# CSSE2310/7231 — 5.2

Processes (continued)

# ZOMBIES

See `zom1.c`.

# ZOMBIES

See `zom1.c`.

The system needs to keep a record of the what happened to a process in case the parent is interested.

- Did it exit normally?
  - with what status?
- It terminated rather than exiting normally.
  - Which signal caused that?

The memory and resources the process was using have already been released but part of the process still hangs around. A process in that state is called a zombie.

# Reaping a zombie

Reaping — when a process' parent has asked about COD.

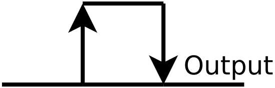
- zombie will be removed

To reap, the parent process calls `wait()`. Wait blocks until either:

- A current child process becomes (or already was) a zombie.
- The parent has no child processes (returns error)

See `zom2.c`

See `zom2.c`



Zombies can't be `killed`. — There is no code left to handle a signal.

Zombies can't be `killed`. — There is no code left to handle a signal.

Q: How do you stop a zombie?

Zombies can't be `killed`. — There is no code left to handle a signal.

Q: How do you stop a zombie?

A: You drop a wait on it.

# Reaping and adoption

If a process

- is “alive”
- has zombie children
- doesn’t reap them

then those zombies will stay on the system (the parent might ask about them evenutally). For long running processes like servers, this could be a problem.

# Reaping and adoption

What if a process dies and has running child processes?

See `adopt.c`

# Reaping and adoption

What if a process dies and has running child processes?

See `adopt.c`

- Only the direct parent of a process can reap it.
- The children will continue to run.
- All that process' children will be adopted by process 1.

# Decoding “status”

The information returned by `wait` needs to be decoded.

- `WIFEXITED(status)` — true if the process exited normally
  - then `WEXITSTATUS(status)` — the exit status of the process.
- `WIFSIGNALED(status)` — true if the process was terminated by a signal.
  - `WTERMSIG(status)` — the signal which caused the process to terminate.

See `f7.c`

Wait can be used to check for things other than a process ending, but we don’t need them now.

# waitpid()

See f8.c

The third arguemnt will be 0 for now.
Later we may use W_NOHANG.

# Changing script

A process can change which program it is running.

`int execl(char* path, char* arg0, char* arg1, char* arg2, ...)`

- The last thing in the list must be a null pointer
- Replaces the old process image with a new one.
  - The old stack and heap are gone.
  - The old program instructions are gone.
  - Resources on the kernel side (eg files) are kept.
- returns $-1$ if the operation fails
- never returns anything but $-1$.

See `exec1.c`

# Pathing

- From `exec1.c`:
  `./a.out ls` doesn't work.
- To the contents of the `PATH` variable into account, use `execlp`.
- In this course, you should always use the `p` forms of exec.

# execvp

See `exec2.c`

`int execvp(char* arg0, char** argv)`.

- More useful for varying numbers of commandline arguments.
- The last element in argv, must be a null pointer.

There is also `execvpe()`[1] but we don't need them in this course.

[1] and others

# Summary so far

So, . . . in order to run another program . . .

# Summary so far

So, . . . in order to run another program . . .

- we clone ourselves

# Summary so far

So, . . . in order to run another program . . .

- we clone ourselves
- replace the clone's memory with a new program.

# Summary so far

So, . . . in order to run another program . . .

- we clone ourselves
- replace the clone's memory with a new program.

Is this overly complicated or is there a reason for this?

# Summary so far

So, . . . in order to run another program . . .

- we clone ourselves
- replace the clone's memory with a new program.

Is this overly complicated or is there a reason for this?
Yes

# Signals

- Signals are very simple messages from the kernel to a process.
  - They don't carry information other than "One or more of some event has happened"
  - If a previous signal of that type hasn't been handled yet, additional signals[2] will be ignored.
- The kernel will send signals:
  - On its own initiative:
    - the process has accessed invalid memory — segfault (SIGSEGV)
    - the process tried to write to a destination that won't accept input (SIGPIPE)
    - . . .
  - Because a process has asked it to:
    - shell `kill` command.
    - C `kill` system call.

[2] of that type

# `kill`?

- Processes have signal handling functions registered with the kernel.
  - When a signal is delivered, the kernel fakes a function call in the process (see “exceptions” last week).
- If your program doesn’t set up new ones, the default handler will be used.
  - The default handler usually terminates the process.
- `kill` commands are actually “send a given signal to a process”.
  - . . . often that results in the process not being alive.

# signal numbers

To see signals available on your system in bash:

`kill -l` (n.b. dash el)

```
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
...
```

In code, use symbolic names:

eg `SIGINT`

# sigaction

From `man sigaction`

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

There are older functions like:

- signal
- sigset

do not use them.

# sig.c

See `sig.c`

# sig.c

See `sig.c`

How do I stop valgrind complaining about my sigaction variable?

- Add `memset(&sa, 0, sizeof(struct sigaction));` after declaration

# sig.c

See `sig.c`

How do I stop valgrind complaining about my sigaction variable?

- Add `memset(&sa, 0, sizeof(struct sigaction));` after declaration

SA_RESTART?

- If a signal arrives while a system call is running, restart the sys call.

# How do I stop it?

What if a process is trapping `SIGINT`?

- `^\`— sends `SIGQUIT`
- kill -3 PID / kill -QUIT PID— sends `SIGQUIT`
- kill -9 PID — sends `SIGKILL`
  - `SIGKILL` can't be blocked or caught

`kill -9` all the time?

[3] Usually

# How do I stop it?

What if a process is trapping `SIGINT`?

- ^\— sends `SIGQUIT`
- kill -3 PID / kill -QUIT PID— sends `SIGQUIT`
- kill -9 PID — sends `SIGKILL`
  - `SIGKILL` can't be blocked or caught

`kill -9` all the time?

- Better[3] to give processes a chance to clean up.

[3] Usually

# SIGCHLD

`Wait()` will block until a child ends.

Not useful if you want to do other work while the child is running.

- You could use:
  ```
  result = waitpid(pid, &status, W_NOHANG)
  ```
  To check the status of a particular child.
  - What if there are lots of children? Need to check each one?
- `SIGCHLD` is sent to the parent process when something happens to one of its children.
  - You can set a handler to act when SIGCHLD arrives.

# SIGCHLD

See `child.c`

# Notes

- Ideally your signal handlers should not:
    - take a long time to execute
    - aquire locks
- `sigwait()` may be be useful if your main program needs to wait for a signal.
- Can't we just do ... and zombies will never be created?
    - The documentation says that techniques vary across systems.
- "Core dumps?" — in ye olden days segfaults (and some other signals) would cause a copy of memory (core) to be put into a file:
    - These were useful for
        - debugging
        - using all your quota
    - Most modern linux systems have core dumps disabled.

# Other signals

- SIGSEGV — surprise segfault
- SIGWNCH — your terminal window changed size