



# Shells and Signals

# shell: program that runs other programs

The image shows a presentation slide titled "shell: program that runs other programs" displayed in a Beamer presentation window. The slide content is as follows:

```
shell
program that runs other programs on behalf of the user
sh      Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
bash   "Bourne-Agash" Shell, widely used
       default on most Unix/Linux/Mac OS X systems
others ...

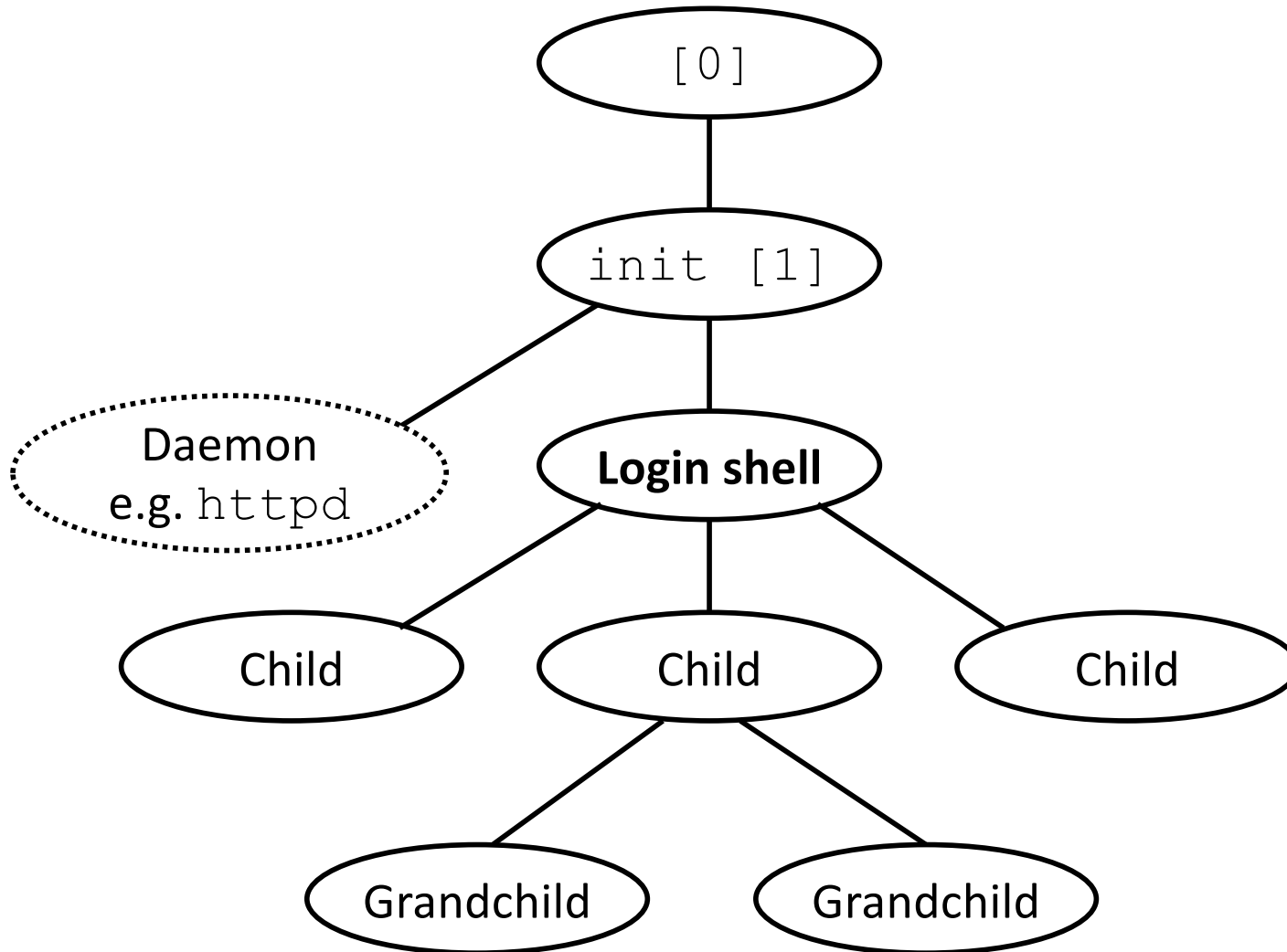
while (true) {
  Print command prompt.
  Read command line from user.
  Parse command line.
  If command is built-in, do it.
  Else fork process to execute command.
  in child:
    Execute requested command with exec.
    (never returns)
  in parent:
    Wait for child to complete.
}
```

Overlaid on the slide is a terminal window with the following output:

```
2. bpw@queets: ~/240/slides (bash)
Last login: Tue Nov 10 18:02:56 on ttys009
[09:50 PM] bpw@queets: ~$ cd 240/slides
[09:50 PM] bpw@queets: ~/240/slides$ ls
alu.pptx          mem.pptx          signals.pptx
arrays.pptx       memalloc.pptx    stuff.pptx
beyond240.pptx   memalloc.pptx.pdf virtual-memory.pptx
bits.pptx        membugs.pptx     x86-basics.pptx
buffer.pptx      mini-mips.pdf    x86-control.pptx
cache.pptx       mini-mips2.pdf   ~$buffer.pptx
datapath.pptx   mux.pptx         ~$datapath.pptx
gates-notes.txt os-ecf.pptx     ~$mux.pptx
gates.pptx      pc.pptx          ~$ram.pptx
gc.pptx         pdfs@            ~$seq.pptx
integers.pptx   procedure-stack.pptx ~$signals.pptx
intro.pptx      process.pptx     ~$x86-basics.pptx
isa.pptx        ram.pptx         ~$x86-control.pptx
java.pptx       seq.pptx
[09:50 PM] bpw@queets: ~/240/slides$
```

The presentation window also shows a "Picture Format" toolbar with options like "Remove Background", "Change Picture", "Corrections", "Color", "Artistic Effects", "Transparency", "Quick Styles", "Bring Forward", "Send Backward", "Reorder Objects", "Align", "Crop", and "Format Pane". The slide number "6" is visible in the bottom right corner.

# Shells and the process hierarchy



# Shell logic

program that runs other programs on behalf of the user

**sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)

**bash** “Bourne-Again” Shell, widely used  
default on most Unix/Linux/Mac OS X systems

many others...

```
while (true) {  
    Print command prompt.  
    Read command line from user.  
    Parse command line.  
    If command is built-in, do it.  
    Else fork process to execute command.  
        in child:  
            Exec requested command (never returns)  
        in parent:  
            Wait for child to complete.  
}
```

# Terminal ≠ shell

User interface to shell and other programs.

Graphical (GUI) vs. command-line (CLI)

Command-line terminal (emulator):

Input (keyboard)

Output (screen, sound)

# To wait or not?

A *foreground* job is a process for which the shell waits.\*

```
$ emacs fizz.txt # shell waits until emacs exits.
```

A *background* job is a process for which the shell does not wait\*... yet.

```
$ emacs boom.txt & # emacs runs in background.  
[1] 9073           # shell saves background job and is...  
$ gdb ./umbrella  # immediately ready for next command.
```

don't do this with emacs unless using X windows version

\*Also: foreground jobs get input from (and "own") the terminal. Background jobs do not.

# Signals

*Signal*: small message notifying a process of event in system

like exceptions and interrupts

sent by kernel, sometimes at request of another process

ID is entire message

<i>ID</i>	<i>Name</i>	<i>Corresponding Event</i>	<i>Default Action</i>	<i>Can Override?</i>
2	SIGINT	Interrupt (Ctrl-C)	Terminate	Yes
9	SIGKILL	Kill process (immediately)	Terminate	<b>No</b>
11	SIGSEGV	Segmentation violation	Terminate & Dump	Yes
14	SIGALRM	Timer signal	Terminate	Yes
15	SIGTERM	Kill process (politely)	Terminate	Yes
17	SIGCHLD	Child stopped or terminated	Ignore	Yes
18	SIGCONT	Continue stopped process	Continue (Resume)	No
19	SIGSTOP	Stop process (immediately)	Stop (Suspend)	<b>No</b>
20	SIGTSTP	Stop process (politely)	Stop (Suspend)	Yes

...

# Sending/receiving a signal

Kernel *sends* (delivers) a signal to a *destination process* by updating state in the context of the destination process.

Reasons:

**System event**, e.g. segmentation fault (SIGSEGV)

**Another process used `kill`** system call:

explicitly request the kernel send a signal to the destination process

Destination process *receives* signal when kernel forces it to react.

Reactions:

***Ignore*** the signal (do nothing)

***Terminate*** the process (with optional core dump)

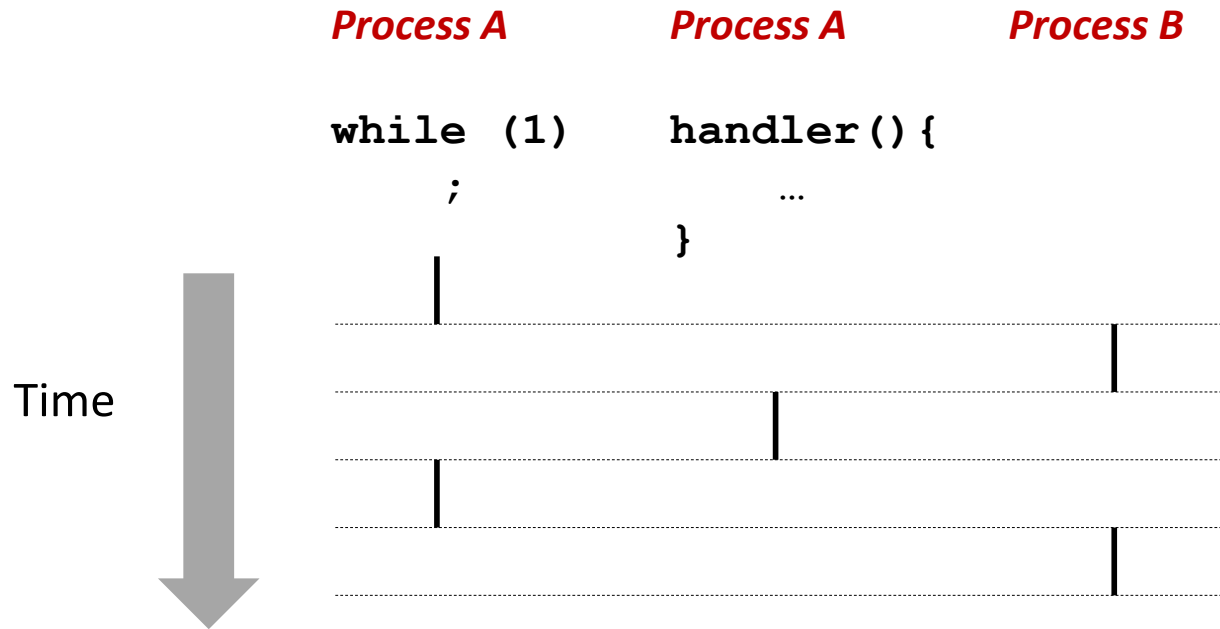
***Catch*** the signal by executing a user-level function called *signal handler*

Like an impoverished Java exception handler



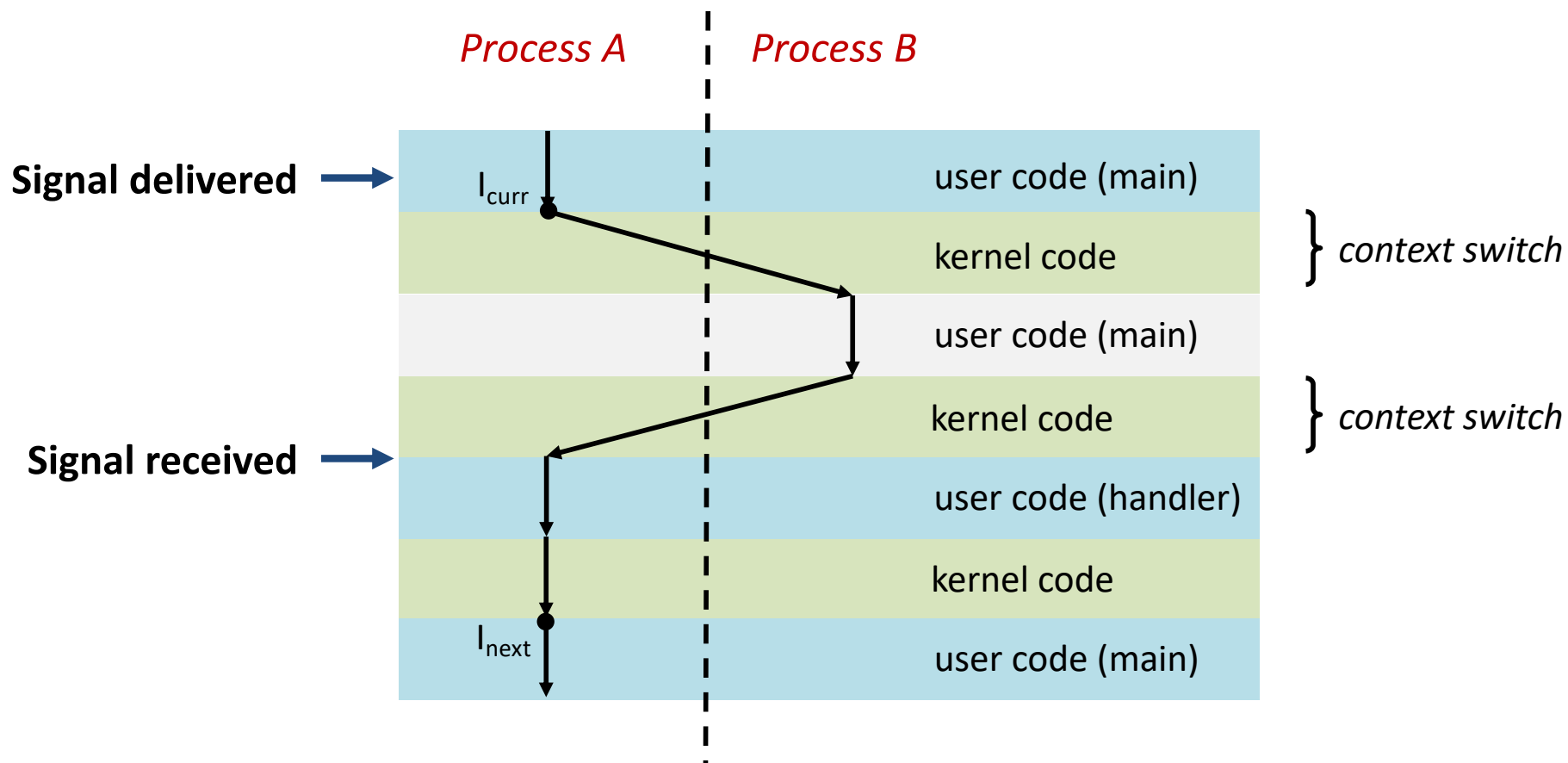
# Signals handlers as concurrent flows

Signal handlers run concurrently with main program (in same process).



# Another view of signal handlers as concurrent flows

optional



# Pending and blocked signals

A signal is *pending* if sent but not yet received

<= 1 pending signal per type per process

No Queue! Just a bit per signal type.

Signals of type S discarded while process has S signal pending.

A process can *block* the receipt of certain signals

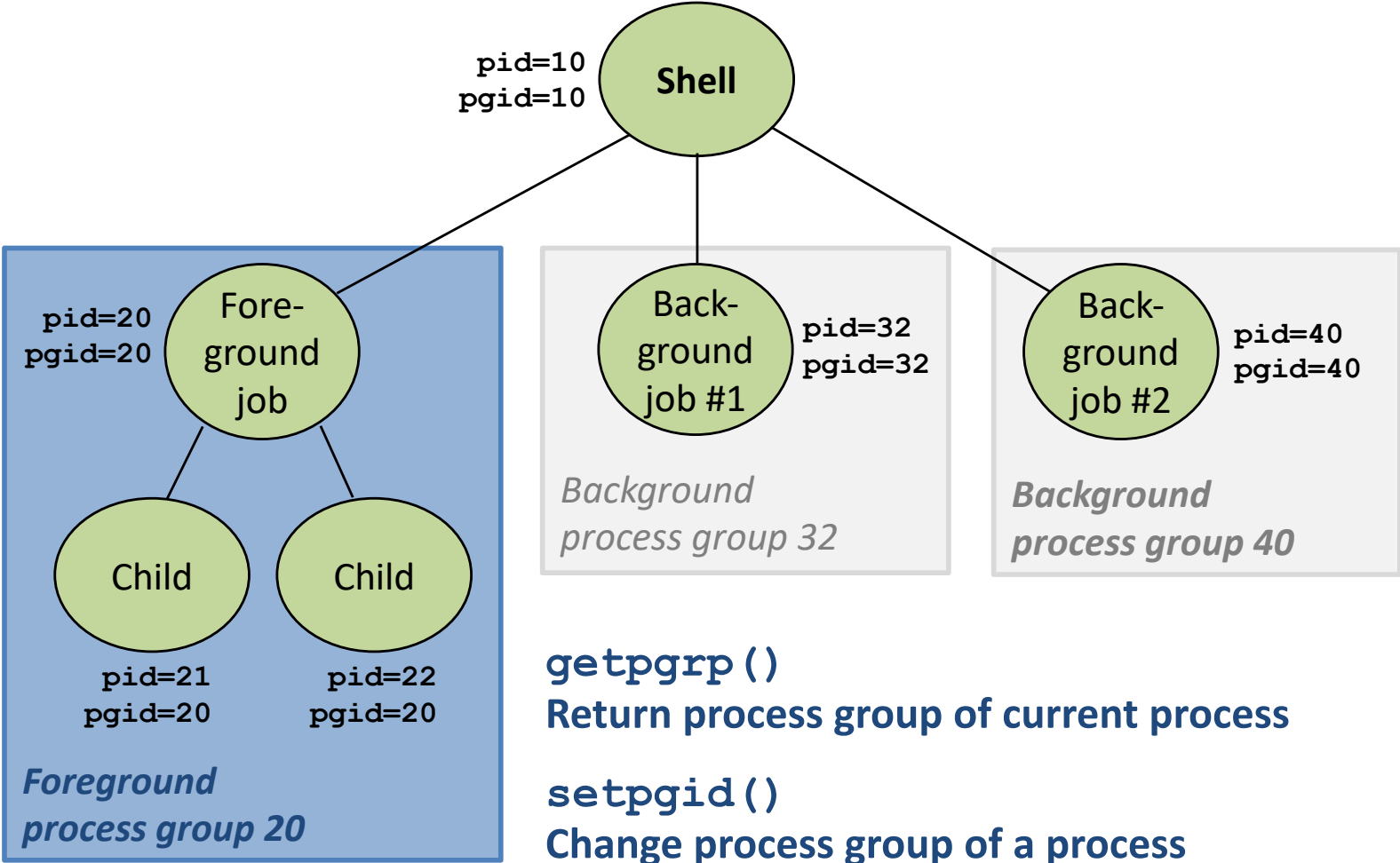
Receipt delayed until the signal is unblocked

A pending signal is received at most once

Let's draw a picture...

# Process Groups

Every process belongs to exactly one process group (default: parent's group)



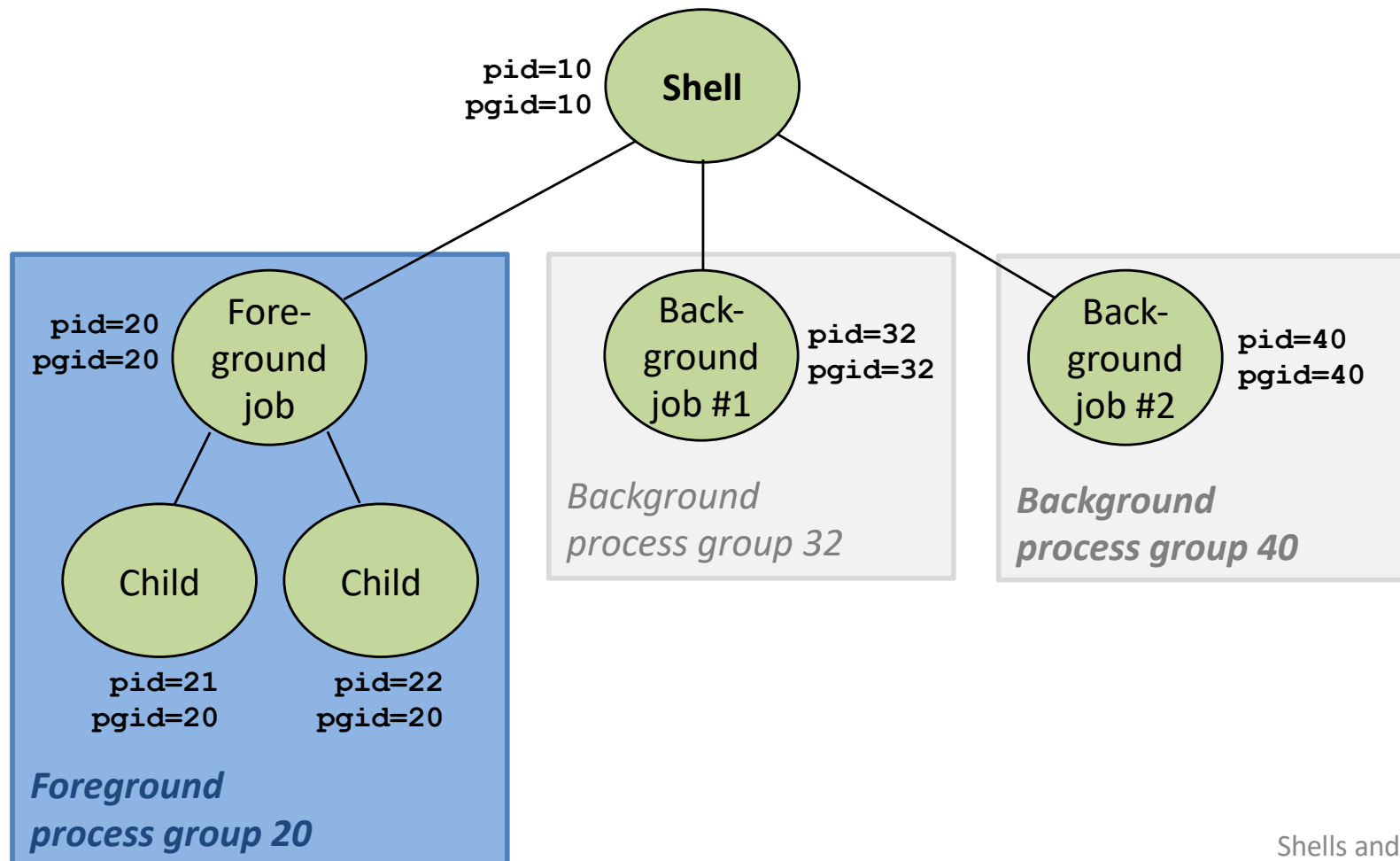
# Sending signals from the keyboard

optional

Shell: Ctrl-C sends SIGINT (Ctrl-Z sends SIGTSTP)  
to every job in the foreground process group.

SIGINT – default action is to terminate each process

SIGTSTP – default action is to stop (suspend) each process



# Signal demos

Ctrl-C

Ctrl-Z

kill

```
kill (pid, SIGINT) ;
```

# A program that reacts to externally generated events (Ctrl-c)

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

void handler(int sig) {
    safe_printf("You think hitting ctrl-c will stop me?\n");
    sleep(2);
    safe_printf("Well...");
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    while(1) {
    }
}
```

```
> ./external
<ctrl-c>
You think hitting ctrl-c will stop me?
Well...OK
>
```

external.c

# A program that reacts to internally generated events

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
    safe_printf("BEEP\n");

    if (++beeps < 5)
        alarm(1);
    else {
        safe_printf("DING DING!\n");
        exit(0);
    }
}
```

internal.c

```
main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in
              1 second */

    while (1) {

    }
}
```

```
> ./internal
BEEP
BEEP
BEEP
BEEP
BEEP
DING DING!
>
```



# Signal summary

Signals provide process-level exception handling

- Can generate from user programs

- Can define effect by declaring signal handler

## Some caveats

- Very high overhead

  - >10,000 clock cycles

  - Only use for exceptional conditions

- Not queued

  - Just one bit for each pending signal type

- Many more complicated details we have not discussed.

  - Book goes into too much gory detail.