Unix Processes

- A **process** is an executing program (task); an execution *context*
Unix Processes

- A **process** is an executing program (task); an execution *context*
- Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)
Unix Processes

- A **process** is an executing program (task); an execution *context*
- Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)
- Child processes have their own virtual address space (**virtual memory**).
Unix Processes

- A **process** is an executing program (task); an execution *context*
- Unix allows users and their programs to run multiple other programs concurrently (ie. *multi-tasking*)
- Child processes have their own virtual address space (*virtual memory*)
- Child processes may execute another program in their stead
Unix Processes

• A **process** is an executing program (task); an execution *context*

• Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)

• Child processes have their own virtual address space (*virtual memory*)

• Child processes may execute another program in their stead

• Executable files are programs held in storage
Unix Processes

• A **process** is an executing program (task); an execution *context*

• Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)

• Child processes have their own virtual address space (**virtual memory**)

• Child processes may execute another program in their stead

• Executable files are programs held in storage

• Executable files are *binary* files, created via compilation and linking into *machine code*
Unix Processes

• A **process** is an executing program (task); an execution *context*

• Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)

• Child processes have their own virtual address space (*virtual memory*)

• Child processes may execute another program in their stead

• Executable files are programs held in storage

• Executable files are *binary* files, created via compilation and linking into *machine code*

• Processes are assigned **pids** which are unique integer identifiers over the lifetime of the process
• Processes are all given unique identifier numbers (non-negative).
Processes are all given unique identifier numbers (non-negative).

PID=0: the *swapper*, a system process, part of the kernel
Processes are all given unique identifier numbers (non-negative).

- PID=0: the *swapper*, a system process, part of the kernel
- PID=1: usually *init (/etc/init or /sbin/init)*.
  Reads system files (/etc/rc*), starts daemons, etc.
  Normal user process, but supervisor privileged.
  Responsible for starting terminal shells, etc.
Shell Commands for Processes

**ps** This command is used to display a snapshot of information about current processes.

- **ps -e** to see every process on the system
- **ps -eLf** to get information about threads

There are *many* options to ps!
Shell Commands for Processes

ps  This command is used to display a snapshot of information about current processes.

   ps -e  to see every process on the system
   ps -eLf to get information about threads

There are many options to ps!

pgrep  look up processes based on their name or other attributes

   pgrep -u root will list processes owned by root
   pgrep vi will list processes containing “vi”
Shell Commands for Processes

ps  This command is used to display a snapshot of information about current processes.
    ps -e  to see every process on the system
    ps -eLf  to get information about threads

There are many options to ps!

pgrep  look up processes based on their name or other attributes
    pgrerp -u root  will list processes owned by root
    pgrerp vi  will list processes containing “vi”

pidof  find process id of a running program
    pidof proclname  give pid of named running process
    killall5  same program as pidof (but with different behavior)
Shell Commands for Processes

**ps**  This command is used to display a snapshot of information about current processes.

- **ps -e**  to see every process on the system
- **ps -eLf**  to get information about threads

There are *many* options to **ps**!

**pgrep**  look up processes based on their name or other attributes

- **pgrep -u root**  will list processes owned by root
- **pgrep vi**  will list processes containing “vi”

**pidof**  find process id of a running program

- **pidof proccname**  give pid of named running process
- **killall5**  same program as pidof (but with different behavior)

**pkill**  will signal processes using same options as does **pgrep**
Unix Processes

- System memory is divided into two parts: user space and kernel space
Unix Processes

• System memory is divided into two parts: user space and kernel space

• Process memory has its own private address space; usually a process is divided into three segments (aka regions):
Unix Processes

• System memory is divided into two parts: user space and kernel space

• Process memory has its own private address space; usually a process is divided into three segments (aka regions):
  
  **text** the “text” segment contains the executable program’s instructions (code) and constant data. Unix marks this segment as read-only; the process cannot modify it. Processes may share the text segment with other processes (ie. threads)
Unix Processes

• System memory is divided into two parts: user space and kernel space

• Process memory has its own private address space; usually a process is divided into three segments (aka regions):
  
  **text**  the “text” segment contains the executable program’s instructions (code) and constant data. Unix marks this segment as read-only; the process cannot modify it. Processes may share the text segment with other processes (ie. threads)

  **data**  is further subdivided into initialized data (static variables) or uninitialized (aka bss = Block Started by Symbol). A process may request additional data segment space (via the brk() and sbrk() function calls); this area of memory is called the heap
Unix Processes

- System memory is divided into two parts: user space and kernel space.

- Process memory has its own private address space; usually a process is divided into three segments (*aka regions*):
  
  **text** The “text” segment contains the executable program’s instructions (*code*) and constant data. Unix marks this segment as read-only; the process cannot modify it. Processes may share the text segment with other processes (*ie. threads*).

  **data** is further subdivided into initialized data (*static variables*) or uninitialized (*aka bss = Block Started by Symbol*). A process may request additional data segment space (*via the brk() and sbrk() function calls*); this area of memory is called the **heap**.

  **stack** The stack is used for automatic variables, register variables, function arguments, and function calls.
Unix Processes

• System memory is divided into two parts: user space and kernel space

• Process memory has its own private address space; usually a process is divided into three segments (aka regions):

  text  the “text” segment contains the executable program’s instructions (code) and constant data. Unix marks this segment as read-only; the process cannot modify it. Processes may share the text segment with other processes (ie. threads)

  data  is further subdivided into initialized data (static variables) or uninitialized (aka bss = Block Started by Symbol). A process may request additional data segment space (via the brk() and sbrk() function calls); this area of memory is called the heap.

  stack  The stack is used for automatic variables, register variables, function arguments, and function calls.

• The o/s also maintains, for each process, a region called the u-area (user area). This region holds open file tables, current directory, signal actions, accounting information, system stack segment. When the process makes a system call, the stack frame information is stored in the process’ system stack segment, which the process doesn’t have access to. (see struct user in sys/user.h)
# Standard Segment Layout

<table>
<thead>
<tr>
<th>Address Space</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernel Space</strong></td>
<td>User code cannot read from/write to these addresses; attempts to do so results in a Segmentation Fault</td>
</tr>
<tr>
<td><strong>Stack</strong></td>
<td>Random stack offset, grows down</td>
</tr>
<tr>
<td><strong>Memory Mapping Segment</strong></td>
<td>File mappings (including dynamic libraries) and anonymous mappings</td>
</tr>
<tr>
<td></td>
<td>Random mmap offset, grows down</td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td>Program break, grows up, start_brk, random brk offset</td>
</tr>
<tr>
<td><strong>BSS Segment</strong></td>
<td>Uninitialized static variables, filled with zeros</td>
</tr>
<tr>
<td><strong>Data Segment</strong></td>
<td>Programmer-initialized static variables</td>
</tr>
<tr>
<td><strong>Text Segment</strong></td>
<td>Binary image of the process</td>
</tr>
</tbody>
</table>

*(see segments.c)*
Process States

user program is executing in user mode
Process States

**user** program is executing in user mode

**kernel** program is executing in kernel mode
Process States

- **user** program is executing in user mode
- **kernel** program is executing in kernel mode
- **ready to run, in memory** ready to run when scheduled by the kernel
Process States

**user**  program is executing in user mode

**kernel**  program is executing in kernel mode

**ready to run, in memory**  ready to run when scheduled by the kernel

**sleeping, in memory**  unable to run until some event (typically, some signal) occurs.

  Process is in memory, but blocked.
Process States

**user**  program is executing in user mode

**kernel**  program is executing in kernel mode

**ready to run, in memory**  ready to run when scheduled by the kernel

**sleeping, in memory**  unable to run until some event (typically, some signal) occurs.
  Process is in memory, but blocked.

**ready to run, swapped**  process must be brought in from swap memory before kernel can schedule it for execution
Process States

- **user** program is executing in user mode
- **kernel** program is executing in kernel mode
- **ready to run, in memory** ready to run when scheduled by the kernel
- **sleeping, in memory** unable to run until some event (typically, some signal) occurs. Process is in memory, but blocked.
- **ready to run, swapped** process must be brought in from swap memory before kernel can schedule it for execution
- **sleeping, swapped** process is blocked, awaiting some event, and has been transferred to swap memory. (blocked)
Process States

**user** program is executing in user mode

**kernel** program is executing in kernel mode

**ready to run, in memory** ready to run when scheduled by the kernel

**sleeping, in memory** unable to run until some event (typically, some signal) occurs. Process is in memory, but blocked.

**ready to run, swapped** process must be brought in from swap memory before kernel can schedule it for execution

**sleeping, swapped** process is blocked, awaiting some event, and has been transferred to swap memory. (blocked)

**preempted** Process was returning from kernel to user mode, but the kernel has preempted it, having scheduled another process
Process States

**user** program is executing in user mode

**kernel** program is executing in kernel mode

**ready to run, in memory** ready to run when scheduled by the kernel

**sleeping, in memory** unable to run until some event (typically, some signal) occurs. Process is in memory, but blocked.

**ready to run, swapped** process must be brought in from swap memory before kernel can schedule it for execution

**sleeping, swapped** process is blocked, awaiting some event, and has been transferred to swap memory. (blocked)

**preempted** Process was returning from kernel to user mode, but the kernel has preempted it, having scheduled another process

**created** Process is newly created (not yet ready to run)
Process States

**user** program is executing in user mode

**kernel** program is executing in kernel mode

**ready to run, in memory** ready to run when scheduled by the kernel

**sleeping, in memory** unable to run until some event (typically, some signal) occurs. Process is in memory, but blocked.

**ready to run, swapped** process must be brought in from swap memory before kernel can schedule it for execution

**sleeping, swapped** process is blocked, awaiting some event, and has been transferred to swap memory. (blocked)

**preempted** Process was returning from kernel to user mode, but the kernel has preempted it, having scheduled another process

**created** Process is newly created (not yet ready to run)

**zombie** Process no longer exists, but a record of its termination status is available
Process States and Transitions

- User Running
  - sys call or interrupt
  - return
- Kernel Running
  - sleep
  - schedule process
- Asleep
  - Context switch possible
  - wakeup
- Ready to Run

Interrupt, Interrupt return
Process Table

• The Process Table contains an entry for each process in the system, running or quiescent
Process Table

- The Process Table contains an entry for each process in the system, running or quiescent
- Entries are created by a `fork()` system call
The Process Table contains an entry for each process in the system, running or quiescent.
Entries are created by a `fork()` system call.
There are several entries in the Process Table for each process:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tick</td>
</tr>
<tr>
<td>1</td>
<td>pid</td>
</tr>
<tr>
<td>2</td>
<td>ppid</td>
</tr>
<tr>
<td>3</td>
<td>userid</td>
</tr>
<tr>
<td>4</td>
<td>state</td>
</tr>
<tr>
<td>6</td>
<td>swap flag</td>
</tr>
<tr>
<td>7</td>
<td>inode index</td>
</tr>
<tr>
<td>8</td>
<td>input buffer</td>
</tr>
<tr>
<td>9</td>
<td>mode flag</td>
</tr>
<tr>
<td>10</td>
<td>user area swap status</td>
</tr>
<tr>
<td>11</td>
<td>user area page number</td>
</tr>
<tr>
<td>12</td>
<td>kernel stack pointer</td>
</tr>
<tr>
<td>13</td>
<td>user stack pointer</td>
</tr>
<tr>
<td>14</td>
<td>ptbr</td>
</tr>
<tr>
<td>15</td>
<td>ptrl</td>
</tr>
</tbody>
</table>
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program
Process Image

User-Level Context

**Process Text** Executable machine-level instructions for the program

**Process Data** Data accessible by the process’ program
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program

**Process Data**  Data accessible by the process’ program

**User Stack**  Contains arguments, local variables, pointers for functions executing in user mode
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program

**Process Data**  Data accessible by the process’ program

**User Stack**  Contains arguments, local variables, pointers for functions executing in user mode

**Shared Memory**  Memory shared with other processes (interprocess communications)
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program

**Process Data**  Data accessible by the process’ program

**User Stack**  Contains arguments, local variables, pointers for functions executing in user mode

**Shared Memory**  Memory shared with other processes (interprocess communications)

System-Level Context

**Process Table Entry**  Defines the state of a process *(always accessible to kernel)*
Process Image

User-Level Context

**Process Text** Executable machine-level instructions for the program

**Process Data** Data accessible by the process’ program

**User Stack** Contains arguments, local variables, pointers for functions executing in user mode

**Shared Memory** Memory shared with other processes (interprocess communications)

System-Level Context

**Process Table Entry** Defines the state of a process *(always accessible to kernel)*

**U (user) Area** Process control information that is accessible only in the context of the process
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program

**Process Data**  Data accessible by the process’ program

**User Stack**  Contains arguments, local variables, pointers for functions executing in user mode

**Shared Memory**  Memory shared with other processes (interprocess communications)

System-Level Context

**Process Table Entry**  Defines the state of a process *(always accessible to kernel)*

**U (user) Area**  Process control information that is accessible only in the context of the process

**Per Process Region Table**  Defines the mapping from virtual to physical addresses, a permission field for type of accessed allowed: read-only, read-write, read-execute
Process Image

User-Level Context

**Process Text**  Executable machine-level instructions for the program

**Process Data**  Data accessible by the process’ program

**User Stack**  Contains arguments, local variables, pointers for functions executing in user mode

**Shared Memory**  Memory shared with other processes (interprocess communications)

System-Level Context

**Process Table Entry**  Defines the state of a process *(always accessible to kernel)*

**U (user) Area**  Process control information that is accessible only in the context of the process

**Per Process Region Table**  Defines the mapping from virtual to physical addresses, a permission field for type of accessed allowed: read-only, read-write, read-execute

**Kernel Stack**  Contains the stack frame of kernel procedures for when the process executes in kernel mode
**Unix Definitions**

**session** a session is a group of processes identified by a common id called a session id. It is capable of establishing a connection with a controlling terminal. Each session is associated with one “login” session. Newly created child processes join the session of its parent process.
Unix Definitions

**session** a session is a group of processes identified by a common id called a session id. It is capable of establishing a connection with a controlling terminal. Each session is associated with one “login” session. Newly created child processes join the session of its parent process.

**session leader** A session leader is the initial process of a session. All other processes in the session are descended from it. Consequently, all processes descended from a process leader are in the same session *unless* they create sessions of their own. If the session has a controlling terminal, then the session leader is also called a *controlling process*.
**Unix Definitions**

**session**  
a session is a group of processes identified by a common id called a session id.  
It is capable of establishing a connection with a controlling terminal.  
Each session is associated with one “login” session.  
Newly created child processes join the session of its parent process.

**session leader**  
A session leader is the initial process of a session.  
All other processes in the session are descended from it.  
Consequently, all processes descended from a process leader are in the same session *unless* they create sessions of their own.  
If the session has a controlling terminal, then the session leader is also called a *controlling process*.

**controlling terminal**  
A controlling terminal is a terminal device that is associated with a session.  
A session can only have one or no controlling terminals.  
A terminal device cannot control more than one session.
**controlling process** a session leader that has established a connection to a controlling terminal.

Processes within the session controlled by a terminal are the only ones subject to job control operations from that terminal. \((\text{ctrl-z}, \text{fg}, \text{bg}, \text{jobs})\)

Terminal devices only send signals to processes within its own session.
Unix Definitions, con’t.

**controlling process** a session leader that has established a connection to a controlling terminal.

Processes within the session controlled by a terminal are the only ones subject to job control operations from that terminal. (*ctrl-z*, *fg*, *bg*, *jobs*)

Terminal devices only send signals to processes within its own session.

**process group** a group of processes that are handled together for job control purposes.

Child processes are in their parent’s process group by default.

Child processes may be moved into another process group within the same session.

A process group leader is the first process in a newly created process group.

There may be processes in the group which were not descended from the group leader.
Unix Definitions, con’t.

**foreground process group** each session that has established a connection with a controlling terminal specifies one process group of the session as the *foreground process group*. 
Unix Definitions, con’t.

**foreground process group** each session that has established a connection with a controlling terminal specifies one process group of the session as the *foreground process group*.

**background process group** any process group not the foreground process group of a session that is connected with a controlling terminal.
Unix Definitions, con’t.

**foreground process group** each session that has established a connection with a controlling terminal specifies one process group of the session as the *foreground process group*.

**background process group** any process group not the foreground process group of a session that is connected with a controlling terminal.

**process group leader** a process whose process id is the same as its process group id
Unix Definitions, con’t.

**foreground process group** each session that has established a connection with a controlling terminal specifies one process group of the session as the *foreground process group*.

**background process group** any process group not the foreground process group of a session that is connected with a controlling terminal.

**process group leader** a process whose process id is the same as its process group id

**process group id** every active process is a member of a process group, and is identified by an integer (*process group id*).

This grouping permits signaling of related processes (ex. if one exits a terminal, all of its processes are killed)
**Unix Definitions**, con’t.

**foreground process group** each session that has established a connection with a controlling terminal specifies one process group of the session as the *foreground process group*.

**background process group** any process group not the foreground process group of a session that is connected with a controlling terminal.

**process group leader** a process whose process id is the same as its process group id

**process group id** every active process is a member of a process group, and is identified by an integer (*process group id*).

This grouping permits signaling of related processes (ex. *if one exits a terminal, all of its processes are killed*).

**orphaned process group** a process group in which the parent of every member in the group is either itself a member of the group or is not a member of the process group’s session. (ie. *There is no process that can handle job control signals for the process group*)
**Unix Definitions**, con’t.

**Process ID** each active process in the system has received a unique, individual positive integer to identify it.

The range of this pid is 1 to PID_MAX. *(run seelimits.c)*
Unix Definitions, con’t.

**Process ID** each active process in the system has received a unique, individual positive integer to identify it. The range of this pid is 1 to PID_MAX. *(run seelimits.c)*

**process lifetime** begins when the process is originally forked and ends after it exits. *(when its termination has been acknowledged)*
Unix Definitions, con’t.

**Process ID** each active process in the system has received a unique, individual positive integer to identify it.
The range of this pid is 1 to PID_MAX. *(run seelimits.c)*

**process lifetime** begins when the process is originally forked and ends after it exits.
(when its termination has been acknowledged)

**process group lifetime** begins when the session is created by its session leader, and ends when:
  * the last process that is a member of the session ends, or when
  * the last process that is a member of the session leaves by calling setsid().
Unix Definitions, con’t.

**Process ID** each active process in the system has received a unique, individual positive integer to identify it. The range of this pid is 1 to PID_MAX. *(run seelimits.c)*

**Process lifetime** begins when the process is originally forked and ends after it exits. (when its termination has been acknowledged)

**Process group lifetime** begins when the session is created by its session leader, and ends when:
- the last process that is a member of the session ends, or when
- the last process that is a member of the session leaves by calling setsid().

**Session lifetime** begins when the session is created by its session leader, and ends when:
- the last process that is a member of the session ends, or when
- the last process that is a member of the session leaves by calling setsid().
Unix Definitions, con’t.

**saved user id/saved group id** the values of the effective user id and effective group id prior to `exec()`ing a file
Unix Definitions, con’t.

**saved user id/saved group id** the values of the effective user id and effective group id prior to `exec()`ing a file

**special processes** processes with pids of 0 or 1 are *special processes*, typically referred to as `proc0` and `proc1`.

`proc0` is the scheduler

`proc1` is the initialization process (`init`). It is the ancestor of every other process in the system, and controls the process structure.
saved user id/saved group id  the values of the effective user id and effective group id prior to executing a file.

special processes  processes with pids of 0 or 1 are special processes, typically referred to as proc0 and proc1.
  - proc0 is the scheduler
  - proc1 is the initialization process (init). It is the ancestor of every other process in the system, and controls the process structure.

superuser  a process is recognized as a superuser process if its effective user id is zero.
  - Superuser processes have special privileges (such as immunity from file permissions).
Querying Process Attributes

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

pid_t getpid(void)  // get current process’ pid
```
Querying Process Attributes

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

pid_t getpid(void)  get current process’ pid

pid_t getppid(void)  get parent process’ pid
```
Querying Process Attributes

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

pid_t getpid(void)  // get current process' pid

pid_t getppid(void)  // get parent process' pid

pid_t getpgid(void)  // get process group id of a process
```
Querying Process Attributes

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

pid_t getpid(void)  // get current process’ pid

pid_t getppid(void) // get parent process’ pid

pid_t getpgrp(void) // get process group id of a process

uid_t getuid(void)  // get real user id of process
```
Querying Process Attributes

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

pid_t getpid(void)  // get current process’ pid

pid_t getppid(void)  // get parent process’ pid

pid_t getpgrp(void)  // get process group id of a process

uid_t getuid(void)  // get real user id of process

uid_t geteuid(void)  // get effective user id of process (which determines file access
                     // privileges, controlled by setuid bit)
```
Querying Process Attributes

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

pid_t getpid(void)  // get current process’ pid

pid_t getppid(void)  // get parent process’ pid

pid_t getpgrp(void)  // get process group id of a process

uid_t getuid(void)  // get real user id of process

uid_t geteuid(void)  // get effective user id of process (which determines file access
                   // privileges, controlled by setuid bit)

gid_t getgid(void)  // get real group id
```
#include <sys/types.h>
#include <unistd.h>

`pid_t getpid(void)` get current process’ pid

`pid_t getppid(void)` get parent process’ pid

`pid_t getpgrp(void)` get process group id of a process

`uid_t getuid(void)` get real user id of process

`uid_t geteuid(void)` get effective user id of process (which determines file access privileges, controlled by setuid bit)

`gid_t getgid(void)` get real group id

`gid_t getegid(void)` get effective group id (determines file access privileges, setgid bit)
Changing Process Attributes

**pid_t setsid(void)**  Process becomes a new session leader and a new process group leader.

session id ≡ group id ≡ pid

Disassociates process from controlling terminal. Good for daemon processes. Will fail if process already is a process group leader. The calling process will be the only process in this new process group and new session.
Changing Process Attributes

**pid_t setsid(void)**  Process becomes a new session leader and a new process group leader.
session id ≡ group id ≡ pid
Disassociates process from controlling terminal. Good for daemon processes. Will fail if process already is a process group leader. The calling process will be the only process in this new process group and new session.

**int setpgid(pid_t pid, pid_t pgid)**  process group id becomes pgid. Becomes new process group leader, and will be the only member of the new group.
Changing Process Attributes

**pid_t setsid(void)**  Process becomes a new session leader and a new process group leader.

session id ≡ group id ≡ pid

Disassociates process from controlling terminal. Good for daemon processes. Will fail if process already is a process group leader. The calling process will be the only process in this new process group and new session.

**int setpgid(pid_t pid, pid_t pgid)**  process group id becomes pgid. Becomes new process group leader, and will be the only member of the new group.

**int setuid(uid_t uid)**  if superuser, real and effective uid set to uid. Otherwise, can only change effective uid to its real uid. (-1 else)
Changing Process Attributes, con’t.

**int setgid(gid_t gid)** If superuser, set real and effective gid to `gid`. Otherwise, can only set effective gid to real gid. (-1 else)
Changing Process Attributes, con’t.

**int setgid(gid_t gid)**  If superuser, set real and effective gid to `gid`. Otherwise, can only set effective gid to real gid. (`-1` else)

**int seteuid(uid_t euid)**  Sets the effective user id of the calling process, if superuser. Otherwise, may only set it to the effective uid to real uid or the effective uid.
Changing Process Attributes, con’t.

**int setgid(gid_t gid)**  If superuser, set real and effective gid to `gid`. Otherwise, can only set effective gid to real gid. (-1 else)

**int seteuid(uid_t euid)**  Sets the effective user id of the calling process, if superuser. Otherwise, may only set it to the effective uid to real uid or the effective uid

**int setegid(gid_t egid)**  Sets the effective group id of the calling process, if superuser. Otherwise, may only set it to the effective gid to real gid or the effective gid
fork()

```
#include <unistd.h>

pid_t fork(void);  This function enables Unix multitasking.
```
fork()

#include <unistd.h>

pid_t fork(void);  This function enables Unix multitasking.

Init uses fork() to start other processes.
fork()

#include <unistd.h>

pid_t fork(void);  This function enables Unix multitasking.

Init uses fork() to start other processes.
A forked process (a child) has copies of the following from its parent process:
#fork()

```c
#include <unistd.h>

pid_t fork(void);
```

This function enables Unix multitasking. Init uses `fork()` to start other processes. A forked process (a **child**) has copies of the following from its parent process:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>real user id,</td>
<td>profiling status</td>
</tr>
<tr>
<td>real group id,</td>
<td>nice value (ie. priority)</td>
</tr>
<tr>
<td>effective user id,</td>
<td>scheduler class</td>
</tr>
<tr>
<td>effective group id</td>
<td>attached shared memory segments</td>
</tr>
<tr>
<td>environment</td>
<td>process group id</td>
</tr>
<tr>
<td>close-on-exec flag</td>
<td>session id</td>
</tr>
<tr>
<td>signal handling settings</td>
<td>current working directory</td>
</tr>
<tr>
<td>signal handling mask</td>
<td>root directory</td>
</tr>
<tr>
<td>supplementary group ids</td>
<td>filemode creation mask</td>
</tr>
<tr>
<td>set-user-id bit</td>
<td>resource limits</td>
</tr>
<tr>
<td>set-group-id bit</td>
<td>controlling terminal</td>
</tr>
</tbody>
</table>
Child Differences

A child differs from its parent process:
Child Differences

A child differs from its parent process:

- has unique pid
Child Differences

A child differs from its parent process:

- has unique pid
- has different ppid
Child Differences

A child differs from its parent process:

- has unique pid
- has different ppid
- does not inherit file locks
Child Differences

A child differs from its parent process:

• has unique pid
• has different ppid
• does not inherit file locks
• semadj values cleared
Child Differences

A child differs from its parent process:

• has unique pid
• has different ppid
• does not inherit file locks
• semadj values cleared
• does not inherit (process, text, data) locks
Child Differences

A child differs from its parent process:

• has unique pid
• has different ppid
• does not inherit file locks
• semadj values cleared
• does not inherit (process, text, data) locks
• does not inherit pending signals
Child Differences

A child differs from its parent process:

- has unique pid
- has different ppid
- does not inherit file locks
- semadj values cleared
- does not inherit (process, text, data) locks
- does not inherit pending signals
- alarm time reset to zero
Typical Usage of fork()

Note that `fork()` returns

+ : pid of child when parent process
0 : this is the child process
-1 : on failure (no memory, system process limit, etc)

```c
pid_t pid;
if ((pid = fork()) > 0) { /* parent process */ }
else if (pid == 0) { /* child process */ }
else { /* error */ }
```

(show runrace, see race.c; showsetsid.c)
vfork()

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

pid_t vfork(void);

Often forks are followed by exec(), which replaces the program (text, data, stack, etc).
**vfork()**

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

pid_t vfork(void);
```

Often forks are followed by `exec()`, which replaces the program (text, data, stack, etc). Consequently, since nearly all of `fork()`'s copying work is wasted, `vfork()` was created which does *not* create that copy.
vfork()

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

pid_t vfork(void);

Often forks are followed by exec(), which replaces the program (text, data, stack, etc). Consequently, since nearly all of fork()’s copying work is wasted, vfork() was created which does not create that copy.

*The only safe thing to do with a vfork() call is to follow it with an exec() family call.*
vfork()

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

pid_t vfork(void);

Often forks are followed by exec(), which replaces the program (text, data, stack, etc). Consequently, since nearly all of fork()’s copying work is wasted, vfork() was created which does not create that copy.

*The only safe thing to do with a vfork() call is to follow it with an exec() family call.*

Recent unix (including linux) versions have improved fork() so that it only copies memory pages that either the child or parent process wish to modify. This method is known as *copy-on-write.*
The only safe thing to do with a vfork() call is to follow it with an exec() family call. Recent unix (including linux) versions have improved fork() so that it only copies memory pages that either the child or parent process wish to modify. This method is known as copy-on-write. Thus, it is now just as efficient to use fork() as vfork(), yet fork() is safe to use.

The parent process is suspended when using vfork() until the child process terminates or calls execve(). The child process may terminate by calling _exit().
clone()

#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg,... /* pid_t */
*pid, struct user_desc *tls, pid_t *ctid */);

• Like fork(), clone() creates a new process.
clone()

#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg, ..., /* pid_t *pid, struct user_desc *tls, pid_t *ctid */);

• Like fork(), clone() creates a new process.

• Cloned processes share parts of the execution context, including memory space, file descriptors, and even signal handlers.
clone()

#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg, ... /* pid_t *pid, struct user_desc *tls, pid_t *ctid */);

• Like fork(), clone() creates a new process.

• Cloned processes share parts of the execution context, including memory space, file descriptors, and even signal handlers.

• When fnc(arg) returns, the child process terminates.
clone()

#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg,... /* pid_t
*pid, struct user_desc *tls, pid_t *ctid */);

- Like fork(), clone() creates a new process.
- Cloned processes share parts of the execution context, including memory space,
  file descriptors, and even signal handlers.
- When fnc(arg) returns, the child process terminates.
- Used primarily to support threads.
clone()

#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg,... /* pid_t pid, struct user_desc *tls, pid_t *ctid */);

- Like fork(), clone() creates a new process.
- Cloned processes *share parts of the execution context*, including memory space, file descriptors, and even signal handlers.
- When fnc(arg) returns, the child process terminates.
- Used primarily to support threads.
- childstack is used to specify the cloned child’s stackspace. (child cannot re-use stack space as child and parent, which share memory, are simultaneously active)
clone() 

```c
#include <sched.h>

pid_t clone(int (*fnc)(void *), void *childstack, int flags, void *arg,... /* pid_t *pid, struct user_desc *tls, pid_t *ctid */);
```

- Like fork(), clone() creates a new process.
- Cloned processes share parts of the execution context, including memory space, file descriptors, and even signal handlers.
- When fnc(arg) returns, the child process terminates.
- Used primarily to support threads.
- childstack is used to specify the cloned child’s stackspace. (child cannot re-use stack space as child and parent, which share memory, are simultaneously active)
- The low byte of flags contains the termination signal number when the child dies. Normally SIGCHLD.
```c
#include <unistd.h>
void _exit(int status);
```

- This function terminates the calling process *immediately*. Never fails!
`_exit()`

```c
#include <unistd.h>
void _exit(int status);
```

- This function terminates the calling process *immediately*. Never fails!
- Any open file descriptors which belong to the process are closed.
include <unistd.h>
void _exit(int status);

• This function terminates the calling process *immediately*. Never fails!
• Any open file descriptors which belong to the process are closed.
• Any children of the process are inherited by process #1 (init)
```c
#include <unistd.h>

void _exit(int status);
```

- This function terminates the calling process *immediately*. Never fails!
- Any open file descriptors which belong to the process are closed.
- Any children of the process are inherited by process #1 (init)
- The process’ parent is sent a SIGCHLD signal
exit()

#include <stdlib.h>
void exit(int status);

- All functions registered via atexit() and on_exit() are called (in reverse order of their registration)
`exit()`

```c
#include <stdlib.h>
void exit(int status);
```

- All functions registered via `atexit()` and `on_exit()` are called (in reverse order of their registration)
- All open streams are flushed and closed.
exit()

#include <stdlib.h>

void exit(int status);

- All functions registered via atexit() and on_exit() are called (in reverse order of their registration)
- All open streams are flushed and closed.
- Files created by tmpfile() are removed
exit()

#include <stdlib.h>

void exit(int status);

• All functions registered via atexit() and on_exit() are called (in reverse order of their registration)
• All open streams are flushed and closed.
• Files created by tmpfile() are removed
• The exit status is transmitted to the parent process
exit()

#include <stdlib.h>

void exit(int status);

- All functions registered via atexit() and on_exit() are called (in reverse order of their registration)
- All open streams are flushed and closed.
- Files created by tmpfile() are removed
- The exit status is transmitted to the parent process
  1. Parent has set SA_NOCLDWAIT, or has set the SIGCHLD handler to SIG_IGN, status is discarded
exit()

#include <stdlib.h>

void exit(int status);

- All functions registered via atexit() and on_exit() are called (in reverse order of their registration)
- All open streams are flushed and closed.
- Files created by tmpfile() are removed
- The exit status is transmitted to the parent process
  1. Parent has set SA_NOCLDWAIT, or has set the SIGCHLD handler to SIG_IGN, status is discarded
  2. If parent was waiting on the child it is notified of the exit status.
exit()

#include <stdlib.h>

void exit(int status);

• All functions registered via atexit() and on_exit() are called (in reverse order of their registration)

• All open streams are flushed and closed.

• Files created by tmpfile() are removed

• The exit status is transmitted to the parent process
  1. Parent has set SA_NOCLDWAIT, or has set the SIGCHLD handler to SIG_IGN, status is discarded
  2. If parent was waiting on the child it is notified of the exit status.
  3. If parent is not waiting, then the exiting process becomes a zombie process; resources are free’d but the process table slot retains the single byte holding its exit status.
wait(), waitpid()

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

pid_t wait(int *status);

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

- `wait()` suspends the parent process until (thereby removing the child’s zombie status)
wait(), waitpid()

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

pid_t wait(int *status);

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

• wait() suspends the parent process until
  • a signal is sent to the process

(thereby removing the child’s zombie status)
wait(), waitpid()

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

pid_t wait(int *status);

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

• wait() suspends the parent process until
  • a signal is sent to the process
  • a child process terminates (exits)

(thereby removing the child’s zombie status)
wait(), waitpid()

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

pid_t wait(int *status);

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

• wait() suspends the parent process until
  • a signal is sent to the process
  • a child process terminates (exits)
• wait() will wait for any child process it has to terminate

(thereby
removing the child’s zombie status)
wait(), waitpid()

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

pid_t wait(int *status);

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

- wait() suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (exits)
- wait() will wait for any child process it has to terminate
- wait() and waitpid() both return the child status in *status and the child pid as function return value

(thereby removing the child’s zombie status)
wait(), waitpid()

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (exits)
- `wait()` will wait for any child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child (thereby removing the child’s zombie status)
wait(), waitpid()

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

pid_t wait(int *status);

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

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (`exits`)
- `wait()` will wait for *any* child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child (`thereby removing the child’s zombie status`)
- `waitpid` allows the parent to specify which child it will wait upon
wait(), waitpid()

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

pid_t wait(int *status);

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

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (*exits*)
- `wait()` will wait for *any* child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child (*thereby removing the child’s zombie status*)
- `waitpid` allows the parent to specify which child it will wait upon

<table>
<thead>
<tr>
<th>child pid</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>wait for child with given pid</td>
</tr>
</tbody>
</table>
wait(), waitpid()

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

pid_t wait(int *status);

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

- wait() suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (exits)
- wait() will wait for any child process it has to terminate
- wait() and waitpid() both return the child status in *status and the child pid as function
  return value
- wait() and waitpid() will de-allocate the process table slot for zombie’d child (thereby
  removing the child’s zombie status)
- waitpid allows the parent to specify which child it will wait upon
  
<table>
<thead>
<tr>
<th>child pid</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>wait for child with given pid</td>
</tr>
<tr>
<td>-1</td>
<td>wait for any child</td>
</tr>
</tbody>
</table>
wait(), waitpid()

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

pid_t wait(int *status);

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

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (`exits`)
- `wait()` will wait for *any* child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child *(thereby removing the child’s zombie status)*
- `waitpid` allows the parent to specify which child it will wait upon
  
<table>
<thead>
<tr>
<th>Child PID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>wait for child with given pid</td>
</tr>
<tr>
<td>-1</td>
<td>wait for any child</td>
</tr>
<tr>
<td>0</td>
<td>wait for any child in same process group as parent</td>
</tr>
</tbody>
</table>
wait(), waitpid()

```c
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);

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

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (*execs*)
- `wait()` will wait for *any* child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child *(thereby removing the child’s zombie status)*
- `waitpid` allows the parent to specify which child it will wait upon
  
<table>
<thead>
<tr>
<th>child pid</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid</td>
<td>wait for child with given pid</td>
</tr>
<tr>
<td>-1</td>
<td>wait for any child</td>
</tr>
<tr>
<td>0</td>
<td>wait for any child in same process group as parent</td>
</tr>
<tr>
<td>≤ −2</td>
<td>wait for any child whose process group is</td>
</tr>
</tbody>
</table>
```
wait(), waitpid()

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

pid_t wait(int *status);

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

- `wait()` suspends the parent process until
  - a signal is sent to the process
  - a child process terminates (*exits*)
- `wait()` will wait for *any* child process it has to terminate
- `wait()` and `waitpid()` both return the child status in `*status` and the child pid as function return value
- `wait()` and `waitpid()` will de-allocate the process table slot for zombie’d child *(thereby removing the child’s zombie status)*
- `waitpid` allows the parent to specify which child it will wait upon
  ```
  child pid     description
  pid           wait for child with given pid
  -1            wait for any child
  0             wait for any child in same process group as parent
  \leq -2       wait for any child whose process group is \(|childpid|\)
  ```

- p.25/36
Waitpid Options and Status

waitpid options  WNOHANG  don’t block
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td>else</td>
<td>block</td>
<td></td>
</tr>
</tbody>
</table>
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>else</td>
<td>block</td>
</tr>
<tr>
<td></td>
<td>WNOTRACED</td>
<td>keep waiting for stopped child</td>
</tr>
</tbody>
</table>

*(job ctrl)*
### Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>else</td>
<td>block</td>
</tr>
<tr>
<td></td>
<td>WNOTRACE</td>
<td>keep waiting for stopped child</td>
</tr>
</tbody>
</table>

*status

<table>
<thead>
<tr>
<th>WIFEXITED(s)</th>
<th>≠ 0: child used _exit()</th>
</tr>
</thead>
<tbody>
<tr>
<td>= 0: else</td>
<td></td>
</tr>
</tbody>
</table>
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>else</td>
<td>block</td>
</tr>
<tr>
<td>WNOTRACED</td>
<td></td>
<td>keep waiting for stopped child</td>
</tr>
<tr>
<td>*status</td>
<td>WIFEXITED(s)</td>
<td>≠ 0: child used <code>_exit()</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>= 0: else</td>
</tr>
<tr>
<td>WEXITSTATUS(s)</td>
<td></td>
<td>returns child exit code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(via <code>_exit()</code>)</td>
</tr>
</tbody>
</table>
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>else</td>
<td>block</td>
</tr>
<tr>
<td></td>
<td>WNOTRACED</td>
<td>keep waiting for stopped child</td>
</tr>
<tr>
<td>*status</td>
<td>WIFEXITED(s)</td>
<td>(\neq 0): child used _exit()</td>
</tr>
<tr>
<td></td>
<td>= 0: else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WEXITSTATUS(s)</td>
<td>returns child exit code</td>
</tr>
<tr>
<td></td>
<td>(via _exit())</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WIFSIGNALED(s)</td>
<td>(\neq 0): if child terminated due to signal</td>
</tr>
</tbody>
</table>
# Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td>else</td>
<td></td>
<td>block</td>
</tr>
<tr>
<td>WNOTRACED</td>
<td></td>
<td>keep waiting for stopped child</td>
</tr>
</tbody>
</table>

*status

<table>
<thead>
<tr>
<th>status</th>
<th>WIFEXITED(s)</th>
<th>$\neq 0$: child used _exit()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>= 0: else</td>
<td></td>
</tr>
<tr>
<td>WEXITSTATUS(s)</td>
<td>returns child exit code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(via _exit())</td>
<td></td>
</tr>
<tr>
<td>WIFSIGNALED(s)</td>
<td>$\neq 0$: if child terminated due to signal</td>
<td></td>
</tr>
<tr>
<td>WTERMSIG(s)</td>
<td>returns signal # that terminated the child</td>
<td></td>
</tr>
</tbody>
</table>
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>Waitpid Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNOHANG</td>
<td>don’t block</td>
</tr>
<tr>
<td>else</td>
<td>block</td>
</tr>
<tr>
<td>WNOTRACED</td>
<td>keep waiting for stopped child</td>
</tr>
<tr>
<td>(job ctrl)</td>
<td></td>
</tr>
</tbody>
</table>

### Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFEXITED(s)</td>
<td>$\neq 0$: child used <code>_exit()</code></td>
</tr>
<tr>
<td></td>
<td>$= 0$: else</td>
</tr>
<tr>
<td>WEXITSTATUS(s)</td>
<td>returns child exit code (via <code>_exit()</code>)</td>
</tr>
<tr>
<td>WIFSIGNALED(s)</td>
<td>$\neq 0$: if child terminated due to signal</td>
</tr>
<tr>
<td>WTERMSIG(s)</td>
<td>returns signal # that terminated the child</td>
</tr>
<tr>
<td>WIFSTOPPED(s)</td>
<td>$\neq 0$: returns value if child stopped due to job control</td>
</tr>
</tbody>
</table>
## Waitpid Options and Status

<table>
<thead>
<tr>
<th>waitpid options</th>
<th>WNOHANG</th>
<th>don’t block</th>
</tr>
</thead>
<tbody>
<tr>
<td>else</td>
<td>block</td>
<td></td>
</tr>
<tr>
<td>WNOTRACED</td>
<td>keep waiting for stopped child</td>
<td></td>
</tr>
</tbody>
</table>

*status

<table>
<thead>
<tr>
<th>WIFEXITED(s)</th>
<th>$\neq 0$: child used <code>_exit()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>$= 0$: else</td>
<td></td>
</tr>
<tr>
<td>WEXITSTATUS(s)</td>
<td>returns child exit code</td>
</tr>
<tr>
<td>(via <code>_exit()</code>)</td>
<td></td>
</tr>
<tr>
<td>WIFSIGNALED(s)</td>
<td>$\neq 0$: if child terminated due to signal</td>
</tr>
<tr>
<td>WTERMSIG(s)</td>
<td>returns signal # that terminated the child</td>
</tr>
<tr>
<td>WIFSTOPPED(s)</td>
<td>$\neq 0$: returns value if child stopped due to job control</td>
</tr>
<tr>
<td>WSTOPSIG(s)</td>
<td>returns signal # that stopped the child process</td>
</tr>
</tbody>
</table>
exec() family

#include <unistd.h>

int execl (const char *path, const char *arg,...);
in
int execlp (const char *file, const char *arg,...);
in
int execle (const char *path, const char *arg,...,char * const envp[]);
in
int execv (const char *path, char *const argv[]);
in
int execvp (const char *file, char *const argv[]);
in
int execve (const char *filename,char *const argv[],char *const envp[]);

• These functions swap a new program image in, taking the place of the current process.
exec() family

#include <unistd.h>

int execl (const char *path, const char *arg,...);
int execlp (const char *file, const char *arg,...);
int execlpe (const char *path, const char *arg,...,char * const envp[]);
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *filename,char *const argv[],char *const envp[]);

• These functions swap a new program image in, taking the place of the current
  process.

• The pid is retained.
exec() family

```c
#include <unistd.h>
int execl  (const char *path,       const char *arg,...);
int execlp (const char *file,      const char *arg,...);
int execlpe(const char *path,      const char *arg,...,char * const envp[]);
int execv  (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *filename,char *const argv[],char *const envp[]);
```

- These functions swap a new program image in, taking the place of the current process.
- The pid is retained.
- Often used with fork(), thus the new program image takes the place of a child process’s image.
exec() family, con’t.

path  full pathname of the program
**exec() family, con’t.**

- **path**  full pathname of the program

- **file**  will use PATH if filename doesn’t start with a “/”; it’s the program name to be executed
exec() family, con’t.

**path**  full pathname of the program

**file**  will use PATH if filename doesn’t start with a “/”; its the program name to be executed

**argc,argv**  There are two ways to set up the new image’s argc,argv:
exec() family, con’t.

**path** full pathname of the program

**file** will use PATH if filename doesn’t start with a “/”; its the program name to be executed

**argc,argv** There are two ways to set up the new image’s argc,argv:

- `execl`, `execlp`, `execle`: these take arguments from function call argument list. The last one should be NULL.
exec() family, con’t.

**path**  full pathname of the program

**file**  will use PATH if filename doesn’t start with a “/”; its the program name to be executed

**argc, argv**  There are two ways to set up the new image’s argc, argv:

- **exec**, **execp**, **execle**: these take arguments from function call argument list. The last one should be NULL.
- **execv**, **execvp**, **execve**: these take arguments from a pointer to an array of strings (char **argv). Again, the last “string” should be the NULL pointer.
exec() family, con’t.

**path**  full pathname of the program

**file**  will use PATH if filename doesn’t start with a “/”; it’s the program name to be executed

**argc, argv**  There are two ways to set up the new image’s argc, argv:

  - **execl, execlp, execle**: these take arguments from function call argument list. The last one should be NULL.
  - **execv, execvp, execve**: these take arguments from a pointer to an array of strings (char **argv). Again, the last “string” should be the NULL pointer.

**envp**  (execle, execve) permit passing an array of environment strings.
exec() family, con’t.

**path**  full pathname of the program

**file**  will use PATH if filename doesn’t start with a “/”; its the program name to be executed

**argc,argv**  There are two ways to set up the new image’s argc,argv:

- *excl, execlp, execle:* these take arguments from function call argument list. The last one should be NULL.
- *execv, execvp, execve:* these take arguments from a pointer to an array of strings (char **argv). Again, the last “string” should be the NULL pointer.

**envp**  (execle, execve) permit passing an array of environment strings.

(see exec1.c)
Argc, Argv, Envp Review

```c
int main(int argc, char *argv[], char *envp[])
```

**argc** count of strings contained by args
Argc, Argv, Envp Review

```c
int main(int argc, char *argv[], char *envp[])
```

**argc**  count of strings contained by args

**argv**  an array of strings (with the last one being NULL)
Argc, Argv, Envp Review

int main(int argc, char *argv[], char *envp[])

argc  count of strings contained by args
argv  an array of strings (with the last one being NULL)
envp  an array of strings (with the last one being NULL) holding the environment strings *(in the form envvar=value)*
int main(int argc, char *argv[], char *envp[])

**argc**  count of strings contained by args

**argv**  an array of strings (with the last one being NULL)

**envp**  an array of strings (with the last one being NULL) holding the environment strings (in the form envvar=value)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgmname</td>
<td>arg₁</td>
<td>arg₂</td>
<td>arg₃</td>
<td>...</td>
<td>NULL</td>
</tr>
</tbody>
</table>
int main(int argc, char *argv[], char *envp[])

**argc** count of strings contained by *args

**argv** an array of strings (with the last one being NULL)

**envp** an array of strings (with the last one being NULL) holding the environment strings *(in the form envvar=value)*

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>pgmname</td>
<td>arg₁</td>
<td>arg₂</td>
<td>arg₃</td>
<td>...</td>
</tr>
</tbody>
</table>

So

\[
\begin{align*}
\text{argv}[0] &= \text{“pgmname”} \\
\text{argv}[1] &= \text{“arg1”} \\
\text{argv}[2] &= \text{“arg2”} \\
... &= ...
\end{align*}
\]

\[
\text{argv}[n] = \text{NULL}
\]
Argc, Argv, Envp Review

int main(int argc, char *argv[], char *envp[])

argc  count of strings contained by args
argv  an array of strings (with the last one being NULL)
envp  an array of strings (with the last one being NULL) holding the environment
strings (in the form envvar=value)

0  1  2  3  n
pgmname  arg₁  arg₂  arg₃  ...  NULL

So

argv[0] = “pgmname”
argv[1] = “arg₁”
... = ...
argv[n] = NULL

List of args : arg₀, arg₁, ..., argₙ, NULL
Vector of args : argv, where argv[0] = “pgm”, argv[1] = “arg₁”, ...
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
Child process may have different effective UID, GID if exec’d program has associated flag bits set.
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
Child process may have different effective UID, GID if exec’d program has associated flag bits set.
Has standard default signal handling. (ie. does not inherit signal masks, etc)
exec() family, con’t.

Upon success:

- Child process has new text, data, and stack segments.
- Child process keeps file descriptor table except close-on-exec flagged file descriptors.
- Child process may have different effective UID, GID if exec’d program has associated flag bits set.
- Has standard default signal handling. (ie. does not inherit signal masks, etc)

On errors:
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
Child process may have different effective UID, GID if exec’d program has associated flag bits set.
Has standard default signal handling. (ie. does not inherit signal masks, etc)

On errors:

BSD does an automatic sleep and retry if ETXTBSY encountered.
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
Child process may have different effective UID, GID if exec’d program has associated flag bits set.
Has standard default signal handling. (ie. does not inherit signal masks, etc)

On errors:

BSD does an automatic sleep and retry if ETXTBSY encountered.
Linux treats it as a hard error and returns immediately.
exec() family, con’t.

Upon success:

Child process has new text, data, and stack segments.
Child process keeps file descriptor table except close-on-exec flagged file descriptors.
Child process may have different effective UID, GID if exec’d program has associated flag bits set.
Has standard default signal handling. (ie. does not inherit signal masks, etc)

On errors:

BSD does an automatic sleep and retry if ETXTBSY encountered.
Linux treats it as a hard error and returns immediately.
Traditionally, execle() and execvp() ignore most errors (except EACCES, ENOEXEC, ENOMEM, E2BIG)
**exec() family, con’t.**

Upon success:

- Child process has new text, data, and stack segments.
- Child process keeps file descriptor table *except close-on-exec flagged file descriptors*.
- Child process may have different effective UID, GID if *exec’d program has associated flag bits set*.
- Has standard default signal handling. *(ie. does not inherit signal masks, etc)*

On errors:

- BSD does an automatic sleep and retry if `ETXTBSY` encountered.
- Linux treats it as a hard error and returns immediately.
- Traditionally, `execlp()` and `execvp()` ignore most errors (except `EACCES`, `ENOEXEC`, `ENOMEM`, `E2BIG`)
- Linux uses the “current directory first” default path to find program.
#include <stdlib.h>

int system(const char *command);

- The `system()` call executes the command by calling `/bin/sh -c command`. 
#include <stdlib.h>

int system(const char *command);

• The system() call executes the command by calling /bin/sh -c command.

• Uses fork(), exec(), waitpid(), and /bin/sh to execute the command.
#system()

```c
#include <stdlib.h>

int system(const char *command);
```

- The `system()` call executes the command by calling `/bin/sh -c command`.
- Uses `fork()`, `exec()`, `waitpid()`, and `/bin/sh` to execute the command.
- The `waitpid()` is used to wait for the `system()`’d process, not just any process.
system()

```
#include <stdlib.h>

int system(const char *command);
```

- The `system()` call executes the command by calling `/bin/sh -c command`.
- Uses `fork()`, `exec()`, `waitpid()`, and `/bin/sh` to execute the command.
- The `waitpid()` is used to wait for the `system()`'d process, not just any process.
- Returns -1 if return PID matches child pid that it forked.
system()

#include <stdlib.h>

int system(const char *command);

• The system() call executes the command by calling /bin/sh -c command.

• Uses fork(), exec(), waitpid(), and /bin/sh to execute the command.

• The waitpid() is used to wait for the system()’d process, not just any process.

• Returns -1 if return PID matches child pid that it forked.

• _exit() is used to terminate the child process.
Races

- `fork()`s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc)
Races

- `fork()`s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc).
- One cannot predict whether the parent or child process will run first after the `fork()`.
Races

• fork()s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc)

• One cannot predict whether the parent or child process will run first after the fork().

• Several ways to coordinate access: (few of which have we gotten to as yet)
Races

- `fork()`s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc).
- One cannot predict whether the parent or child process will run first after the `fork()`.
- Several ways to coordinate access: (few of which have we gotten to as yet)
  - signals
Races

- fork()s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc)

- One cannot predict whether the parent or child process will run first after the fork().

- Several ways to coordinate access: (few of which have we gotten to as yet)
  - signals
  - semaphores, mutexes
Races

• fork()s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc)

• One cannot predict whether the parent or child process will run first after the fork().

• Several ways to coordinate access: (few of which have we gotten to as yet)
  • signals
  • semaphores, mutexes
  • file locks
Races

- fork()s can easily cause races – multiple processes trying to do something with shared resources simultaneously (memory, i/o channels, graphics, etc)
- One cannot predict whether the parent or child process will run first after the fork().
- Several ways to coordinate access: (few of which have we gotten to as yet)
  - signals
  - semaphores, mutexes
  - file locks
  - polling. A crude way would be to poll to see if the parent process has terminated: while(getppid() != -1) sleep(1); (polling wastes cpu)
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe.
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels. (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
#include <unistd.h>

int pipe(int pipefd[2]);

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (a stream). No `seek()`s permitted on the “file”
#include <unistd.h>

int pipe(int pipefd[2]);

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (a stream). No `lseek()`s permitted on the “file”
- PIPE_BUF bytes can be buffered in a pipe
Pipes

#include <unistd.h>

int pipe(int pipefd[2]);

• The `pipe()` function opens one pipe, yielding two file descriptors.
• `pipefd[0]` is the read end of the pipe
• `pipefd[1]` is the write end of the pipe
• Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
• Pipes can be used only between processes with a common ancestor.
• Pipes are first-in, first-out (a stream). No `lseek()`s permitted on the “file”
• PIPE_BUF bytes can be buffered in a pipe
• There must be a process that has opened the pipe for reading `prior` to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
Pipes

#include <unistd.h>

int pipe(int pipefd[2]);

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (a stream). No `lseek()`s permitted on the “file”
- `PIPE_BUF` bytes can be buffered in a pipe
- There must be a process that has opened the pipe for reading prior to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
- If communication is truly one way (say, parent → child) then the process may close the other pipe.
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (**AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable**).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (**a stream**). No `lseek()`s permitted on the “file”
- PIPE_BUF bytes can be buffered in a pipe
- There must be a process that has opened the pipe for reading **prior** to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
- If communication is truly one way (**say, parent → child**) then the process may close the other pipe.
- **Warning**: when multiple child processes have access to a pipe, it is easy to get interleaved messages → chaos!
Pipes

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

• The pipe() function opens one pipe, yielding two file descriptors.
• pipefd[0] is the read end of the pipe
• pipefd[1] is the write end of the pipe
• Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
• Pipes can be used only between processes with a common ancestor.
• Pipes are first-in, first-out (a stream). No lseek()s permitted on the “file”
• PIPE_BUF bytes can be buffered in a pipe
• There must be a process that has opened the pipe for reading prior to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
• If communication is truly one way (say, parent → child) then the process may close the other pipe.
• Warning: when multiple child processes have access to a pipe, it is easy to get interleaved messages → chaos!
• Returns 0:success, -1:failure
Pipes

\#include <unistd.h>

int pipe(int pipefd[2]);

- The pipe() function opens one pipe, yielding two file descriptors.
- pipefd[0] is the read end of the pipe
- pipefd[1] is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (a stream). No lseek()s permitted on the “file”
- PIPE_BUF bytes can be buffered in a pipe
- There must be a process that has opened the pipe for reading prior to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
- If communication is truly one way (say, parent \rightarrow child) then the process may close the other pipe.
- Warning: when multiple child processes have access to a pipe, it is easy to get interleaved messages \rightarrow chaos!
- Returns 0:success, -1:failure
  EFAULT: pipefd is not valid
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (*AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable*).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (*a stream*). No `lseek()`s permitted on the “file”
- `PIPE_BUF` bytes can be buffered in a pipe
- There must be a process that has opened the pipe for reading *prior* to attempts to write to it. The process will receive a `SIGPIPE` signal otherwise.
- If communication is truly one way (*say, parent → child*) then the process may close the other pipe.
- *Warning*: when multiple child processes have access to a pipe, it is easy to get interleaved messages → chaos!
- Returns 0: success, -1: failure
  - `EFAULT`: `pipefd` is not valid
  - `EMFILE`: too many file descriptors are in use by the process
Pipes

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

- The `pipe()` function opens one pipe, yielding two file descriptors.
- `pipefd[0]` is the read end of the pipe
- `pipefd[1]` is the write end of the pipe
- Pipes are unidirectional communication channels (AT&T Unix Sys V.4 has bidirectional pipes, but they aren’t portable).
- Pipes can be used only between processes with a common ancestor.
- Pipes are first-in, first-out (a stream). No `lseek()`s permitted on the “file”
- `PIPE_BUF` bytes can be buffered in a pipe
- There must be a process that has opened the pipe for reading *prior* to attempts to write to it. The process will receive a SIGPIPE signal otherwise.
- If communication is truly one way (say, *parent* → *child*) then the process may close the other pipe.
- *Warning*: when multiple child processes have access to a pipe, it is easy to get interleaved messages → chaos!
- Returns 0: success, -1: failure
  - EFAULT: `pipefd` is not valid
  - EMFILE: too many file descriptors are in use by the process
  - ENFILE: the system limit on the total number of open files has been reached
I/O Redirection

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

• This is how the shells implement i/o redirection (ie. sort <inputfile >outputfile)
I/O Redirection

#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

- This is how the shells implement i/o redirection (ie. sort <inputfile >outputfile)
- A copy is made of the file descriptor oldfd
I/O Redirection

```c
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- This is how the shells implement i/o redirection (ie. `sort <inputfile >outputfile`)
- A copy is made of the file descriptor `oldfd`
- `dup()` returns the lowest-numbered unused descriptor for the new descriptor
# I/O Redirection

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- This is how the shells implement i/o redirection (i.e., `sort <inputfile >outputfile`)
- A copy is made of the file descriptor `oldfd`
- `dup()` returns the lowest-numbered unused descriptor for the new descriptor
- `dup2()` makes `newfd` be a copy of `oldfd`. This is useful for i/o redirection:
I/O Redirection

```c
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);

• This is how the shells implement i/o redirection (i.e. sort <inputfile >outputfile)

• A copy is made of the file descriptor `oldfd`

• `dup()` returns the lowest-numbered unused descriptor for the new descriptor

• `dup2()` makes `newfd` be a copy of `oldfd`. This is useful for i/o redirection:
  ```
  dup2(ofd,STDIN_FILENO) // makes stdin come from ofd
  ```
```
# I/O Redirection

```c
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- This is how the shells implement i/o redirection (i.e. `sort <inputfile >outputfile`)

- A copy is made of the file descriptor `oldfd`

- `dup()` returns the lowest-numbered unused descriptor for the new descriptor

- `dup2()` makes `newfd` be a copy of `oldfd`. This is useful for i/o redirection:
  ```c
dup2(ofd,STDIN_FILENO) // makes stdin come from ofd
dup2(ofd,STDOUT_FILENO) // makes stdout go to ofd
```
I/O Redirection

```c
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- This is how the shells implement i/o redirection (i.e. `sort <inputfile >outputfile`)
- A copy is made of the file descriptor `oldfd`
- `dup()` returns the lowest-numbered unused descriptor for the new descriptor
- `dup2()` makes `newfd` be a copy of `oldfd`. This is useful for i/o redirection:
  - `dup2(ofd,STDIN_FILENO)` // makes stdin come from ofd
  - `dup2(ofd,STDOUT_FILENO)` // makes stdout go to ofd
  - `dup2(ofd,STDERR_FILENO)` // makes stderr go to ofd
I/O Redirection, con’t.

• `dup2()` is an “atomic” call, although it otherwise appears to act like
I/O Redirection, con’t.

• `dup2()` is an “atomic” call, although it otherwise appears to act like
  `close(nfd);`
  `fcntl(ofd,F_DUPFD,nfd);`
I/O Redirection, con’t.

- `dup2()` is an “atomic” call, although it otherwise appears to act like `close(nfd);` and `fcntl(ofd,F_DUPFD,nfd);`

A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but not with `dup2()`.
I/O Redirection, con’t.

- `dup2()` is an “atomic” call, although it otherwise appears to act like
  ```c
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  ```
  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but
  not with `dup2()`.

  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to
  the rest of the system as if it occurred as one instruction.
I/O Redirection, con’t.

• **dup2()** is an “atomic” call, although it otherwise appears to act like
  
  ```c
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  ```
  
  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but not with `dup2()`.

  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to the rest of the system as if it occurred as one instruction.

• **Command Pipes**: One can couple `pipe()` with `dup2()` so that the stdout of a parent process is connected to the stdin of the child process
I/O Redirection, con’t.

• **dup2()** is an “atomic” call, although it otherwise appears to act like
  
  ```c
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  ```

  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but not with `dup2()`.

  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to the rest of the system as if it occurred as one instruction.

• **Command Pipes**: One can couple `pipe()` with `dup2()` so that the stdout of a parent process is connected to the stdin of the child process

• Typical reasons for failures include:
I/O Redirection, con’t.

• `dup2()` is an “atomic” call, although it otherwise appears to act like
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but
  not with `dup2()`.
  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to
  the rest of the system as if it occurred as one instruction.

• **Command Pipes**: One can couple `pipe()` with `dup2()` so that the stdout of a parent
  process is connected to the stdin of the child process

• Typical reasons for failures include:
  **EBADF** `oldfd` isn’t an open file descriptor, or `newfd` is out of the allowed range for
  file descriptors
I/O Redirection, con’t.

• **dup2()** is an “atomic” call, although it otherwise appears to act like
  
  ```
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  ```

  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but not with `dup2()`.

  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to the rest of the system as if it occurred as one instruction.

• **Command Pipes**: One can couple `pipe()` with `dup2()` so that the stdout of a parent process is connected to the stdin of the child process

• Typical reasons for failures include:

  **EBADF**  `oldfd` isn’t an open file descriptor, or `newfd` is out of the allowed range for file descriptors

  **EBUSY** can occur in race conditions against `open()` and `dup()`
I/O Redirection, con’t.

• **dup2()** is an “atomic” call, although it otherwise appears to act like:
  ```c
  close(nfd);
  fcntl(ofd,F_DUPFD,nfd);
  ```
  A signal could potentially interrupt the duplication between `close()` and `fcntl()`, but not with `dup2()`.

  **atomic**: a set of operations is *atomic* if it cannot be interrupted; ie. it appears to the rest of the system as if it occurred as one instruction.

• **Command Pipes**: One can couple `pipe()` with `dup2()` so that the stdout of a parent process is connected to the stdin of the child process

• Typical reasons for failures include:
  - **EBADF** `oldfd` isn’t an open file descriptor, or `newfd` is out of the allowed range for file descriptors
  - **EBUSY** can occur in race conditions against `open()` and `dup()`
  - **EINTR** `dup2()` was interrupted by a signal
#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);

int pclose(FILE *stream);

shellcmd is a string that the shell may execute
popen() and pclose()

#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);

int pclose(FILE *stream);

shellcmd is a string that the shell may execute

mode “r”=read, “w”=write
#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);

int pclose(FILE *stream);

- shellcmd is a string that the shell may execute
- mode “r”=read, “w”=write

- The `popen()` call opens a process by creating a pipe, forking, and invoking the shell.
popen() and pclosen()

#include <stdio.h>
FILE *popen(const char *shellcmd, const char *mode);
int pclosen(FILE *stream);

- shellcmd is a string that the shell may execute
- mode “r”=read, “w”=write

- The popen() call opens a process by creating a pipe, forking, and invoking the shell.
- Equivalent to:
popen() and pclose()

#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);
int pclose(FILE *stream);

shellcmd is a string that the shell may execute

mode “r”=read, “w”=write

• The popen() call opens a process by creating a pipe, forking, and invoking the shell.

• Equivalent to:
  parent: fork, pipe and dup2, fdopen (yielding a FILE*)
  child : exec()s /bin/sh
#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);
int pclose(FILE *stream);

- **shellcmd** is a string that the shell may execute
- **mode** “r”=read, “w”=write

- The **popen()** call opens a process by creating a pipe, forking, and invoking the shell.
- Equivalent to:
  - parent: fork, pipe and dup2, fdopen (yielding a FILE*)
  - child: exec()s /bin/sh
- The shell command’s output is sent to the FILE * stream, which may be read as usual with streams (fscanf, fread, fgets, etc)
popen() and pclose()

#include <stdio.h>

FILE *popen(const char *shellcmd, const char *mode);

int pclose(FILE *stream);

shellcmd is a string that the shell may execute

mode “r”=read, “w”=write

• The popen() call opens a process by creating a pipe, forking, and invoking the shell.

• Equivalent to:
  parent: fork, pipe and dup2, fdopen (yielding a FILE*)
  child : exec()s /bin/sh

• The shell command’s output is sent to the FILE * stream, which may be read as usual with streams (fscanf, fread, fgets, etc)

• pclose() returns the exit status of the shell command that was executed (-1 on failure)
popen() and pclose()

#include <stdio.h>
FILE *popen(const char *shellcmd, const char *mode);
int pclose(FILE *stream);

shellcmd is a string that the shell may execute

mode “r”=read, “w”=write

• The popen() call opens a process by creating a pipe, forking, and invoking the shell.
• Equivalent to:
  parent: fork, pipe and dup2, fdopen (yielding a FILE*)
  child : exec()s /bin/sh
• The shell command’s output is sent to the FILE * stream, which may be read as usual with streams (fscanf, fread, fgets, etc)
• pclose() returns the exit status of the shell command that was executed (-1 on failure)
• Pipes are unidirectional! The type argument may specify only reading or writing, not both.