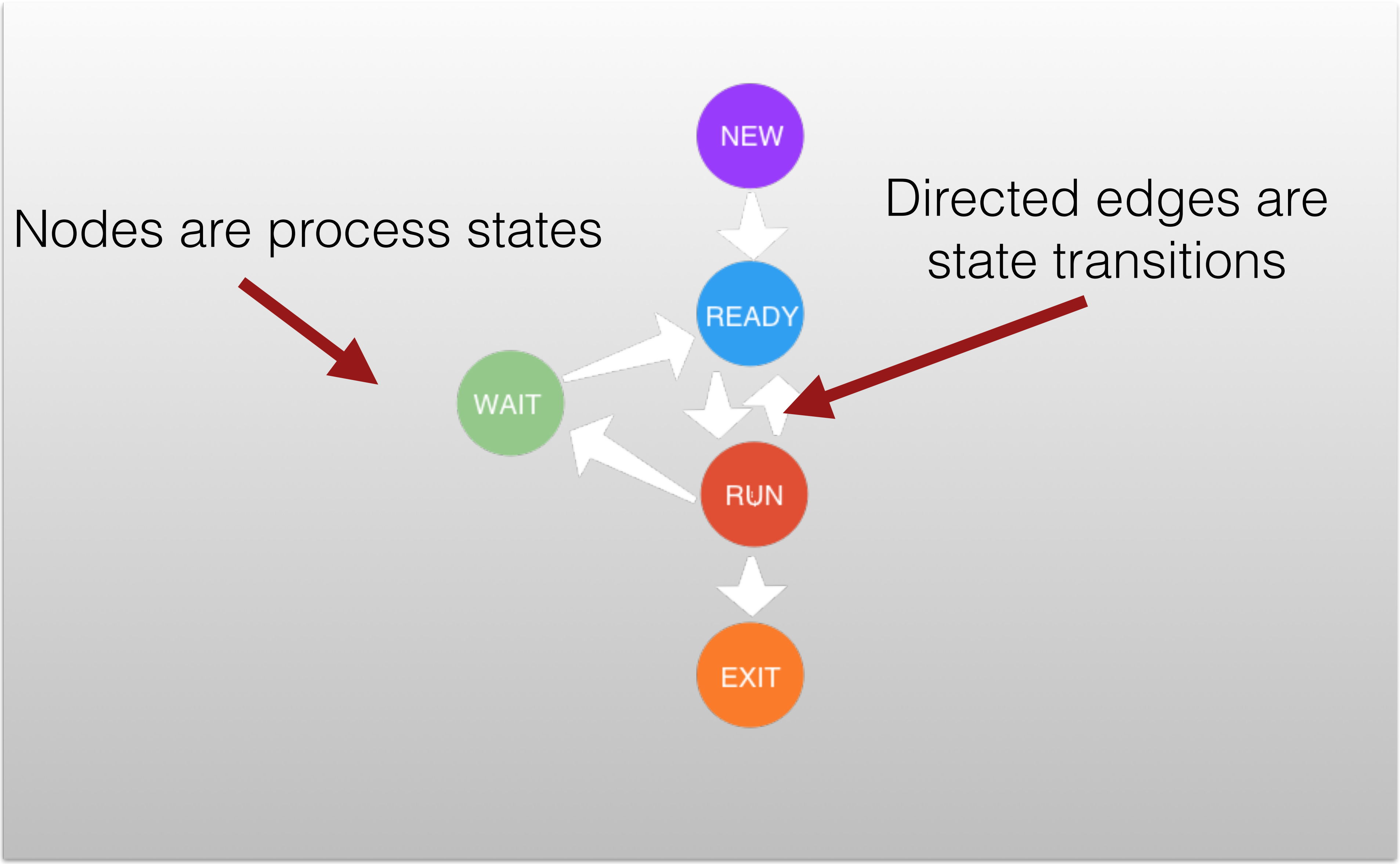
Process State Diagram



Process Creation

- Processes are created when an existing process calls the fork() function
- New process created by fork is called the *child process*
- fork() is a function that is called once but returns twice
    - Only difference in the return value. Returns 0 in the child process and the child's PID in the parent process

fork() man page

FORK(2) Linux Programmer's Manual FORK(2)

NAME

fork – create a child process

SYNOPSIS

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

pid_t fork(void);
```

DESCRIPTION

fork() creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

fork() return values

```
RETURN VALUE
       On success, the PID of the child process is returned  in  the  parent's
       thread  of execution, and a 0 is returned in the child's thread of exe-
       cution.  On failure, a -1 will be returned in the parent's context,  no
       child process will be created, and errno will be set appropriately.

ERRORS
       EAGAIN fork()  cannot  allocate  sufficient memory to copy the parent's
              page tables and allocate a task structure for the child.

       EAGAIN It was not possible to create a new process because the caller's
              RLIMIT_NPROC  resource  limit  was  encountered.  To exceed this
              limit, the process must have either  the  CAP_SYS_ADMIN  or  the
              CAP_SYS_RESOURCE capability.

       ENOMEM fork()  failed  to  allocate  the  necessary  kernel  structures
              because memory is tight.
```

Parent and children PIDs

- Why does fork return the child's PID to parent processes but it returns 0 to the children?
  - Provides a way to determine which process you are in.
  - Processes can only have one parent. To determine the parent PID you can call `getppid()`.
  - Processes can have multiple children so there is no way for a function to give a parent its child PID

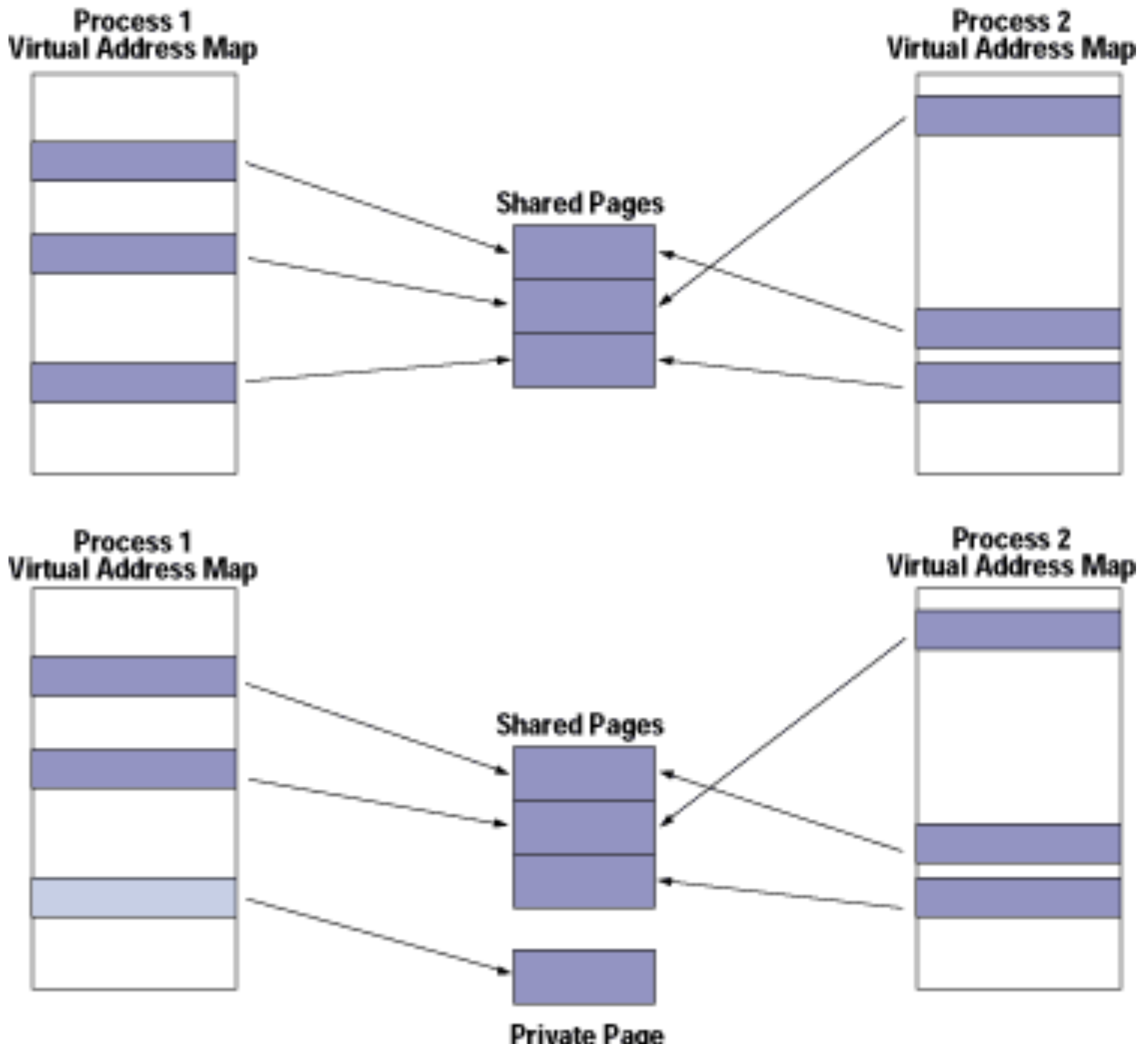
Process Creation

- After fork() both the parent and child continue executing with the instruction that follows the call to fork().
- The child process is a copy of the parent process. The child gets:
  - copy of the parent's data space
  - copy of the parent's heap
  - copy of the parent's stack
  - text segment if it's read-only

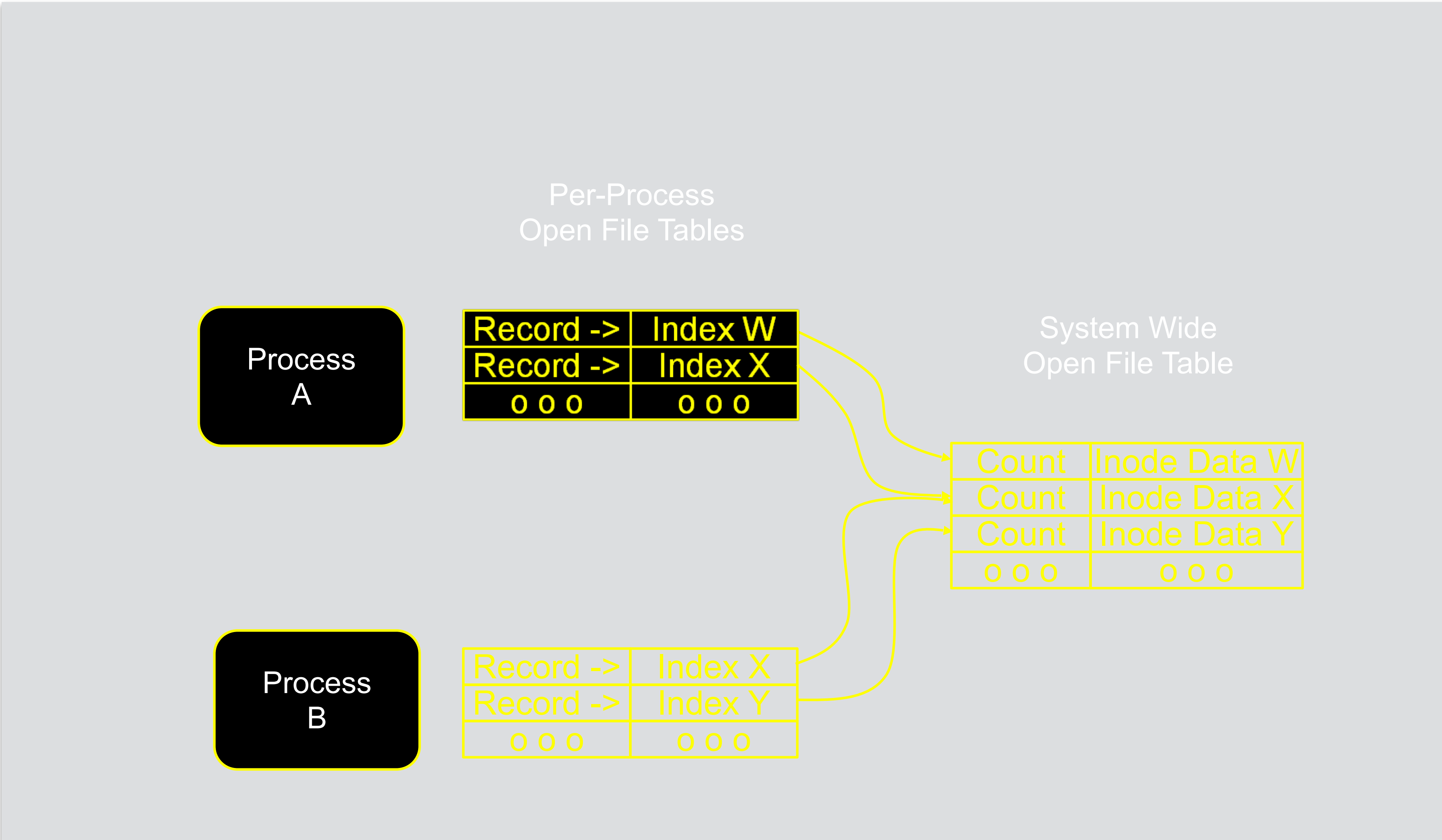
Copy-On-Write

- On Linux the parent process's pages are not copied for the child process.
- The pages are shared between the child and the parent process.
  - Pages are marked read-only
- When either process modifies a page, a page fault occurs and a separate copy of that particular page is made for that process which performed the modification.
- This process will then use the newly copied page rather than the shared one in all future references. The other process continues to use the original copy of the page

Copy-On-Write



File Control Blocks



Inherited Properties

- user ID, group ID
- process group ID
- controlling terminal
- current working directory
- root directory
- file mode creation mask
- signal mask
- environment
- attached shared memory segments
- resource limits

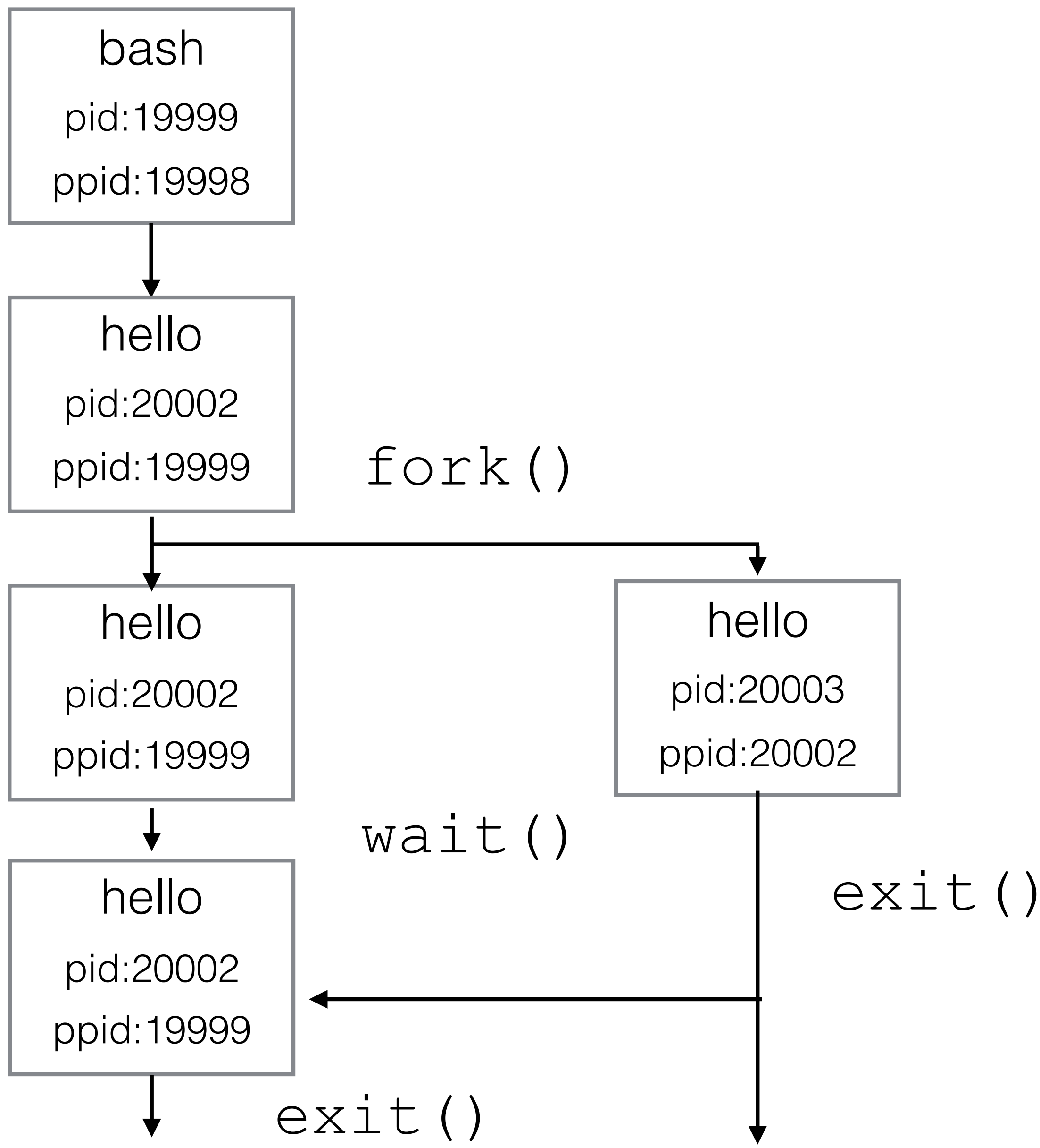
Unique Properties

- the return value from `fork()`
- the process IDs
- the parent process IDs
- file locks
- pending alarms are cleared for the child
- the set of pending signals for the child is set to zero

fork() code example

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

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

Hello World



bash
pid:19999
ppid:19998
hello
pid:20002
ppid:19999
fork()
hello
pid:20002
ppid:19999
hello
pid:20003
ppid:20002
wait()
exit()
hello
pid:20002
ppid:19999
exit()

fork() code example

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

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

I/O is buffered. Make sure to use `flush()`

`fork()` failures

- Too many processes already.
  - Usually a sign something else has gone wrong
  - `cat /proc/sys/kernel/pid_max`
- Too many processes for the current user
  - `ulimit -a`

Terminating a Process

- Three normal ways

1. Executing a return from the `main` function

2. Calling the `exit` function. C library function.

3. Calling the `_exit` function. System call.

    1. You should use _exit to kill the child program when the exec fails. Otherwise the child process may interfere with the parent process' external data by calling its signal handlers, or flushing buffers.

    2. You should also use _exit in any child process that does not do an exec, but those are rare.

Terminating a Process

- Two abnormal ways

1. Calling `abort`. Generates a SIGABORT signal.
2. The process received a signal

What if the parent terminates first?

- `exit` and `_exit` return the termination status of child processes to the parent. Who gets the status if the parent has already terminated?

- The `init` process becomes the parent of any process whose parent terminates.

  - The orphaned process is inherited by the `init` process

  - When a process is terminated the kernel iterates through all the active processes to see if the terminating process is the parent of any remaining processes. If so, it inherits that process

Zombie Process

- A process that has completed execution but still has an entry in the process table.
  - The entry is still needed to allow the parent process to read its child's exit status
- A child process always becomes a zombie before being removed from the resource table.
  - Normally, zombies are immediately waited on by their parent and then reaped by the system

`wait()` and `waitpid()`

- When a process terminates the parent is notified by the operating system sending a SIGCHLD signal.
  - Default handling is to ignore the signal
- Child termination is asynchronous
- POSIX provides two system calls `wait` and `waitpid`

wait()

```
WAIT(2)                    Linux Programmer's Manual                    WAIT(2)

NAME
       wait, waitpid - wait for process to change state

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

       pid_t wait(int *status);
       pid_t waitpid(pid_t pid, int *status, int options);
       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

- `wait` suspends the parents operation until the next child process terminates.
- If there is a child process already terminated and waiting for reaping `wait` will return immediately
- PID of the terminated child process is returned on success. -1 if failure.

`wait()` status parameter

WIFEXITED(status**)**
returns true if the child terminated normally, that is, by calling **exit**(3) or **_exit**(2), or by returning from main().

WEXITSTATUS(status**)**
returns the exit status of the child. This consists of the least significant 16-8 bits of the status argument that the child specified in a call to **exit**() or **_exit**() or as the argument for a return statement in main(). This macro should only be employed if **WIFEXITED** returned true.

WIFSIGNALED(status**)**
returns true if the child process was terminated by a signal.

WTERMSIG(status**)**
returns the number of the signal that caused the child process to terminate. This macro should only be employed if **WIFSIGNALED** returned true.

`wait()` status parameter

```
WCOREDUMP(status)
       returns true if the child produced  a  core  dump.   This  macro
       should  only  be  employed  if  WIFSIGNALED returned true.  This
       macro is not specified in POSIX.1-2001 and is not  available  on
       some  Unix  implementations  (e.g.,  AIX, SunOS).  Only use this
       enclosed in #ifdef WCOREDUMP ... #endif.

WIFSTOPPED(status)
       returns true if the child process was stopped by delivery  of  a
       signal;  this  is  only possible if the call was done using WUN-
       TRACED or when the child is being traced (see ptrace(2)).

WSTOPSIG(status)
       returns the number of the signal which caused the child to stop.
       This  macro should only be employed if WIFSTOPPED returned true.

WIFCONTINUED(status)
       (Since Linux 2.6.10) returns  true  if  the  child  process  was
       resumed by delivery of SIGCONT.
```

waitpid()

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

- Allows the parent process to wait on a specific child process.

wait() code example

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    // Sleep for a second
    sleep(1);

    // Intentionally SEGFAULT the child process
    int *p = NULL;
    *p = 1;

    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  // See if the child was terminated by a signal
  if( WIFSIGNALED( status ) )
  {
    // Print the signal that the child terminated with
    printf("Child returned with status %d\n", WTERMSIG( status ) );
  }

  return 0;
}
```

Process Creation

`fork()` creates an exact copy of a process. How do we create a unique process?

By combining `fork` with an `exec` function

`exec` functions

- When exec is called the new program, specified by exec, completely replaces the running process.
  - text, data, heap and stack are all replaced
  - PID stays the same since it’s not a new process

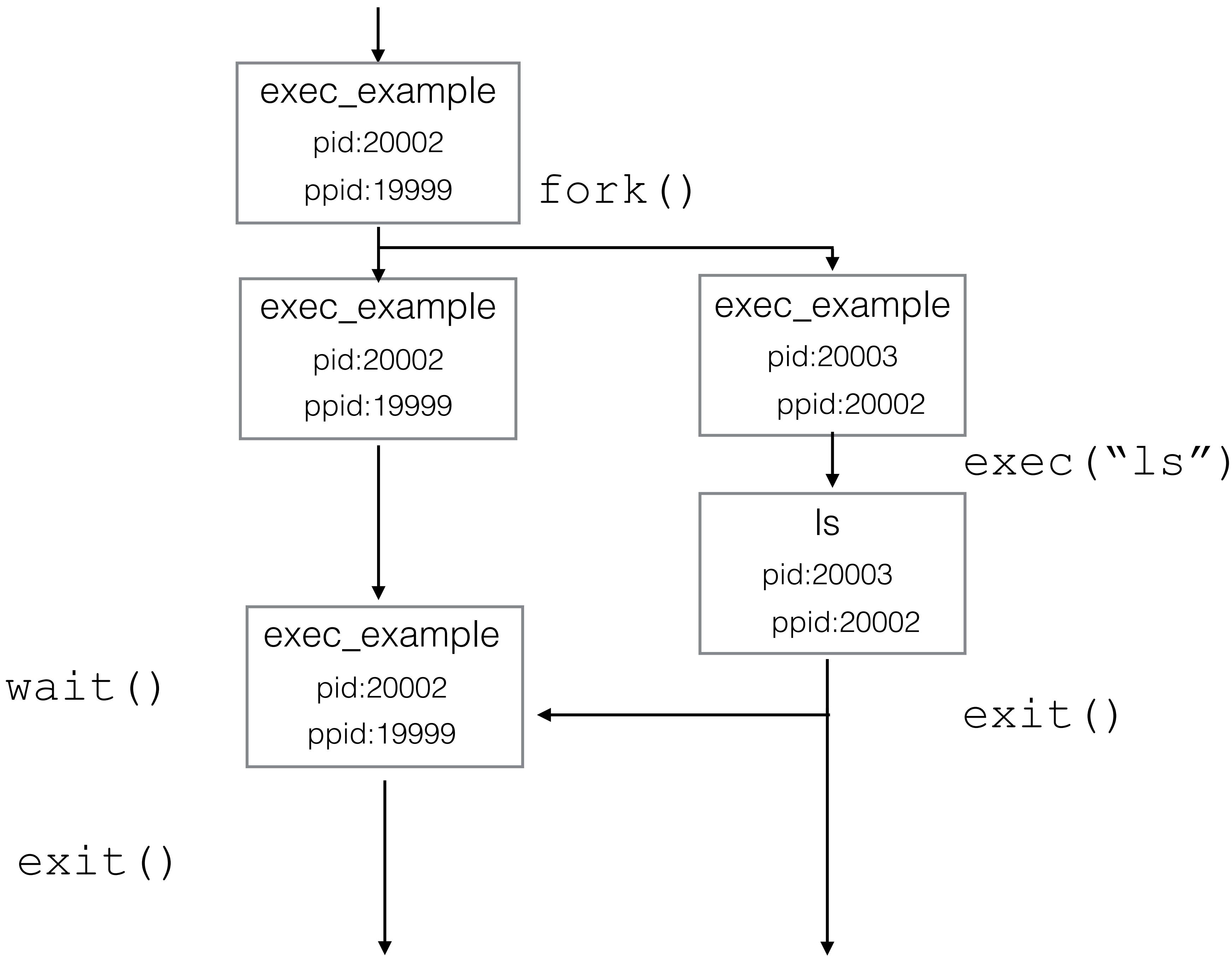
`exec` family

```
int execl(char const *path, char const *arg0, ...);
int execle(char const *path, char const *arg0, ..., char const *envp[]);
int execlp(char const *file, char const *arg0, ...);
int execv(char const *path, char const *argv[]);
int execve(char const *path, char const *argv[], char const *envp[]);
int execvp(char const *file, char const *argv[]);
```

Meaning of the letters after exec:
l – A list of command-line arguments are passed to the function.
e – An array of pointers to environment variables is passed to the new process image.
p – The PATH environment variable is used to find the file named in the path argument.
v – Command-line arguments are passed to the function as an array of pointers.

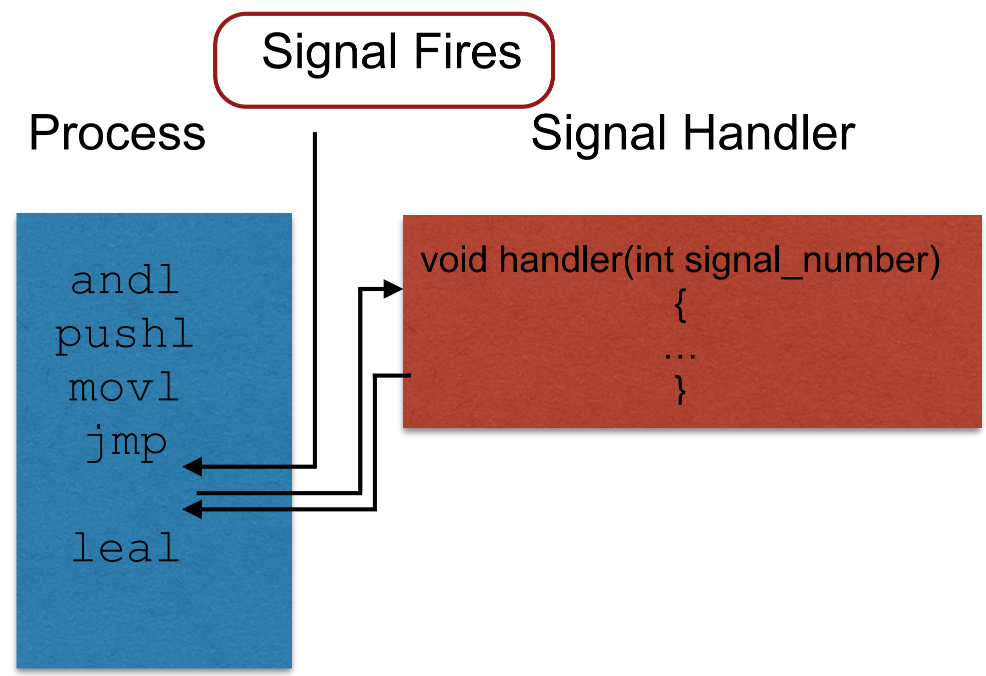
exec code example

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    execl("/bin/ls", "ls", NULL );
    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  return 0;
}
```


exec_example
pid:20002
ppid:19999
fork()
exec_example
pid:20002
ppid:19999
exec_example
pid:20003
ppid:20002
exec("ls")
ls
pid:20003
ppid:20002
wait()
exec_example
pid:20002
ppid:19999
exit()
exit()

Signals

- A signal is a asynchronous notification of an event
- Signals are how the operating system communicates with an application process.
- When a signal is received the process stops running and the assigned signal handler is run.
- When the signal has been handled the process resumes where it left off.



Some examples

- SIGSEGV - Known as a segmentation fault. It occurs when a process makes an illegal memory reference. The process halts and the default signal handler exits the process.

- SIGINT - The interrupt signal occurs when the user types ctrl-c. The default signal handler will exit the process.

Sending Signals

- By keyboard
  - ctrl-c sends the SIGINT signal and the receiving process exits
  - ctrl-z sends the SIGTSTP signal and the receiving process is suspended
  - ctrl-\ sends the SIGQUIT signal and the receiving process exits

Sending signals via shell commands

kill -signal pid

- Send a signal, specified by -signal, to the process with the process id pid.
- If no signal is specified then the signal sent is SIGTERM

Example:

```
kill -9 3412
kill -SIGQUIT 3412
```

Send signal via a function

int kill(pid_t pid, int sig)

- Send a signal, specified by -signal, to the process with the process id pid.
- If no signal is specified then the signal sent is SIGTERM

Example:

```
pid_t pid = getpid();       // process gets its own pid
kill(pid, SIGINT);          // and sends itself a SIGINT
```

Send signal via a function

int raise(int sig)

- Send a signal to the current process

Example:

```
raise(SIGINT);  // process sends itself a SIGINT
```

Signal Numbers

```
[bakker@crystal ~]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS      34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Signal Terminology

- A signal is *generated* for a process when the event that causes the signal occurs. The event can be a hardware exception, a software condition, or a call to the kill function

- When a signal is generated the kernel usually sets a flag in the process table

Signal Terminology

- A signal is *delivered* to a process when the action for a signal is taken.
- During the time between the generation of the signal and the delivery of the signal the signal is said to be *pending*.

Signal Handling

- Every signal is assigned a default handler. Usually they just exit the process.
- Programs can install their own signal handlers for most signals.
  - Can’t install handlers for:
    - SIGKILL
    - SIGSTOP

Installing a signal handler

```
SIGACTION(2)                Linux Programmer's Manual               SIGACTION(2)

NAME
       sigaction - examine and change a signal action

SYNOPSIS
       #include <signal.h>

       int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- The function *handler* specified in the sigaction struct is installed as the new handler for the signal specified by signum.
- The new handler is called every time the process receives a signal of type signum.

Signal Handler Example

- See sigint.c and multiple_signal_handlers.c on course website under Code Samples

Blocking Signals

- We can block signals, but why would we want to?
  - Race conditions. What happens when a signal is received while we are in the middle of handling the same type of signal?
- The POSIX standard provides a way to block signals.

Blocking Signals

- Each process has a signal mask
    - The operating system uses the mask to determine which signals to deliver.
- The function sigprocmask() provides a way for programs to modify their signal mask to block and unblock signals.

Blocking Signals

- If a signal is generated for a process but blocked then the signal remains pending for the process until the process either:
  - unblocks the signal
  - changes the action to ignore the signal

Multiple Blocked Signals

- The POSIX standard allows the OS to deliver one or multiple copies of the signal.
- Most UNIX implementations do not queue signals and just deliver a singe one.

sigprocmask()

```
SIGPROCMASK(2)              BSD System Calls Manual              SIGPROCMASK(2)

NAME
     sigprocmask -- manipulate current signal mask

SYNOPSIS
     #include <signal.h>

     int
     sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

- First parameter is *how.* It can be:
  - SIG_BLOCK - adds the provided set to the current signal mask
  - SIG_UNBLOCK - removes the provided set from the current signal mask
  - SIG_SETMASK - the current mask is replaced by the provided set

sigprocmask()

```
SIGPROCMASK(2)              BSD System Calls Manual              SIGPROCMASK(2)

NAME
     sigprocmask -- manipulate current signal mask

SYNOPSIS
     #include <signal.h>

     int
     sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset);
```

- Second parameter is a pointer to the a signal mask called *set.*

- Third parameter is a pointer to which the old set can be returned.

Signal Blocking Example

- See sig_set_example.c on course website under Code Samples

Process Control