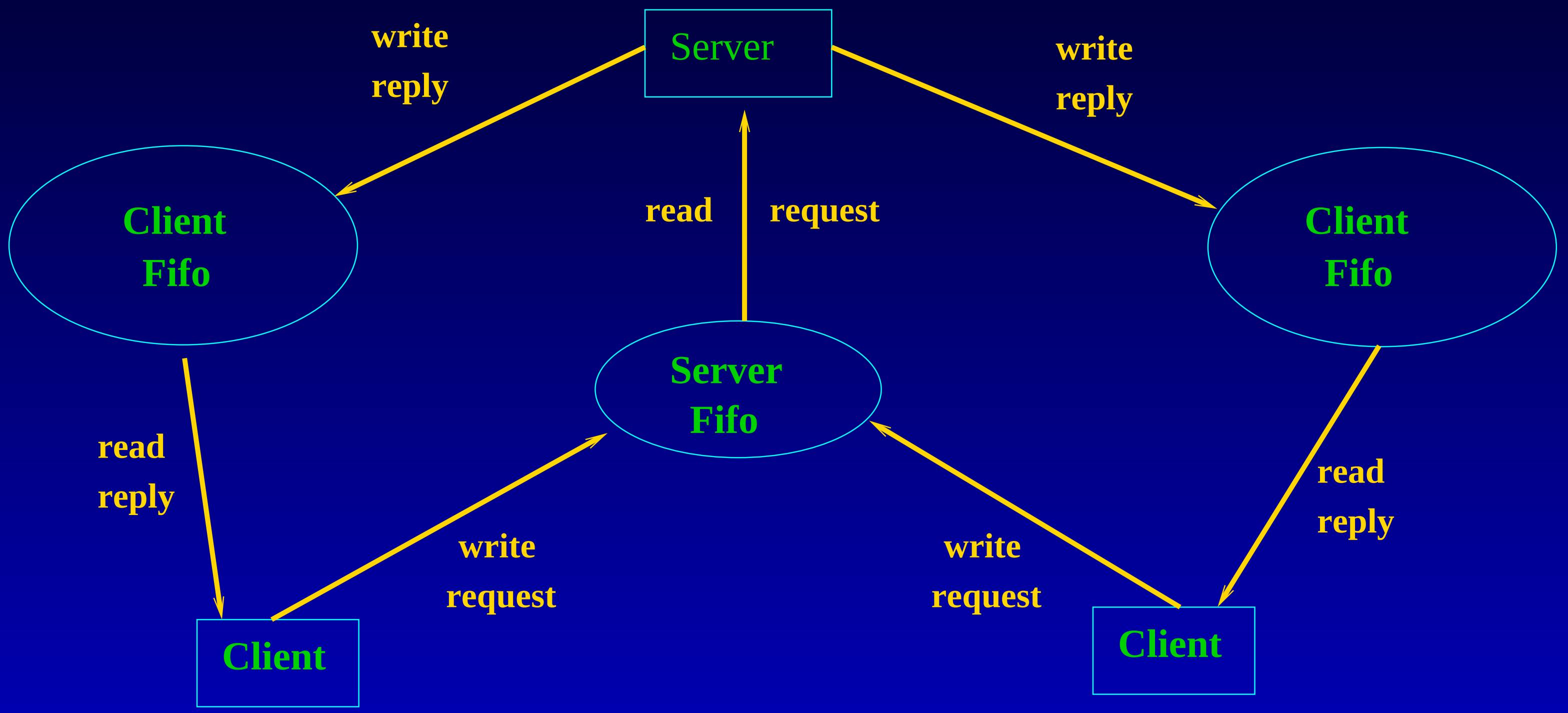
Client-Server Communications

with FIFOS

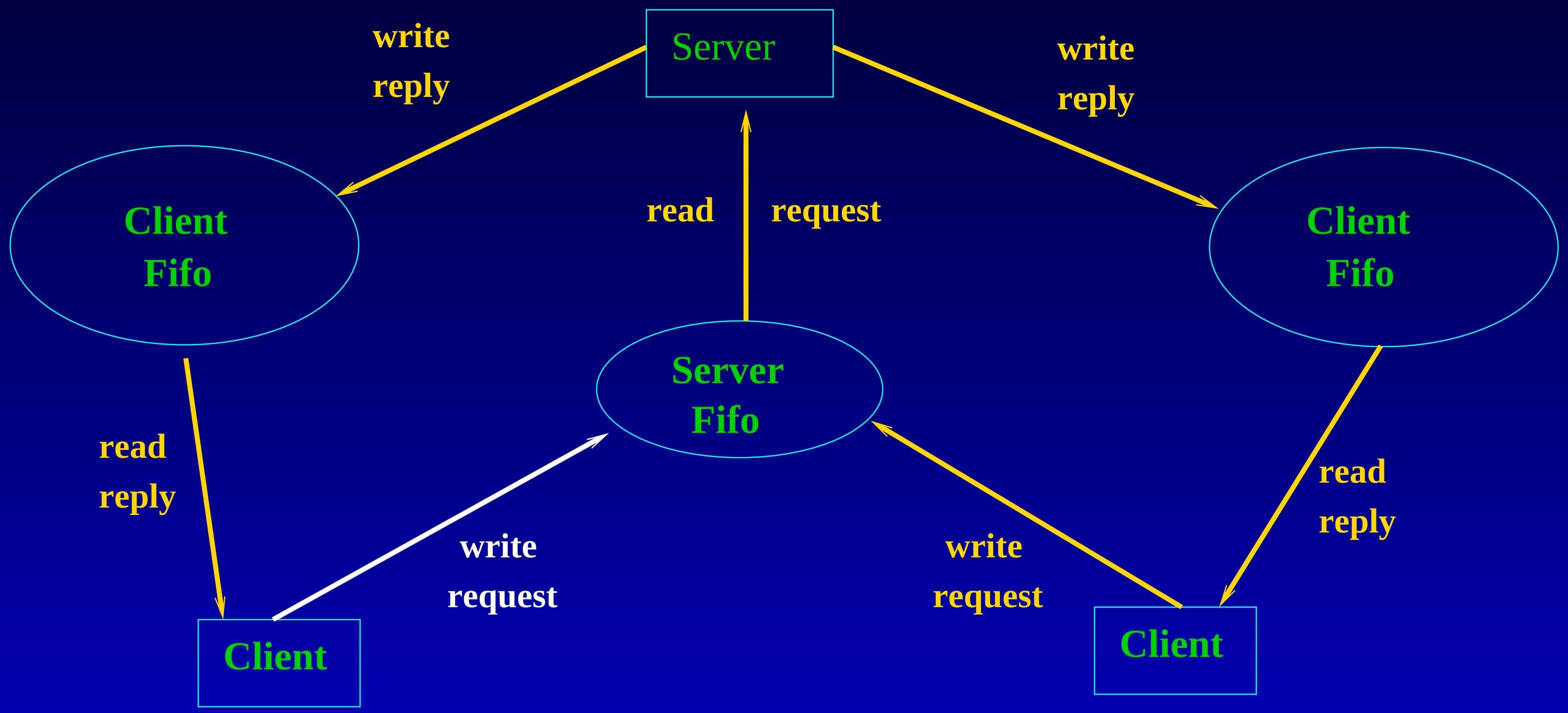


A communications net with one server and multiple clients typically involves a server FIFO and (multiple) client FIFOs, one per client.

To send a packet of information:

Client-Server Communications

with FIFOS



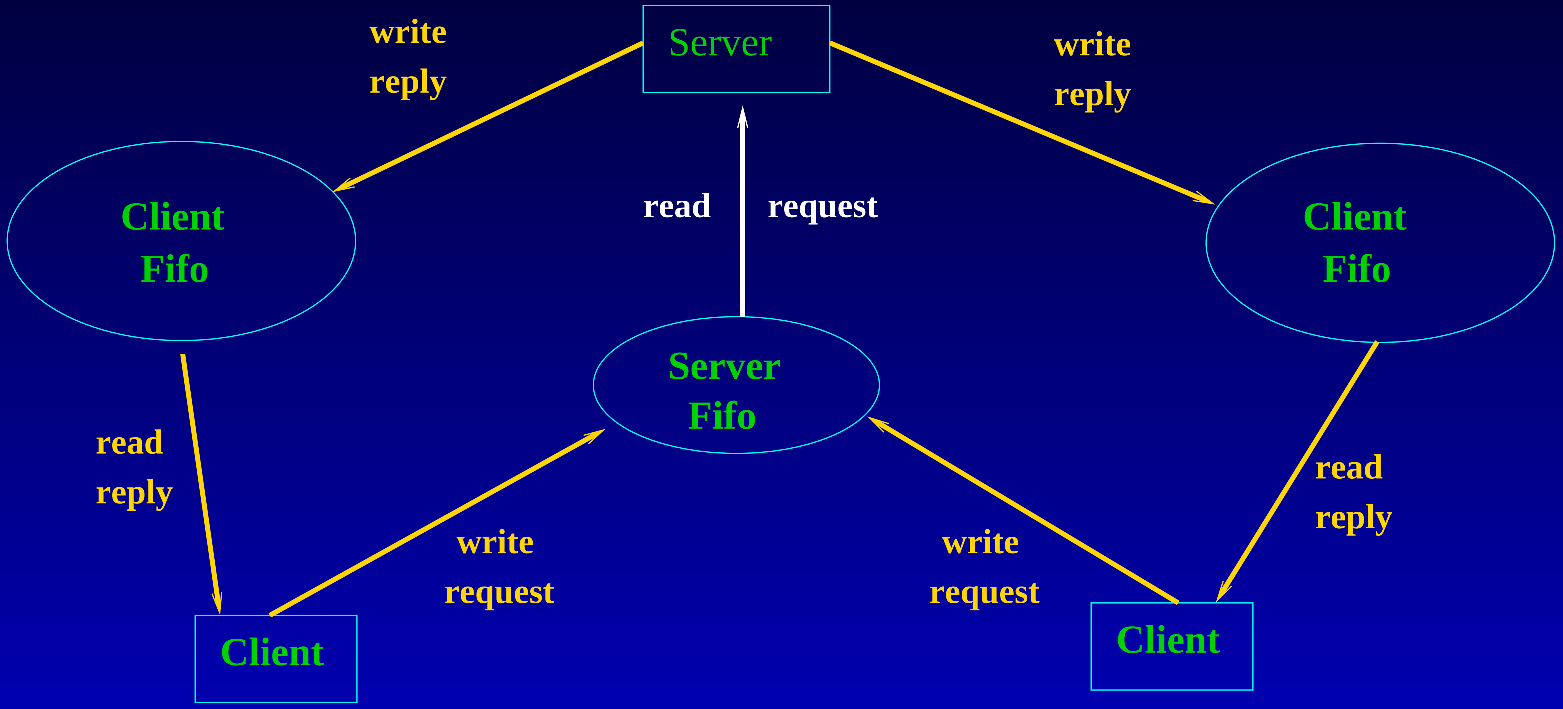
A communications net with one server and multiple clients typically involves a server FIFO and (multiple) client FIFOs, one per client.

To send a packet of information:

Client writes to server FIFO: client pid, qty bytes, data

Client-Server Communications

with FIFOS

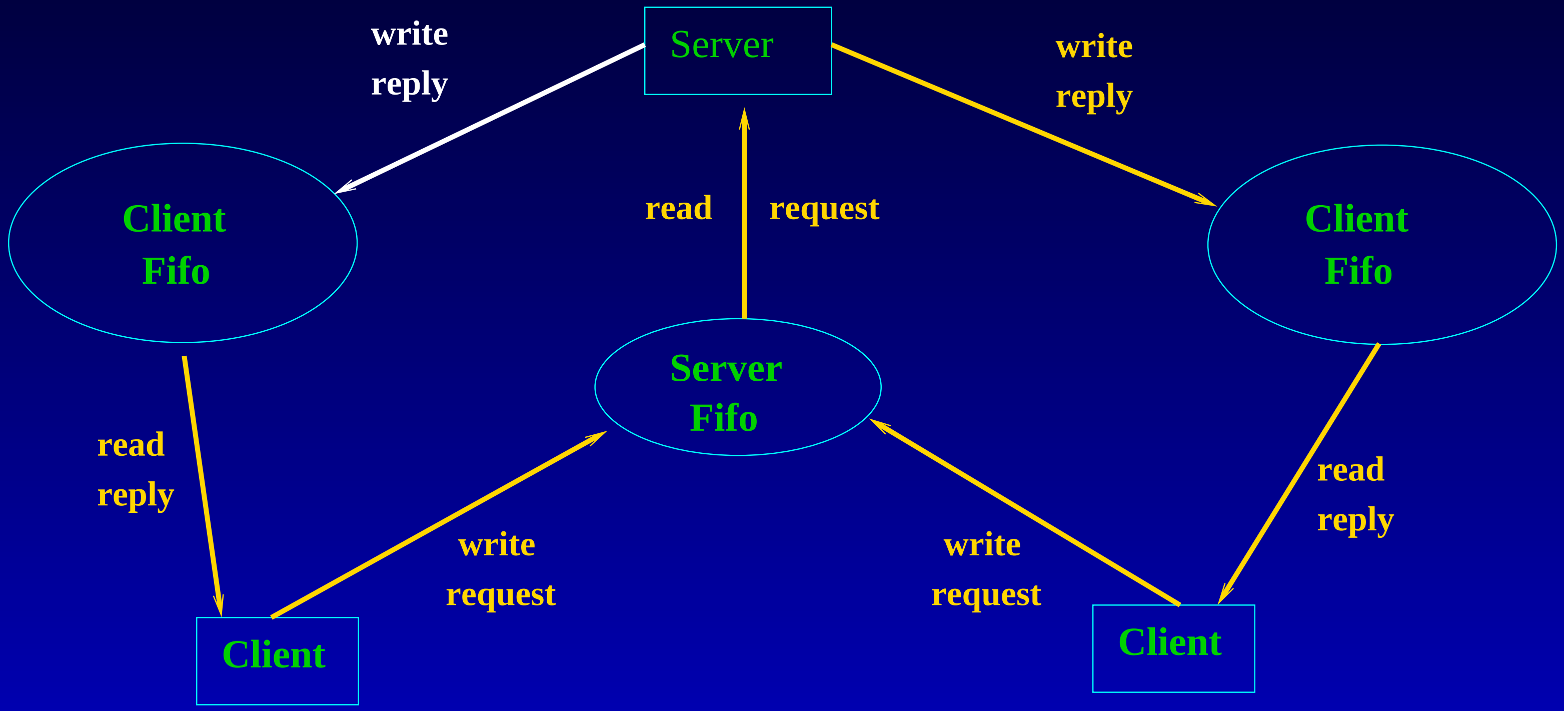


A communications net with one server and multiple clients typically involves a server FIFO and (multiple) client FIFOs, one per client.

To send a packet of information:

Server reads from server’s FIFO

Client-Server Communications *with FIFOS*



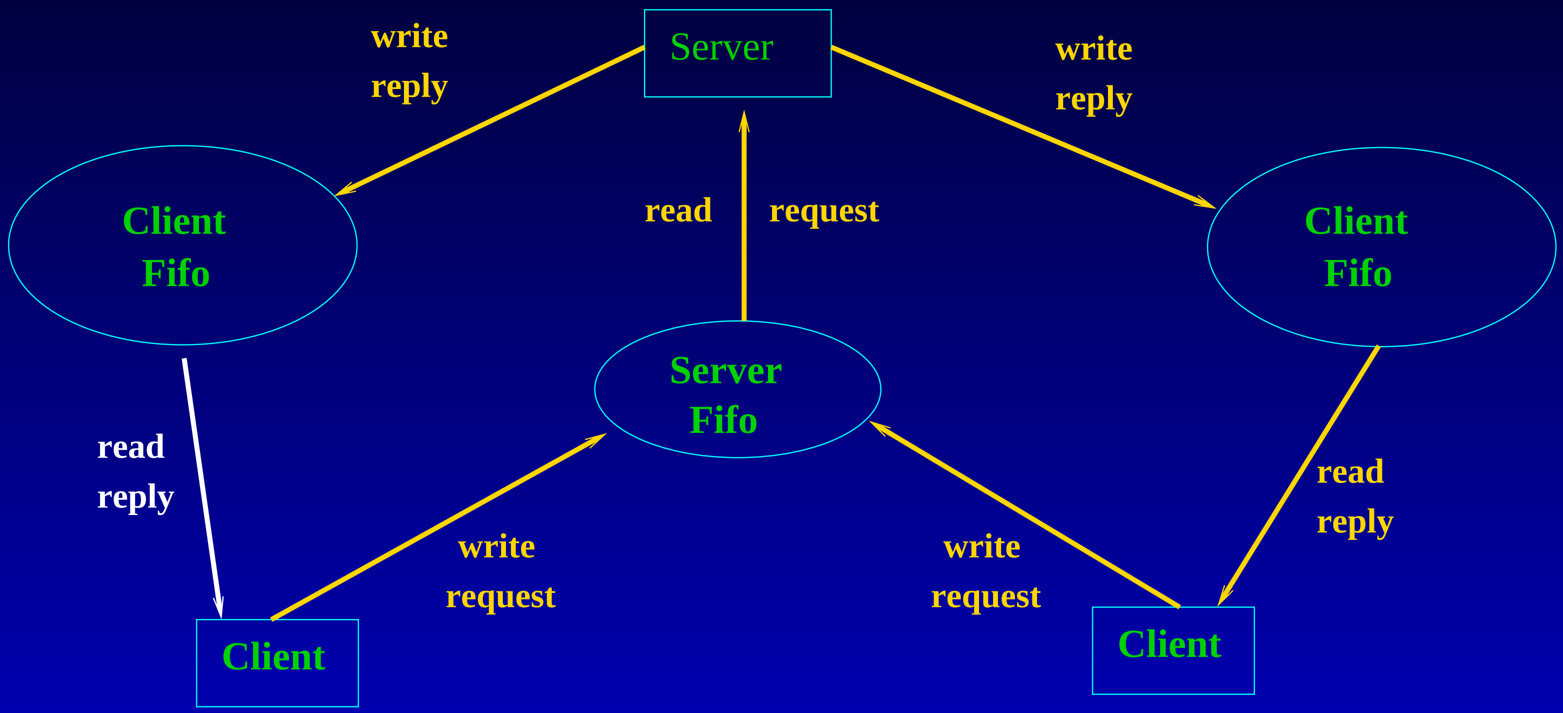
A communications net with one server and multiple clients typically involves a server FIFO and (multiple) client FIFOs, one per client.

To send a packet of information:

Server writes to client's FIFO

Client-Server Communications

with FIFOS



A communications net with one server and multiple clients typically involves a server FIFO and (multiple) client FIFOs, one per client.

To send a packet of information:

Client reads from client’s FIFO

Symbolic Link API

```
#include <unistd.h>
int symlink(const char *reallink, const char *fakelink);
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
int lstat(const char *path, struct stat *buf);
```

symlink creates a symbolic link named fakelink which links to the file named by reallink. A symbolic link may point to an existing file or even a nonexistent one. In the latter case, its called a **dangling link**.

Symbolic Link API

```
#include <unistd.h>
int symlink(const char *reallink, const char *fakelink);
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
int lstat(const char *path, struct stat *buf);
```

symlink creates a symbolic link named fakelink which links to the file named by reallink. A symbolic link may point to an existing file or even a nonexistent one. In the latter case, its called a **dangling link**.

readlink original (real or symbolic) link contents read into buffer buf, which is presumed to have the size bufsiz. This function does not append a null byte to buf. It will truncate the contents to bufsiz as necessary.

Symbolic Link API

```
#include <unistd.h>
int symlink(const char *reallink, const char *fakelink);
ssize_t readlink(const char *path, char *buf, size_t bufsiz);
int lstat(const char *path, struct stat *buf);
```

symlink creates a symbolic link named fakelink which links to the file named by reallink. A symbolic link may point to an existing file or even a nonexistent one. In the latter case, its called a **dangling link**.

readlink original (real or symbolic) link contents read into buffer buf, which is presumed to have the size bufsiz. This function does not append a null byte to buf. It will truncate the contents to bufsiz as necessary.

lstat query file atttributes of the link file itself (as opposed to what its pointing to). Otherwise, its just like the stat() function.

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

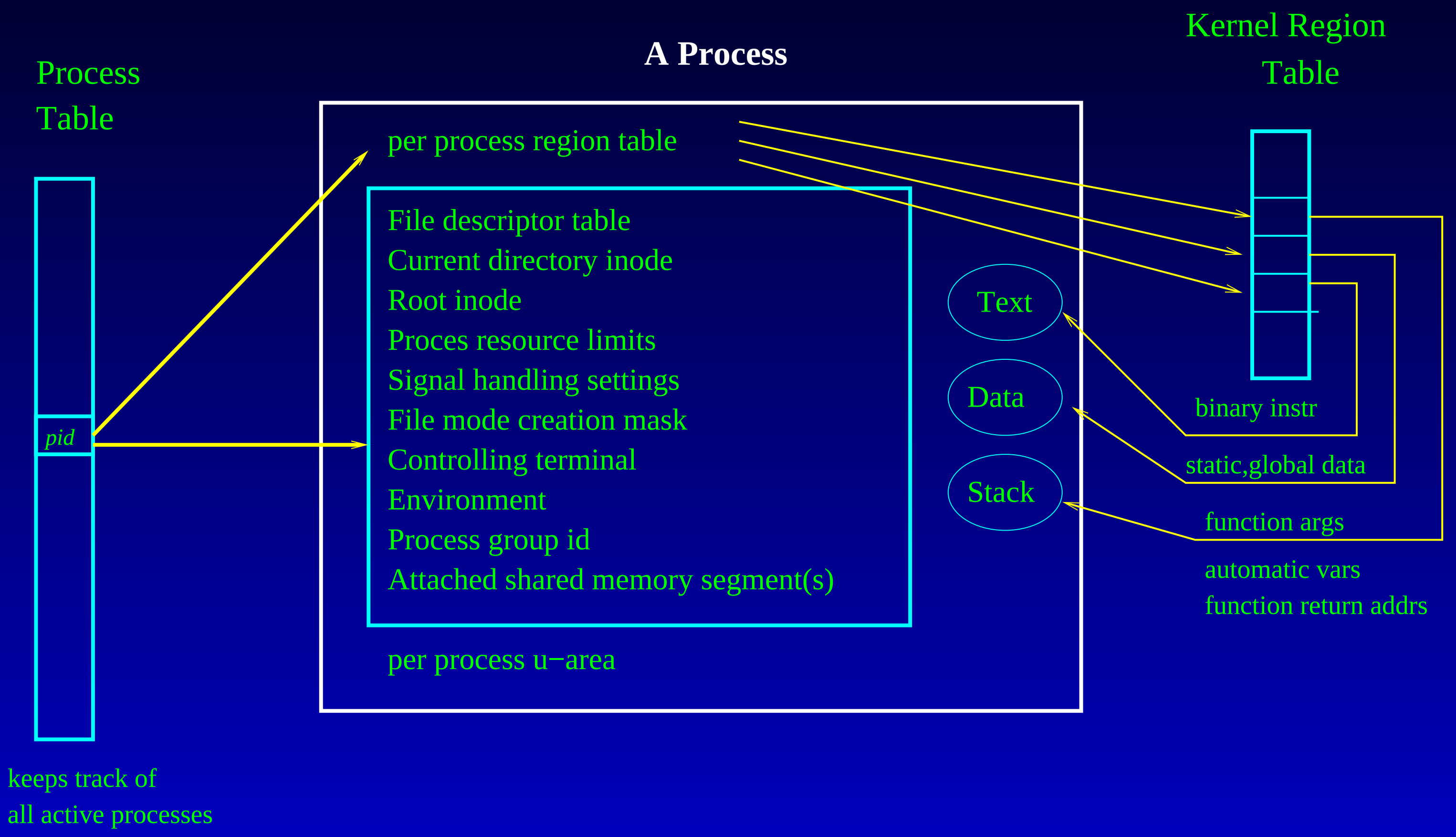
Unix Processes

- 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

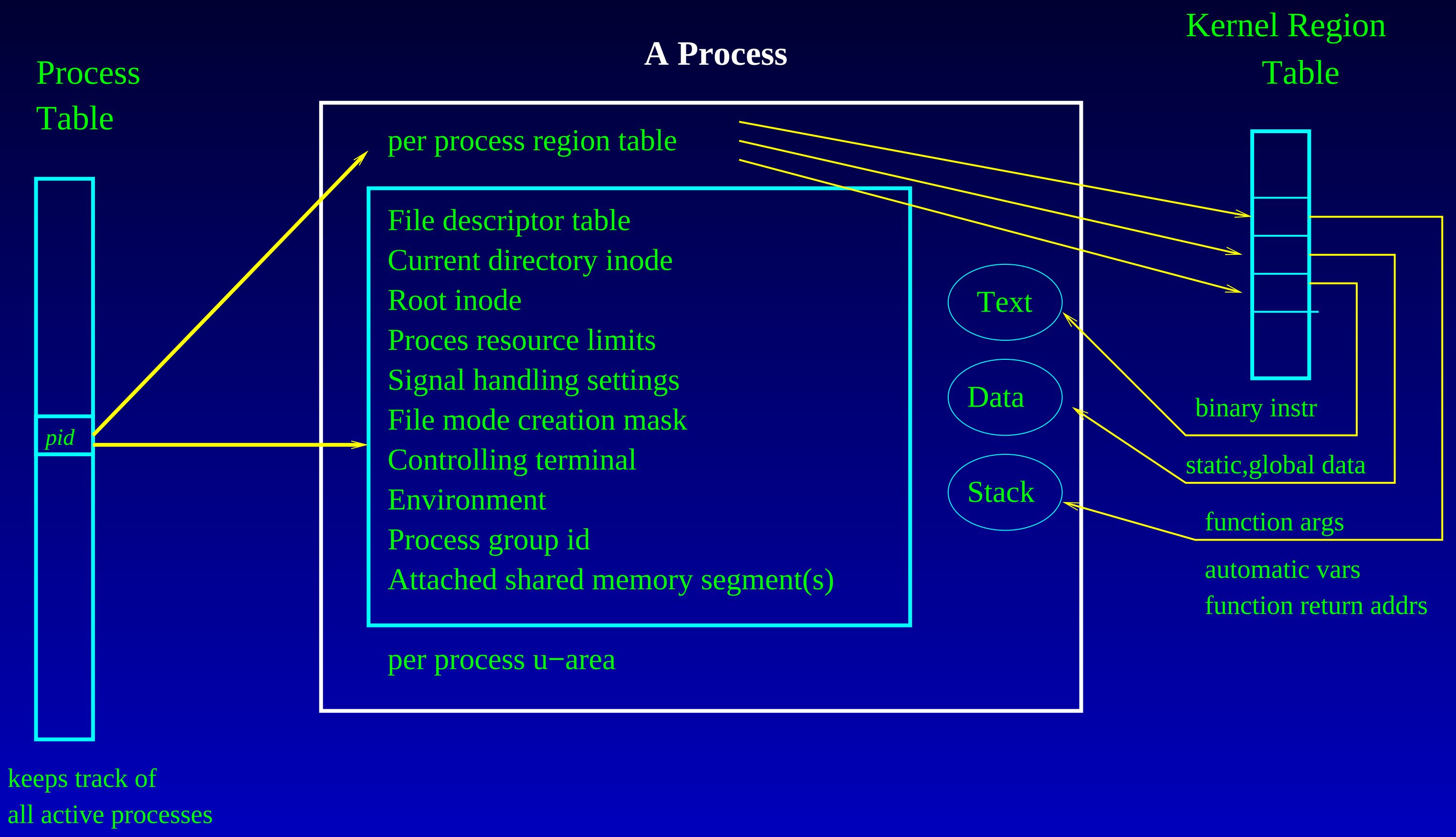
- A **process** is an executing program (task); an execution *context*
- Unix allows users and their programs to run multiple other programs concurrently (ie. multi-tasking)
- Child processes have their own virtual address space (virtual memory)
- Child processes may execute another program in their stead
- Executable files are programs held in storage
- Executable files are *binary* files, created via compilation and linking into *machine code*
- Processes are assigned **pid**s which are unique integer identifiers over the lifetime of the process

Unix Processes



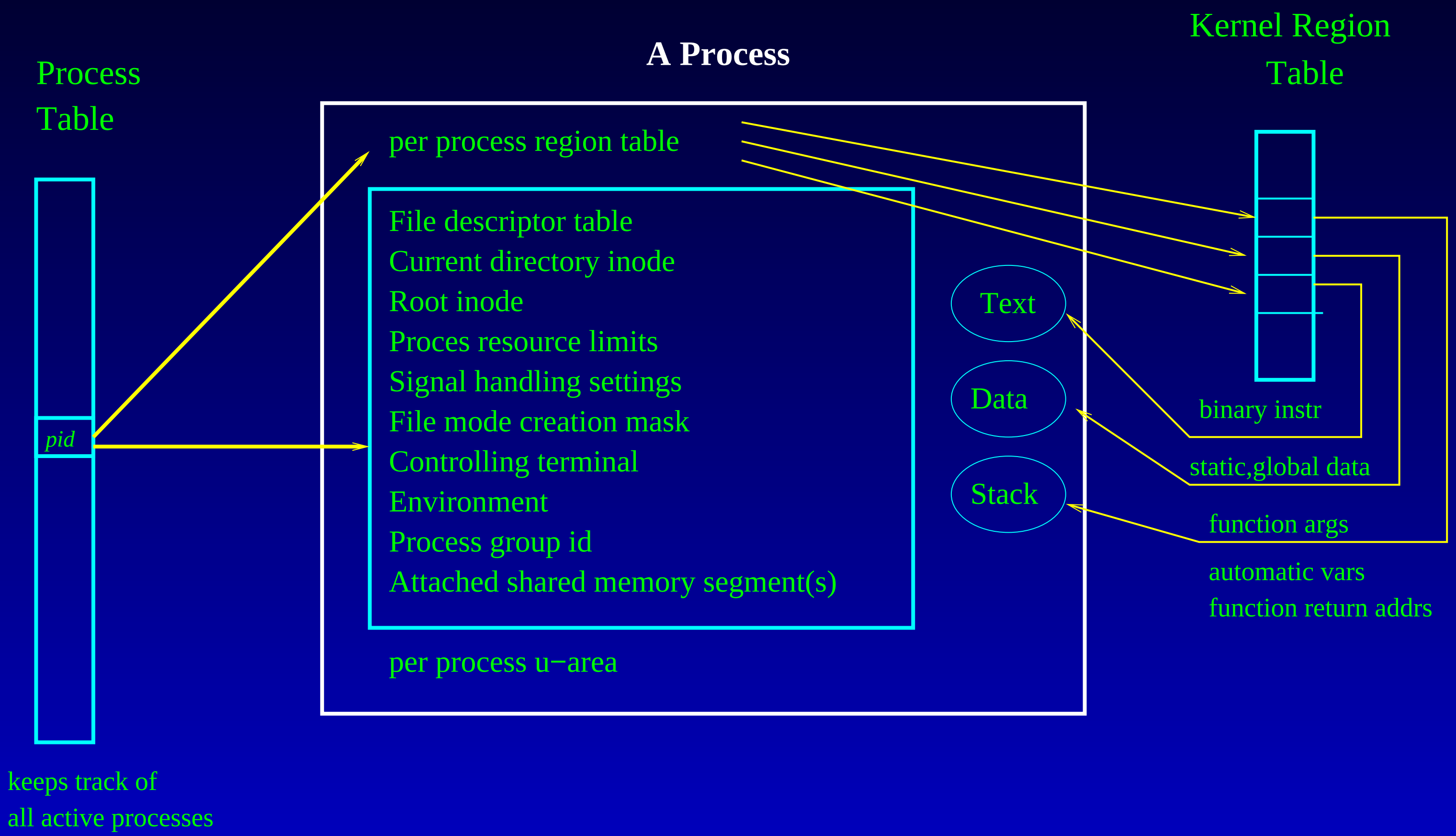
- Processes are all given unique identifier numbers (non-negative).

Unix Processes



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

Unix Processes



A Process

Process Table

pid

keeps track of
all active processes

per process region table

File descriptor table
Current directory inode
Root inode
Proces resource limits
Signal handling settings
File mode creation mask
Controlling terminal
Environment
Process group id
Attached shared memory segment(s)

per process u−area

Text

Data

Stack

Kernel Region Table

binary instr

static,global data

function args
automatic vars
function return addrs

- 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.

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)* :

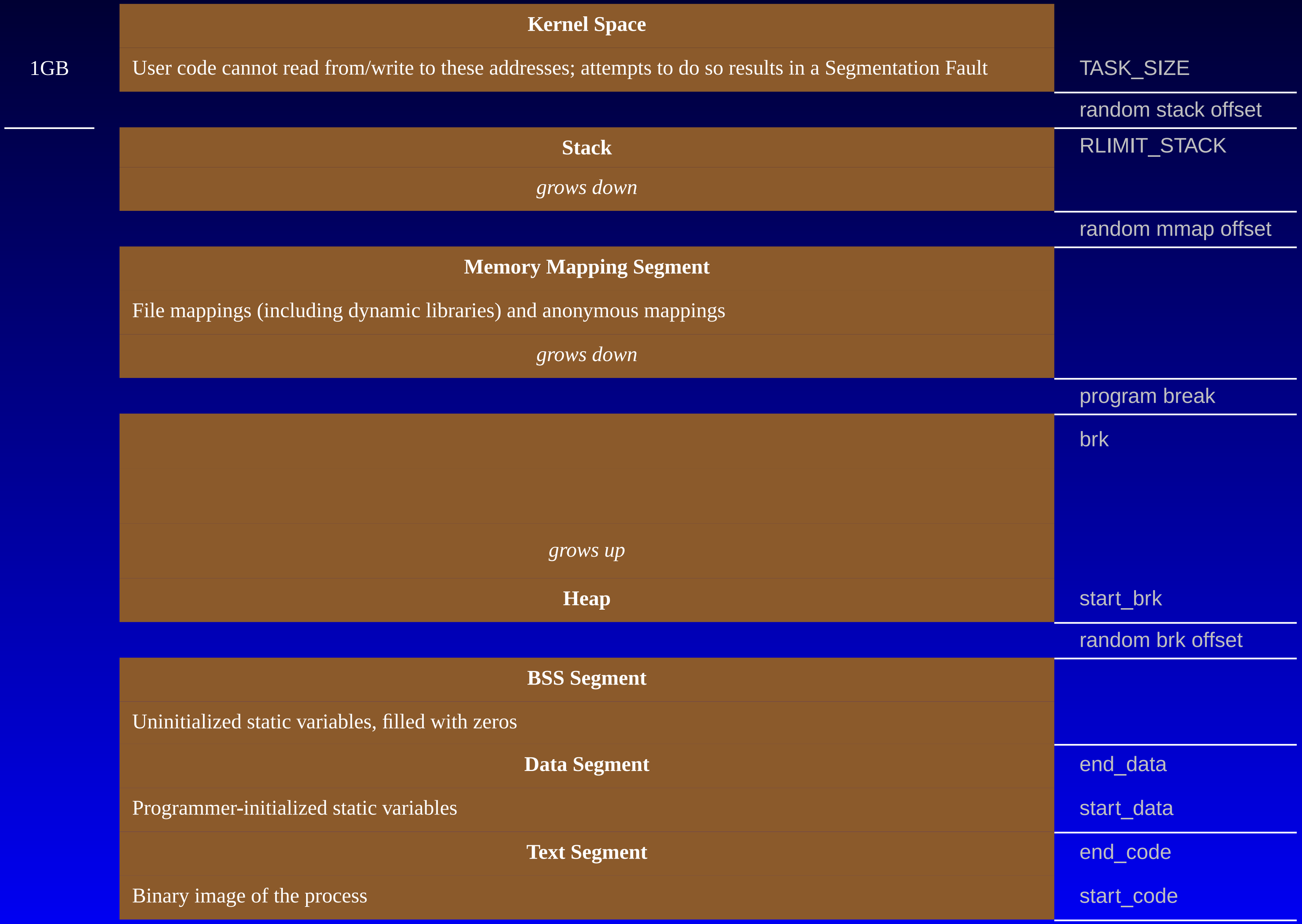
  text the "text" segment contains the executable program's instructions *(code)* and constant data. Unix marks this segment as read-only; the process cannot modify it. Processes may share the text segment with other processes *(ie. threads)*

  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.

Standard Segment Layout



1GB

Kernel Space

User code cannot read from/write to these addresses; attempts to do so results in a Segmentation Fault

TASK_SIZE

random stack offset

Stack

grows down

RLIMIT_STACK

random mmap offset

Memory Mapping Segment

File mappings (including dynamic libraries) and anonymous mappings

grows down

program break

brk

grows up

Heap

start_brk

random brk offset

BSS Segment

Uninitialized static variables, filled with zeros

Data Segment

end_data

Programmer-initialized static variables

start_data

Text Segment

end_code

Binary image of the process

start_code

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

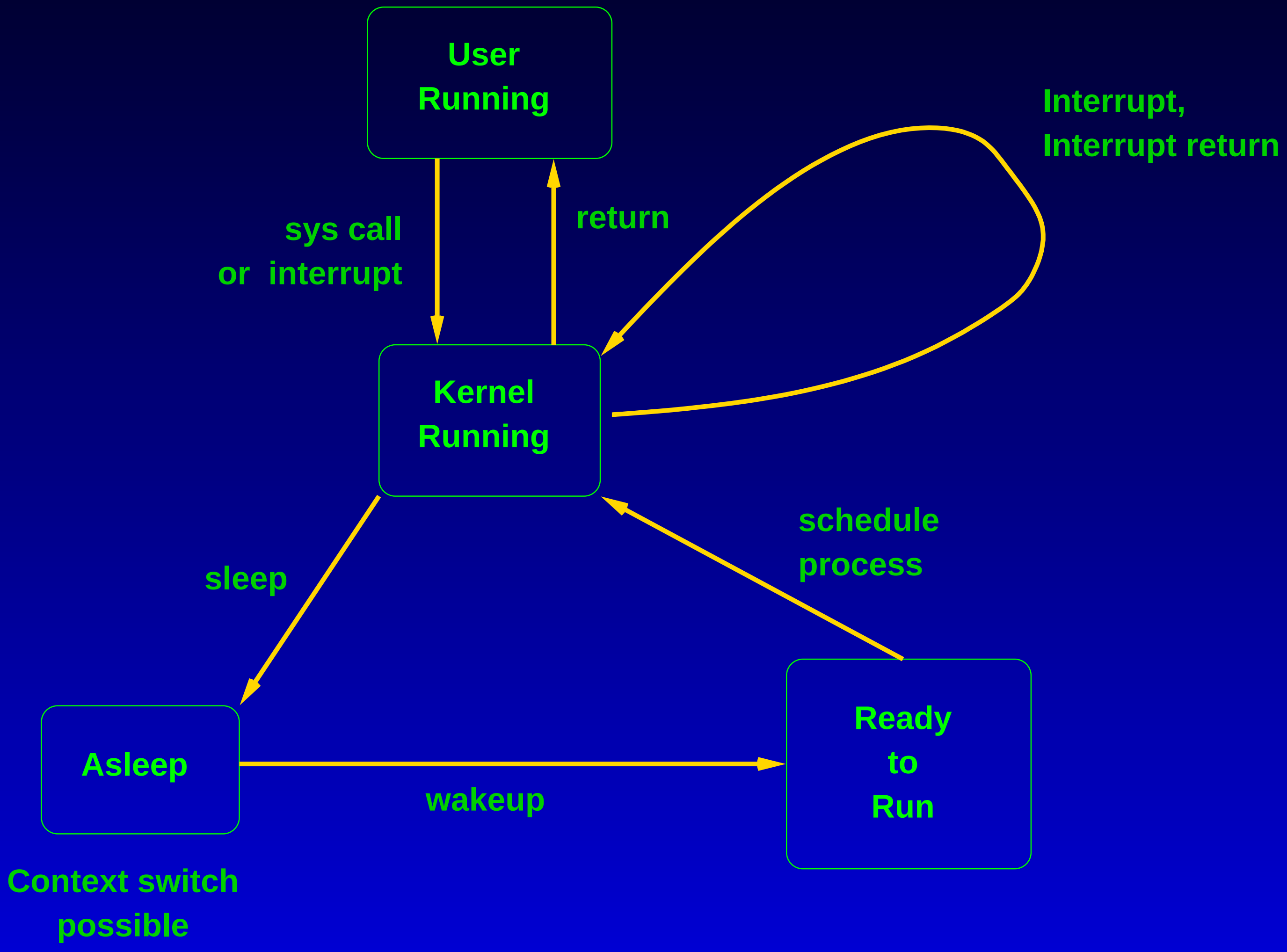
sleeping, swapped process is blocked, awaiting some event, and has been transferred to swap memory. (blocked)

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



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.

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.

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 acknowedged)

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 acknowedged)

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 acknowedged)

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.

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.

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

#include <sys/types.h>

#include <unistd.h>

pid_t getpid(void) get current process' pid

Querying Process Attributes

#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

#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

Querying Process Attributes

#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

#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

#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

Querying Process Attributes

#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 $\equiv$ group id $\equiv$ 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 $\equiv$ group id $\equiv$ 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()

#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:

real user id,
real group id,
effective user id,
effective group id
environment
close-on-exec flag
signal handling settings
signal handling mask
supplementary group ids
set-user-id bit
set-group-id bit

profiling status
nice value (ie. priority)
scheduler class
attached shared memory segments
process group id
session id
current working directory
root directory
filemode creation mask
resource limits
controlling terminal

text segment
data segment
stack segment
file descriptor 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)

```
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()

```
#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**.

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**.

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()

```
#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.

_exit()

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

- This function terminates the calling process *immediately*. Never fails!

_exit()

```
#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.

_exit()

```
#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)

_exit()

```
#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()

```
#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()

```
#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

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

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(), 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(), 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(), 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)*

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

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

child pid	*description*
pid	wait for child with given pid

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

child pid	*description*
pid	wait for child with given pid
-1	wait for any child

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

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

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

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
≤ -2	wait for any child whose process group is |childpid|

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

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
≤ -2	wait for any child whose process group is $\vert$childpid$\vert$

(run waiter)

Waitpid Options and Status

waitpid options	WNOHANG	don’t block

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child
		(job ctrl)

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child *(job ctrl)*
*status	WIFEXITED(s)	$\neq$ 0: child used _exit() $=$ 0: else

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child
		(job ctrl)
*status	WIFEXITED(s)	$\neq$ 0: child used _exit()
		$=$ 0: else
	WEXITSTATUS(s)	returns child exit code
		(via _exit())

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child *(job ctrl)*
*status	WIFEXITED(s)	$\neq$ 0: child used _exit() $=$ 0: else
	WEXITSTATUS(s)	returns child exit code (via _exit())
	WIFSIGNALED(s)	$\neq$ 0: if child terminated due to signal

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child *(job ctrl)*
*status	WIFEXITED(s)	$\neq$ 0: child used _exit() $=$ 0: else
	WEXITSTATUS(s)	returns child exit code (via _exit())
	WIFSIGNALED(s)	$\neq$ 0: if child terminated due to signal
	WTERMSIG(s)	returns signal # that terminated the child

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child *(job ctrl)*
*status	WIFEXITED(s)	$\neq$ 0: child used _exit() $=$ 0: else
	WEXITSTATUS(s)	returns child exit code (via _exit())
	WIFSIGNALED(s)	$\neq$ 0: if child terminated due to signal
	WTERMSIG(s)	returns signal # that terminated the child
	WIFSTOPPED(s)	$\neq$ 0: returns value if child stopped due to job control

Waitpid Options and Status

waitpid options	WNOHANG	don’t block
	else	block
	WNOTRACED	keep waiting for stopped child *(job ctrl)*
*status	WIFEXITED(s)	$\neq$ 0: child used _exit() $=$ 0: else
	WEXITSTATUS(s)	returns child exit code (via _exit())
	WIFSIGNALED(s)	$\neq$ 0: if child terminated due to signal
	WTERMSIG(s)	returns signal # that terminated the child
	WIFSTOPPED(s)	$\neq$ 0: returns value if child stopped due to job control
	WSTOPSIG(s)	returns signal# that stopped the child process

exec() family

```
#include <unistd.h>
int execl   (const char *path,     const char *arg,...);
int execlp  (const char *file,     const char *arg,...);
int execle  (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.

exec() family

```
#include <unistd.h>
int execl  (const char *path,     const char *arg,...);
int execlp (const char *file,     const char *arg,...);
int execle (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

```
#include <unistd.h>
int execl  (const char *path,     const char *arg,...);
int execlp (const char *file,     const char *arg,...);
int execle (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 "/"; its 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:

- 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.

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.
- 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:

- 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.

(see exec1.c)

Argc, Argv, Envp Review

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

argc count of strings contained by args

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)

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)*

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	arg_1	arg_2	arg_3	. . .	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	arg_1	arg_2	arg_3	$\ldots$	NULL

So

$$\begin{aligned} argv[0] &= \text{“pgmname”} \\ argv[1] &= \text{“arg1”} \\ argv[2] &= \text{“arg2”} \\ \ldots &= \ldots \\ argv[n] &= \text{NULL} \end{aligned}$$

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	arg_1	arg_2	arg_3	. . .	NULL

So

$$\begin{aligned} argv[0] &= \text{“pgmname”} \\ argv[1] &= \text{“arg1”} \\ argv[2] &= \text{“arg2”} \\ \ldots &= \ldots \\ argv[n] &= \text{NULL} \end{aligned}$$

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, execlp() 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.

system()

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

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

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.

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.

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

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

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

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

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

```
#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

#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

#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"

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

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 $\longrightarrow$ child) then the process may close the other 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.
- 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 ⟶ 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

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 $\longrightarrow$ 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 $\longrightarrow$ 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

```
#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

```
#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
- dup2() makes newfd be a copy of oldfd. This is useful for i/o redirection:

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
- 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

```
#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
- 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
  ```

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
- 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
```

Pipes, con’t.

- dup2() is an “atomic” call, although it otherwise appears to act like

Pipes, con’t.

- dup2() is an “atomic” call, although it otherwise appears to act like

```
close(nfd);
fcntl(ofd,F_DUPFD,nfd);
```

Pipes, 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().

Pipes, 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().

- **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

Pipes, 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().

- **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:

Pipes, 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().

- **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

Pipes, 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().

- **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()

Pipes, 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().

- **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

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

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

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.

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:

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

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)

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.