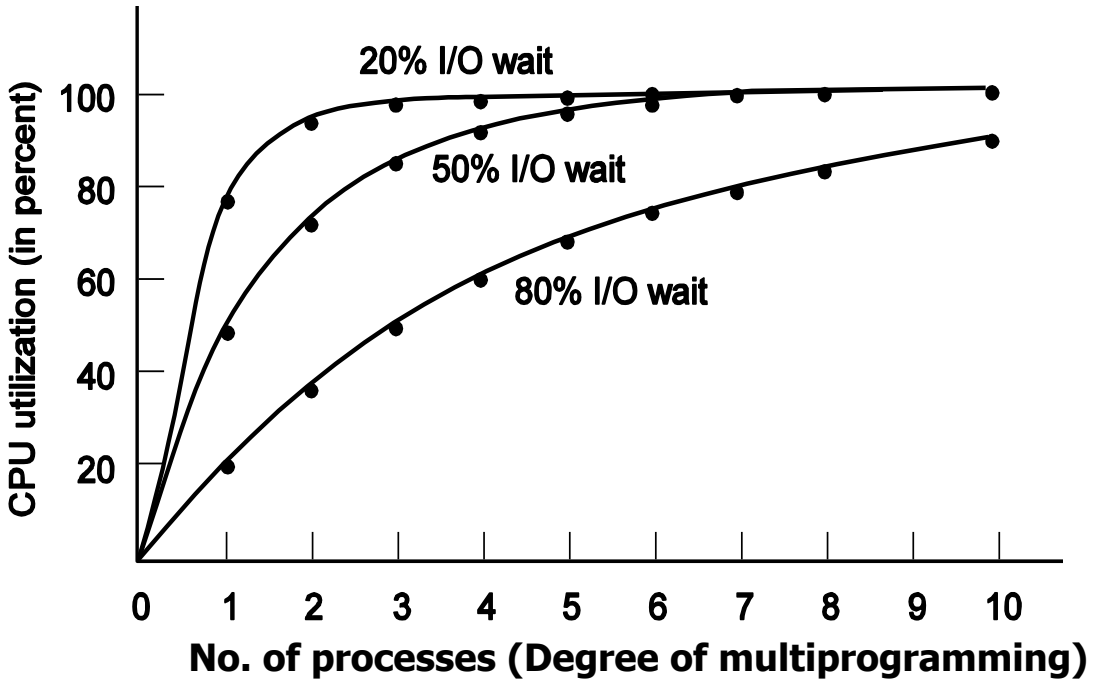
$$\text{Prob(all processes waiting for I/O)} = p^n$$

$$\text{CPU utilization} = 1 - p^n$$

Tutorial Question: CPU Utilisation = $1 - p^n$

How many processes need to be running to only waste 10% of CPU if they spend 80% waiting for I/O (i.e. this is a data-oriented or interactive system)?

A: $1 - 0.8^n = 0.9 \Rightarrow 0.8^n = 0.1 \Rightarrow n = \log_{0.8} 0.1 \approx 10$



Context Switches

On a **context switch**, the processor switches from executing process A to executing process B

OS may take periodic **scheduling** decisions

OS may switch processes in response to events/interrupts (e.g. I/O completion)

- OS switches between processes cannot be pre-determined because the events causing them are non-deterministic

Context Switches

On a **context switch**, the processor switches from executing process A to executing process B

Process A may be **restarted later**, therefore, all information concerning the process, needed to restart **safely**, should be stored

For each process, all this data is stored in a **process descriptor**, or **process control block (PCB)**, which is kept in the **process table**

Process Control Block (PCB)

Process has its own virtual machine, e.g.:

- Its own **virtual CPU**
- Its own **address space** (stack, heap, text, data etc.)
- Open **file descriptors**, etc.

What information should be stored?

- Program counter (PC), page table register, stack pointer, ...
- Process management info:
  - Process ID (PID), parent process, process group, priority, CPU used, ...
- File management info
  - Root directory, working directory, open file descriptors, ...

Context Switches Are Expensive

Direct cost: save/restore process state

Indirect cost: perturbation of memory caches, TLB

- Translation lookaside buffers (TLBs) → caches mappings of virtual addresses to physical addresses, and is typically flushed on a context switch
- More in memory management lectures

Important to avoid unnecessary context switches

Process Creation

When are processes created?

- System initialisation
- User request
- System call by a running process

Processes can be

- Foreground processes: interact with users
- Background processes: handle incoming mail, printing requests, ... (**daemons**)

Process Termination

Normal completion: Process completes execution of body

System call:

`exit()` in UNIX

`ExitProcess()` in Windows

Abnormal exit: Process has run into an error or an unhandled exception

Aborted: Process stops because another process has overruled its execution (e.g. killed from terminal)

Never: Some processes run in endless loop and never terminate unless error occurs (Examples?)

Process Hierarchies

UNIX allows processes to create **process hierarchies**, e.g. parent, child, child's child, ...

- When UNIX boots, it starts running `init`
- It reads a file saying how many terminals to run, and forks off one process per terminal
- They wait for someone to login
- When login successful, the login process executes a shell to accept commands which in turn may start more processes
- All processes in the entire system form a process tree with `init` as the root (**process group**)

Windows has no notion of hierarchy

- When a child process is created, the parent is given a token (**handle**) to use to control it
- The handle can be passed to other processes

Case Study: UNIX

Creating processes

```
int fork(void)
```

Creates new child process by making exact copy of parent process image

Child process inherits resources of parent process and will be executed concurrently with parent process

`fork()` returns twice:

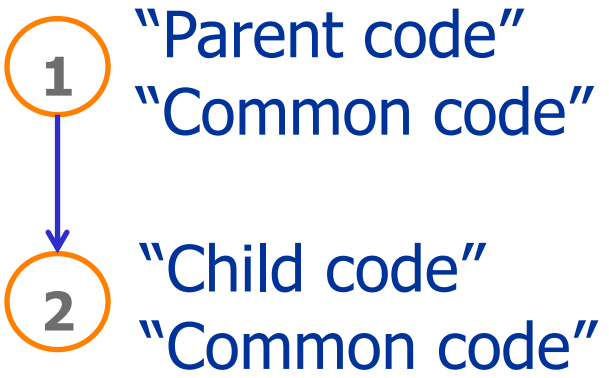
- In **parent** process: `fork()` returns process ID of child
- In **child** process: `fork()` returns 0

On error, no child is created, and -1 is returned to parent

How can `fork()` fail?

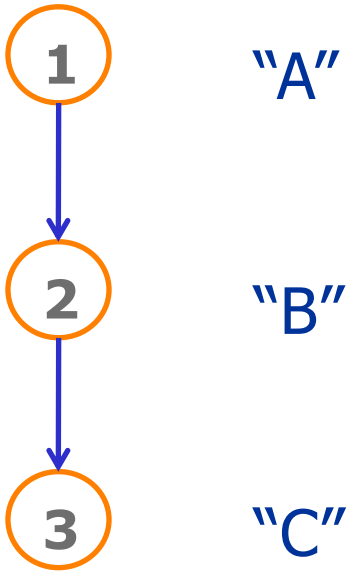
`fork()` Example I

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

int main() {
  if (fork() != 0)
    printf("Parent code\n");
  else printf("Child code\n");

  printf("Common code\n");
}
```



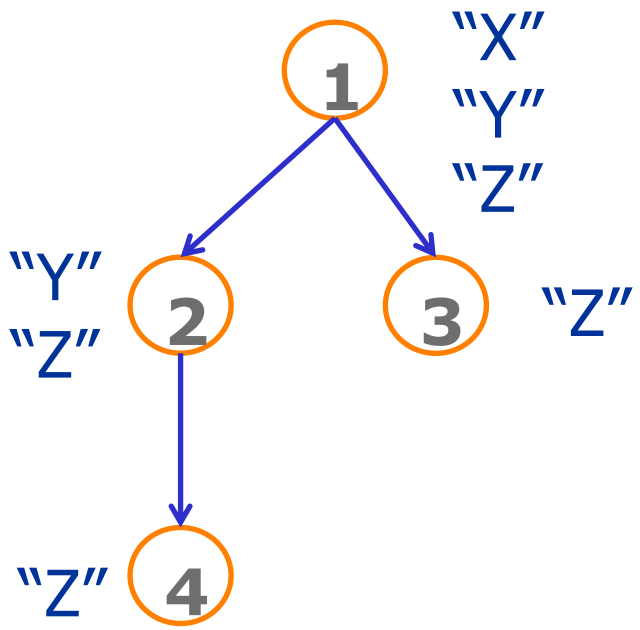
`fork()` Example II

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

int main() {
  if (fork() != 0)
    printf("A\n");
    else
      if (fork() != 0)
        printf("B\n");
      else printf("C\n");
}
```



Tutorial Question: `fork()`

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

int main() {
  if (fork() != 0)
    printf("X\n");

  if (fork() != 0)
    printf("Y\n");

  printf("Z\n");
}
```



Tutorial Question: `fork()`

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

int main() {
  if (fork() != 0)
    printf("%d: X\n",getpid());

  if (fork() != 0)
    printf("%d: Y\n",getpid());

  printf("%d: Z\n", getpid());
}
```

```
$ ./a.out
29221: X
29221: Y
29221: Z
29222: Y
29222: Z
29223: Z
29224: Z
```

Executing Processes

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

Arguments:

- **path** – full pathname of program to run
- **argv** – arguments passed to main
- **envp** – environment variables (e.g. $PATH, $HOME)

Changes process image and runs new process

Lots of useful wrappers:

- E.g. **`execl`**, **`execle`**, **`execvp`**, **`execv`**, …

```
man execve
```

Consult man(ual) pages!

Waiting For Process Termination

```
int waitpid(int pid, int* stat, int options)
```

Suspends execution of calling process until process with PID pid terminates normally or a signal is received

Can wait for more than one child:

- pid = -1 wait for any child
- pid = 0 wait for any child in the same process group as caller
- pid = -gid wait for any child with process group gid

Returns:

- pid of the terminated child process
- 0 if **`WNOHANG`** is set in options (indicating call should not block), and there are no terminated children
- -1 on error, with errno set to indicate the error

`fork()`, `execve()` and `waitpid()` in Action

A command interpreter could do:

```
while (TRUE)  { /* repeat forever */
  read_command (command,parameters)
  if (fork () != 0)     /* fork off child process */
    waitpid(-1, &status, 0);   /* Parent code */
  else          /* Child code */
    execve (command, parameters, 0);
                  /* execute command */
}
```

Why both `fork()` and `execve()`?

UNIX design philosophy: **simplicity**

- Simple basic blocks that can be easily combined

Contrast with Windows:

- `CreateProcess()` => equivalent of fork() + execve()
- Call has 10 parameters!
  - program to be executed
  - parameters
  - security attributes
  - meta data regarding files
  - priority
  - pointer to structure in which info regarding new process is stored and communicated to caller
  - …

`CreateProcess()` in Windows

```
BOOL WINAPI CreateProcess(
      __in_opt LPCTSTR lpApplicationName,
      __inout_opt LPTSTR lpCommandLine,
      __in_opt LPSECURITY_ATTRIBUTES
lpProcessAttributes,
      __in_opt LPSECURITY_ATTRIBUTES
lpThreadAttributes,
      __in BOOL bInheritHandles,
      __in DWORD dwCreationFlags,
      __in_opt LPVOID lpEnvironment,
      __in_opt LPCTSTR lpCurrentDirectory,
      __in LPSTARTUPINFO lpStartupInfo,
      __out LPPROCESS_INFORMATION
lpProcessInformation )
```

Process Termination

```
void exit(int status)
```

Terminates a process

- Called implicitly when program finishes execution

Never returns in the calling process

- Returns an exit status to the parent process

```
void kill(int pid, int sig)
```

Sends signal sig to process pid

How Can Processes Communicate?

Files

Signals (UNIX)

Events, exceptions (Windows)

Pipes

Message Queues (UNIX)

Mailslots (Windows)

Sockets

Shared memory

Semaphores

UNIX Signals

Inter-Process Communication (IPC) mechanism

Signal delivery similar to delivery of hardware interrupts

- Used to notify processes when event occurs

Process can send signal to another process if it has permission to do so:

- "the real or effective user ID of the receiving process must match that of the sending process or the user must have appropriate privileges (such as given by a set-user-ID program or the user is the super-user)." (man page)
- Kernel can send signals to any process

When Are Signals Generated?

When an exception occurs

- e.g. division by zero => `SIGFPE`,
  segment violation => `SIGSEGV`

When the kernel wants to notify the process of an event

- e.g. if process writes to a closed pipe => `SIGPIPE`

When certain key combinations are typed in a terminal

- e.g., Ctrl-C => `SIGINT`

Programmatically using the `kill()` system call

Examples of UNIX Signals

`SIGINT`	Interrupt from keyboard
`SIGABRT`	Abort signal from **`abort`**
`SIGFPE`	Floating point exception
`SIGKILL`	Kill signal
`SIGSEGV`	Invalid memory reference
`SIGPIPE`	Broken pipe: write to pipe with no readers
`SIGALRM`	Timer signal from **`alarm`**
`SIGTERM`	Termination signal

UNIX Signals

Default action for most signals is to terminate the process

But the receiving process may choose to:

- Ignore it
- Handle it by installing a **signal handler**
- Two signals cannot be ignored/handled: `SIGKILL` and `SIGSTOP`

```
signal(SIGINT, my_handler);

void my_handler(int sig) {
    printf("Received SIGINT. Ignoring...")
}
```

Example: Signal Handlers

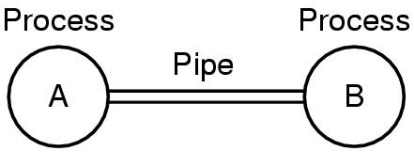
```
#include <signal.h>
#include <stdio.h>

void my_handler(int sig) {
  fprintf(stderr, "SIGINT
 caught!”);
}

int main(int argc, char *argv[])
{
  signal(SIGINT, my_handler);
  while (1) {}
}
```

```
$ ./a.out
[ctrl-C]
SIGINT caught
```

UNIX Pipes

A **pipe** is a method of connecting the standard output of one process to the standard input of another

- Allows for **one-way** communication between processes



Widely-used on the command line and in shell scripts

```
ls | less
cat file.txt | grep hello | wc -l
```

Two types of pipes

- **unnamed**
- **named**

pipe()

```
int pipe(int fd[2])
```

Returns two file descriptors in **fd**:

- `fd[0]` – the read end of the pipe
- `fd[1]` – the write end of the pipe

Sender should close the read end

Receiver should close the write end

If receiver reads from empty pipe, it blocks until data is written at the other end

If sender attempts to write to full pipe, it blocks until data is read at the other end

`pipe()` example

```
int main(int argc, char *argv[]) {
  int fd[2]; char buf;
  assert(argc == 2);
  if (pipe(fd) == -1) exit(1);

  if (fork() != 0) {
    close(fd[0]);
    write(fd[1], argv[1],
 strlen(argv[1]));
    close(fd[1]);
    waitpid(-1, NULL, 0);
  } else {
    close(fd[1]);
    while (read(fd[0], &buf, 1) > 0)
      printf("%c", buf);
    printf("\n");
    close(fd[0]);
  }
}
```

```
$ ./a.out abc
abc
```

UNIX Named Pipes (FIFOs)

Persistent pipes that outlive process which created them

Stored on file system

Any process can open it like regular file

- Why ever use named pipes instead of files?

```
$ mkfifo /tmp/abc
$ echo ABC >/tmp/abc
```

```
$ cat /tmp/abc
ABC
```

Tutorial Question

1. When two processes processes communicate through a pipe, the kernel allocated a buffer (say 64KB). What happens when the process at the write-end of the pipe attempts to send additional bytes on a full pipe?

2. What happens when the process at the write-end of the pipe attempts to send additional bytes but the other process already closed the file descriptor associated with the pipe?

3. The process at the write-end of the pipe wants to transmit a linked list data structure (with one integer field and a “next” pointer) over a pipe. How can it do this?