Processes
(part 2)

M1 MOSIG – Operating System Design

Renaud Lachaize

Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (many slides/figures directly adapted from those of the CS140 class)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Randall Bryant, David O’Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
    - Textbook: Computer Systems: A Programmer’s Perspective (2nd Edition) a.k.a. “CSAPP”
    - CS 15-213/18-243 classes (many slides/figures directly adapted from these classes)
  - Textbooks (Silberschatz et al., Tanenbaum)
  - Michael Kerrisk, The Linux Programming Interface

Outline

- Signals
- Inter-Process Communication facilities

(Remember previous lecture on processes) Problem with mini-shell example

- The shell correctly waits for and reaps the foreground jobs

- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could theoretically run the kernel out of memory
  - Modern Unix systems: once you exceed your process quota, your shell can't run any new commands for you: **`fork()`** returns -1

Problem with mini-shell example (continued) Introducing signals

- Problem
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow cannot reap exited background processes in a timely fashion
  - Regular control flow is "wait until running job completes, then reap it"

- Solution: Exceptional control flow
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a ***signal***
  - **This mechanism is actually more general and is also used for other purposes (not just job completion)**

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts (but managed in software, and handled at the user level)
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID (1-30)
  - only information in a signal is its ID and the fact that it arrived

Some examples:

ID	*Name*	*Default Action*	*Corresponding Event*
2	`SIGINT`	**Terminate**	**Interrupt (e.g., ctl-c from keyboard)**
9	`SIGKILL`	**Terminate**	**Kill program (cannot override or ignore)**
11	`SIGSEGV`	**Terminate & Dump**	**Segmentation violation**
14	`SIGALRM`	**Terminate**	**Timer signal**
17	`SIGCHLD`	**Ignore**	**Child stopped or terminated**

Signals (continued)

- Many signals are generated and received **asynchronously**
  - i.e., these signals are triggered by an event that is not related to the last instruction executed by the destination process
  - Example: **`SIGCHLD`** indicating that a child process has terminated or has been stopped (suspended)

Signals (continued)

- ... However, in some cases, signals are **synchronously** generated and received immediately
  - A process can send a signal to itself
  - A signal may be triggered by a hardware fault, such as a floating point exception/arithmetic error or a memory protection fault
    - In general, it makes little sense to block or ignore such signals, as the process cannot make progress
    - The only appropriate reaction is often to terminate the process
    - ... But there are some counter examples:
      - User-level virtual memory management using **`mprotect()`** and **`SIGSEGV`** handler (e.g., for user-level checkpointing)
      - Calling a special function (e.g., **`siglongjmp`**) in the signal handler allowing to “rewind” the execution of the process, so that it does not keep re-executing the same problematic instruction

Sending a signal

- The kernel *sends* a signal to a *destination process* by updating some state in the context of the destination process

- The kernel sends a signal for one of the following reasons:

  - **The kernel has detected a system event**. Examples:
    - an erroneous arithmetic operation, such as a divide-by-zero error (**`SIGFPE`**)
    - the expiration of a timer (**`SIGALRM`**)
    - the termination of a child process (**`SIGCHLD`**)

  - **Another process has invoked the `kill` system call** to explicitly request the kernel to send a signal to the destination process

Receiving a signal

- A destination process *receives* a signal when it is forced by the kernel to react in some way to the arrival of the signal

- Three possible ways to react:
  - ***Ignore*** the signal (do nothing)
  - ***Terminate*** the process (with optional core dump)
  - ***Catch*** the signal by executing a user-level function called ***signal handler***
    - Akin to a hardware-managed handler being called in response to an interrupt/exception

Signal concepts

- A signal is *pending* if sent but not yet received
  - For a given signal type, a process can at most have one pending signal instance
  - Important: **Signals are not queued**
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- A process can *block* the receipt of certain signals
  - Blocked signals can be sent, but will not be received/handled by the destination process until it unblocks the signal
  - Useful to make sure that a signal handler does not inappropriately interfere with the "regular" code of the receiver process

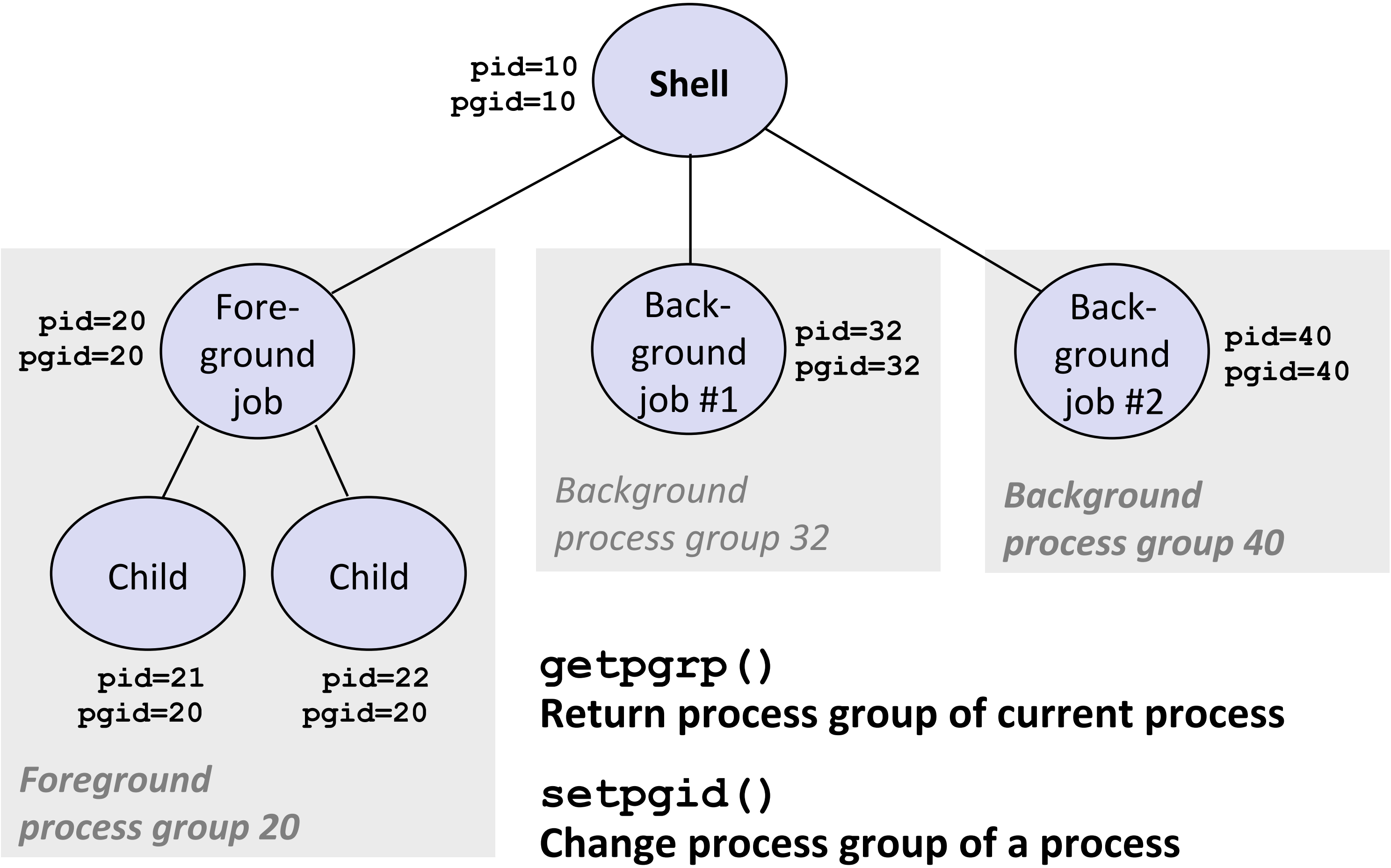
- A pending signal is received at most once
- Two particular signals: **`SIGKILL`** and **`SIGSTOP`**
  - For security reasons, these signal cannot be blocked and their default handler cannot be replaced

Signal concepts (continued)

- The kernel maintains **`pending`** and **`blocked`** bit vectors in the context of each (receiver) process P
  - **`pending`**: represents the set of pending signals
    - Kernel sets bit k in **`pending`** when a signal of type k is sent to process P
    - Kernel clears bit k in **`pending`** when a signal of type k is received by process P
  - **`blocked`**: represents the set of currently blocked signals
    - Can be set and cleared by using the **`sigprocmask`** function

Process groups

- Every process belongs to exactly one process group



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

Sending signals with the **kill** program

- **The kill** program allows sending an arbitrary signal to a process or process group (see **man 1 kill** for details)

- Examples

– **kill -9 24818**
  Send **SIGKILL** to process 24818

– **kill -9 -24817**
  Send **SIGKILL** to every process in process group 24817

```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24818 pts/2    00:00:02 forks
24819 pts/2    00:00:02 forks
24820 pts/2    00:00:00 ps
linux> kill -9 -24817
linux> ps
  PID TTY          TIME CMD
24788 pts/2    00:00:00 tcsh
24823 pts/2    00:00:00 ps
linux>
```

Sending signals with the `kill` function

(see **`man 2 kill`** for details)

```
void example()
{
    pid_t pid[N];
    int i, child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* Child infinite loop */

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

Receiving signals

- Suppose kernel is returning from an exception handler (e.g., a syscall handler) and is ready to pass control to process *p*

- Kernel computes **`pnb = pending & ~blocked`**
  - The set of pending nonblocked signals for process *p*

- If (**`pnb == 0`**)
  - Pass control to next instruction in the (user-level) logical flow for *p*
- Else
  - Choose least nonzero bit *k* in **`pnb`** and force process *p* to ***receive*** signal *k*
  - The receipt of the signal triggers some ***action*** by *p*
  - Repeat for all nonzero *k* in **`pnb`**
  - Pass control to next instruction in (user-level) logical flow for *p*

Default actions

- Each signal type has a predefined *default action/ handler*, which is one of:
  - The process terminates
  - The process terminates and dumps core
  - The process stops until restarted by a **`SIGCONT`** signal
  - The process ignores the signal

Installing signal handlers

- The **`signal`** function modifies the default action associated with the receipt of signal **`signum`**:
  - **`handler_t *signal(int signum, handler_t *handler)`**

- Different values for **`handler`**:
  - **`SIG_IGN`**: ignore signals of type **`signum`**
  - **`SIG_DFL`**: revert to the default action on receipt of signals of type **`signum`**
  - Otherwise, **`handler`** is the address of a ***signal handler***
    - Called when process receives signal of type **`signum`**
    - Referred to as ***"installing"*** the handler
    - Executing handler is called ***"catching"*** or ***"handling"*** the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal

- **Warning:** the **`signal`** function is not portable (different behavior on different Unix systems) and should be avoided – Preferably use **`sigaction`** instead

Installing signal handlers (continued)

```
int sigaction(int sig,
              const struct sigaction *act,
              struct sigaction *oldact)
```

```
struct sigaction {
   void    (*sa_handler)(int);    /* Address of handler */
   sigset_t sa_mask;              /* Signals blocked during handler
                                     invocation */
   int      sa_flags;             /* Flags controlling handler
                                    invocation */
   void    (*sa_restorer)(void); /* Not for application use */
};
```

See **`man 2 sigaction`** and **`man 7 signal`** for further details

Installing signal handlers (continued)

- In order to avoid the complexity of **`sigaction`**, the “CSAPP” textbook provides a wrapper function called **`Signal`** (beware of the case: **`signal`** vs. **`Signal`**)
  - **`handler_t *Signal(int signum, handler_t *handler)`**

- Same interface as **`signal`**, but portable because internally relies on **`sigaction`**

- Note that the list of blocked signals during handler invocation only contains signal **`signum`** (i.e., signals of same type)

- Code available in the same files as the RIO functions:
  - as part of the **`csapp.h`** and **`csapp.c`** files available from: http://csapp.cs.cmu.edu/public/code.html

Signal handler example

```
void chld_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    return;
}

int main()
{
    Signal(SIGCHLD, chld_handler);
    if (fork() == 0) {
        sleep(2);
        exit(0);
    } else {
        while (1);
    }
    exit(0);
}
```

Note: a signal handler must have the following signature:

- `void` return type
- a single input parameter of type `int`

```
linux> ./test
Process 24977 received signal 17
^C
linux>
```

User: `ctrl-c` (once)

The default `SIGINT` handler terminates the process

Signal handler example: a program that reacts to externally generated events (`ctrl-c`)

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

void handler(int sig) {
  printf("received SIGINT\n");
  sleep(2);
}

int main() {
  Signal(SIGINT, handler); /* installs ctl-c handler */
  while(1) {
  }
}
```

Note that **`main`** will not resume before the execution of the whole handler is completed

Signal handler example: a program that reacts to externally generated events (`ctrl-c`) - variant

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

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

int main() {
  Signal(SIGINT, handler); /* installs ctl-c handler */
  while(1) {
  }
}
```

Note that **`main`** will never resume because **`exit`** terminates the process

Signal handler example: a program that reacts to internally generated events

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  printf("BEEP\n");
  fflush(stdout);

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

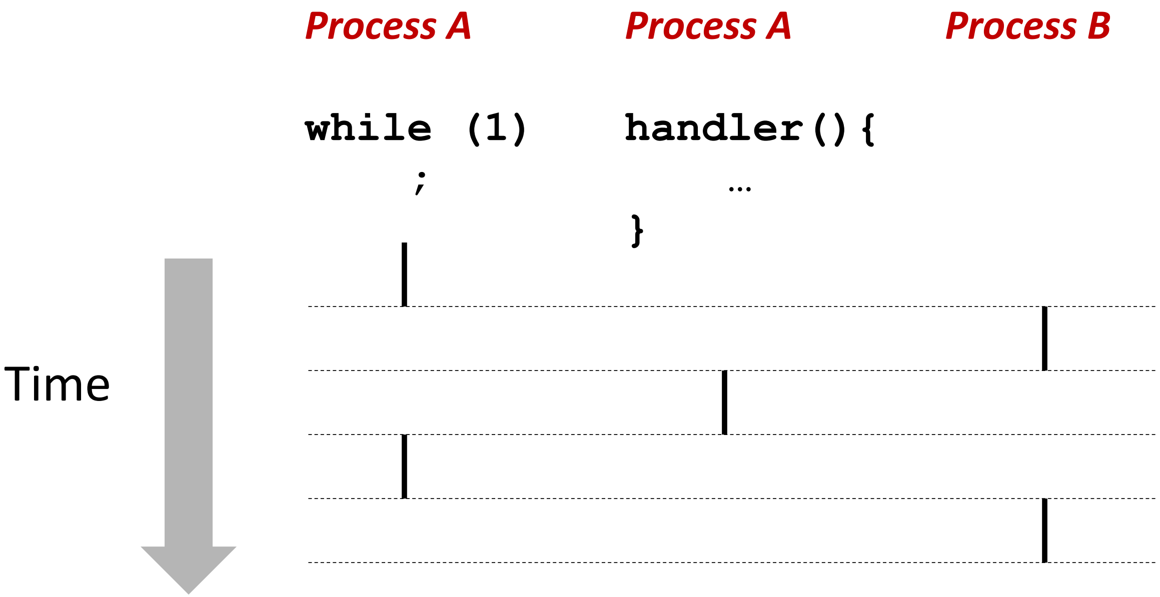
```
main() {
  Signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
linux>
```

Signal handlers as concurrent flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program
  - “*concurrently*” in the “*not sequential*” sense



Process A

```
while (1)
    ;
```

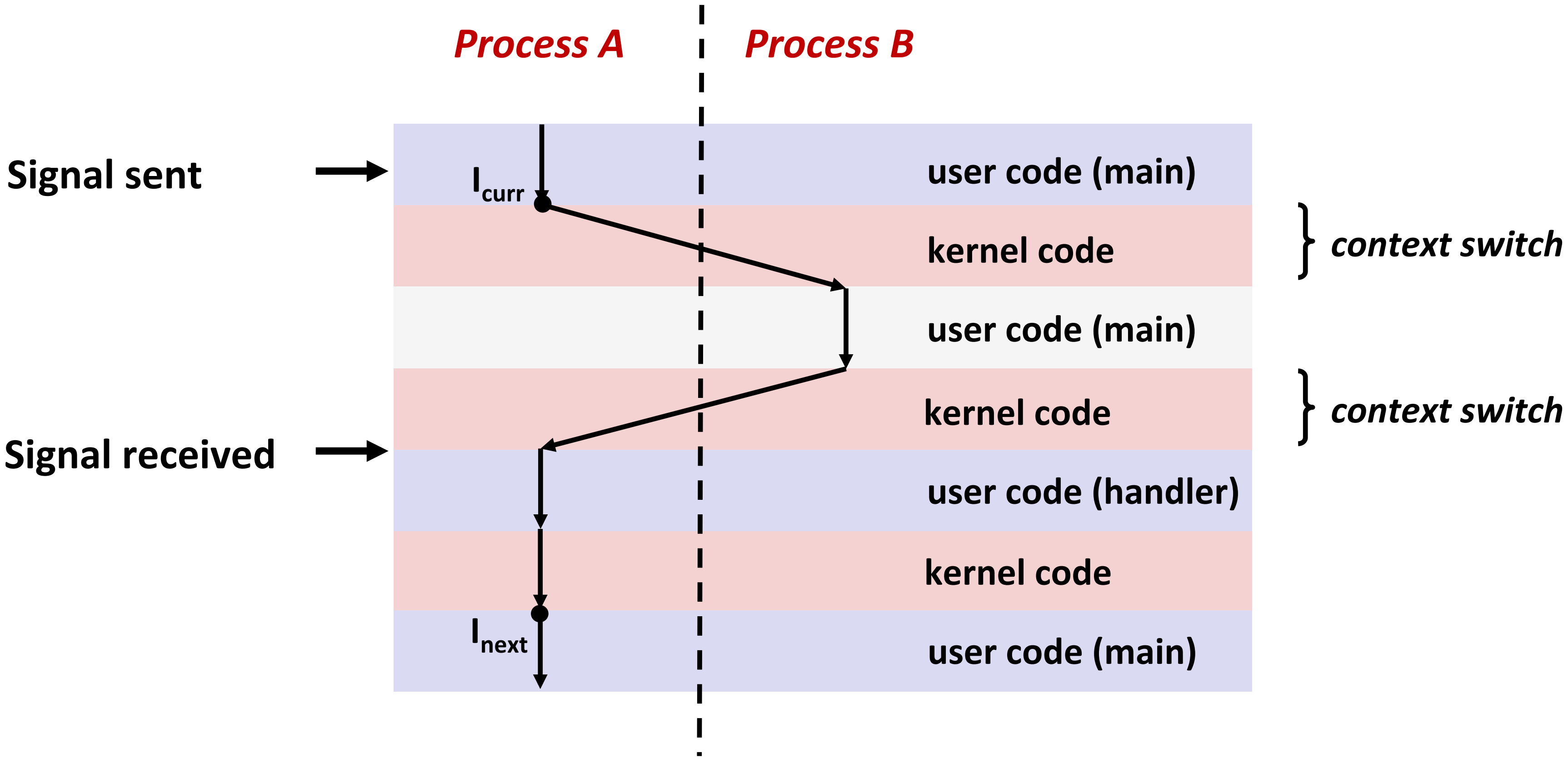
Process A

```
handler(){
    …
}
```

Process B

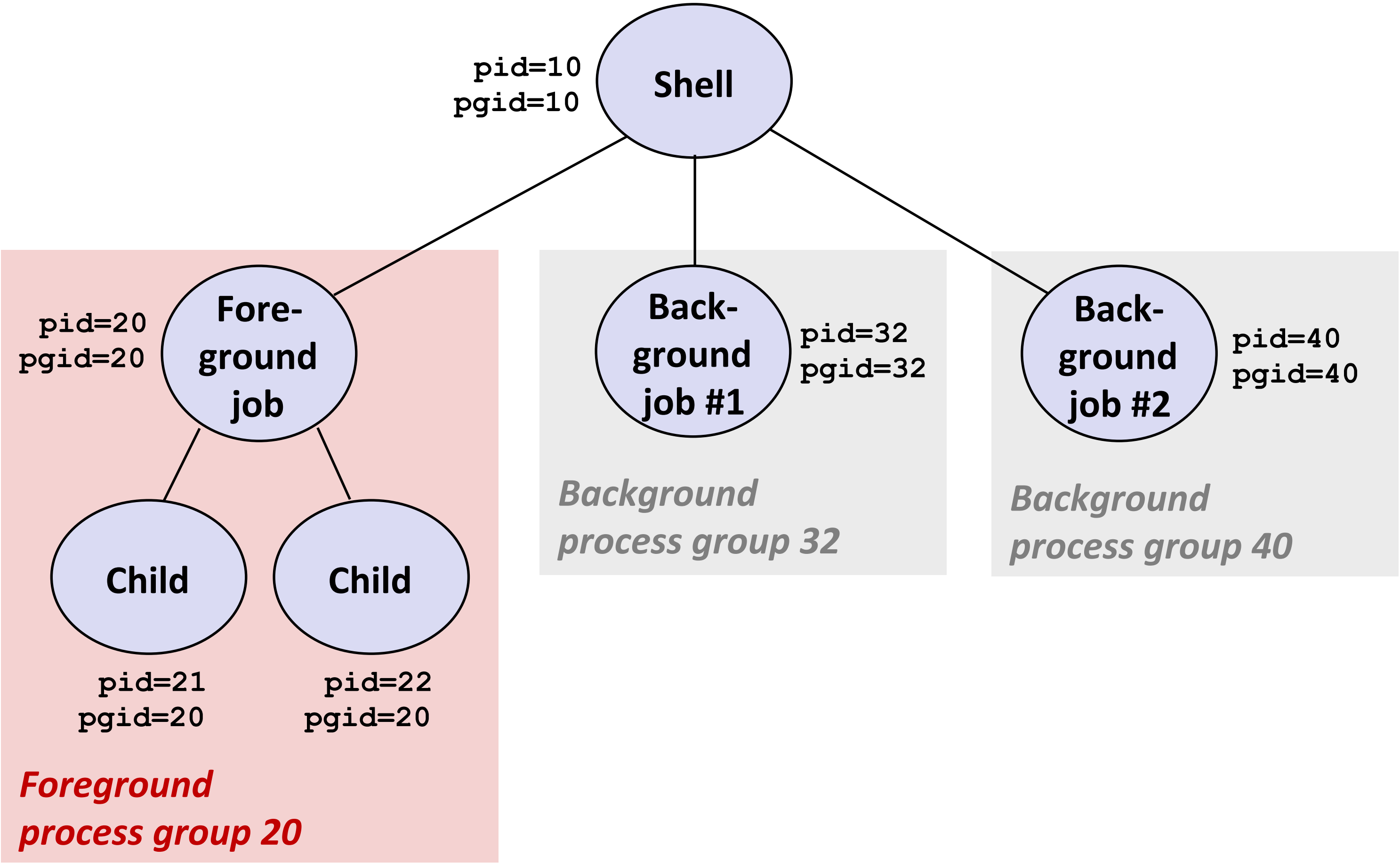
Time

Another view of signal handlers as concurrent flows



Sending signals from the keyboard

- Typing **`ctrl-c`** (**`ctrl-z`**) sends a **`SIGINT`** (**`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



Example of `ctrl-c` and `ctrl-z`

```
bluefish> ./forks 17
Child: pid=28108 pgrp=28107
Parent: pid=28107 pgrp=28107
<user types ctrl-z>

[1]+  Stopped
bluefish> ps w
  PID TTY       STAT   TIME COMMAND
27699 pts/8     Ss     0:00 -tcsh
28107 pts/8     T      0:01 ./forks 17
28108 pts/8     T      0:01 ./forks 17
28109 pts/8     R+     0:00 ps w
bluefish> fg %1
./forks 17
<user types ctrl-c>
bluefish> ps w
  PID TTY       STAT   TIME COMMAND
27699 pts/8     Ss     0:00 -tcsh
28110 pts/8     R+     0:00 ps w
```

job number

bring job on foreground

STAT (process state) Legend:

First letter:

S: sleeping

T: stopped

R: running

Second letter:

s: session leader

+: foreground proc group

See "**`man ps`**" for more details

Signal handler funkiness

```
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n",
           sig, pid);
}

void example()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    Signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
         if ((pid[i] = fork()) == 0) {
              sleep(1); /* deschedule child */
              exit(0);  /* Child: Exit */
         }
    while (ccount > 0)
         pause(); /* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - For a given recipient process, for each signal type, there is just a single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal

Dealing with nonqueuing signals

- Must check for all terminated jobs
  - Typically loop with **`waitpid`**

wait for any child process

non-blocking mode

```
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

void example()
{
    . . .
    Signal(SIGCHLD, child_handler2);
    . . .
}
```

Signal handler funkiness (continued)

- Signal arrival during long system calls (say a **`read`**)
- Signal handler interrupts **`read`** call
  - Linux: upon return from signal handler, the **`read`** call is restarted automatically
  - Some other flavors of Unix can cause the **`read`** call to fail with an **`EINTR`** error number (**`errno`**)
    In this case, the application program can restart the slow system call.
- Subtle differences like these complicate the writing of portable code that uses signals

Async-signal safety

- What happens if a process is interrupted by a signal in the middle of a call to a library/system function (e.g., **`malloc`**)?
  - The signal handler may also execute the same function or a related one (e.g., **`free()`**)
  - If the function(s) relies (rely) on global or static variables, the variables may become inconsistent because of interleaved streams of instructions

- Only a limited set of functions can be safely called from a signal handler
  - Such functions are called *“async-signal-safe”*
  - They either do not use global variables or are not interruptible by signals

Async-signal safety (continued)

- Notes:
  - The POSIX standard defines a list of 117 async-signal-safe functions
  - See **`man 7 signal`** for details
  - Examples
    - **`write`** is on the list
    - **`printf`** is not (the previous code examples are actually unsafe)

Summary about signals

- 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
  - Do not have queues
    - Just one bit for each pending signal type

Outline

- Signals
- Inter-Process Communication facilities

IPC facilities

- Unix systems provide a rich set of mechanisms for Inter-Process Communication (IPC)

- These facilities can be divided into three broad functional categories:
  - **Communication**: exchanging data between processes
  - **Synchronization**: synchronizing the actions of processes (or threads)
  - **Signals**: intended primarily for other purposes but can sometimes be used for synchronization (or, more rarely, for communication)
    - Standard signals (studied previously)
    - Real-time signals (not studied in this lecture)

IPC facilities (continued)

- For a given kind of IPC facility, a modern system will often provide several implementations, with slightly different APIs

- Why?
  - Similar facilities evolved on different Unix variants (e.g., System V, BSD) and later came to be ported to other Unix systems
  - New facilities have been developed to address design deficiencies in similar earlier facilities. For instance, the POSIX IPC facilities (message queues, semaphores and shared memory) were designed as an improvement on the older System V IPC facilities

Synchronization facilities

- Semaphores
- File locks
- Mutexes (for threads)
- Condition variables (for threads)

- These facilities will be studied in other lectures

Communication facilities

- **Shared memory**
  - Memory mappings (studied previously – see **`mmap()`**)
    - Mapped file
    - Anonymous mappings
  - Shared memory segments
- **Data transfers**
  - **Byte stream**
    - Pipes (studied previously)
    - FIFOs, a.k.a. “named pipes” (studied previously)
    - Stream sockets
  - **Messages**
    - Message queues
    - Datagram sockets

Communication facilities
Data transfers vs. shared memory

- **Data-transfer facilities**
  - One process writes data into the IPC facility, another process reads it
  - Require two transfers: from source to kernel, then from kernel to destination
  - Reads are “destructive”: a read operation consumes data, and that data is not available to any other process
  - Reader-writer synchronization is automatic: reader will block until some data becomes available

- **Shared memory**
  - Requires a single copy of the data (the kernel makes page-table entries in each process point to the same pages in RAM)
  - Data placed in shared memory is visible to all the processes that share that memory
  - Fast communication: once set up, a shared memory zone does not require data transfers, nor syscalls
  - ... But requires explicit synchronization (see details in a future lecture)

Data transfers
Subcategories

- **Byte stream**
  - Undelimited (“opaque”) byte stream
  - Each read operation may read an arbitrary number of bytes from the IPC facility, regardless of the size of the blocks written by the writer

- **Messages**
  - The data exchanged via messaging facilities takes the form of delimited messages. Each read operation reads a whole message, as written by the writer process
  - It is not possible to read part of a message, leaving the remainder on the IPC facility, nor is it possible to read multiple messages in a single read operation

Higher-level IPC facilities

- **Remote procedure call (RPC)**
  - Allows a “client” process to invoke the execution of a procedure in the context of a “server” process
  - Applicable to processes running on the same machine, but also to processes running on different machines
  - Typically built on top on other IPC facilities, such as sockets
  - Uses “stub” procedures on each side to hide the details from application code (e.g., packing parameters)
- **Remote method invocation (RMI)**
  - Mechanism similar to RPC in the context of the Java object-oriented system/ language
  - Allows a (Java) program running on one machine to invoke a method on an object within a program running on another machine
- **Message passing interface (MPI)**
  - Designed for communication and synchronization between multiple (potentially distributed) processes in a parallel application
  - Offers a large set of primitives for point-to-point and collective communications

References

- W. Richard Stevens & Stephen A. Rago, Advanced Programming in the Unix Environment, 2nd Edition, Addison Wesley, 2005

- Michael Kerrisk. The Linux Programming Interface (*a Linux and UNIX system Programming Handbook*). No Starch Press, 2010