Today

- System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- **Signals**
- **Non-Local Jumps**

ECF to the rescue!

- **Problem with the sample shell**
  - Background process is not communicating with shell once complete.
  - Shell is not reaping child, since it doesn't know when a background process completes.

- **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***

Signals

- **A *signal* is a small message that notifies a process that an event of some type has occurred in the system**
  - Similar to exceptions and interrupts
  - Sent from the kernel (sometimes at the request of another process) to a process
  - Signal type is identified by small integer ID's (1-30, + system dependent ones)
  - Only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	User typed ctrl-c
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

kill -l

- Returns a complete list of signals for your system

```
[sonmeza@cmsc257 ~]$ 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
```

- SIGRT versions became available after Linux 2.0
  - Real time, user defined signals
  - Have different features, e.g. can be queued, others not
  - Out of our scope

Signal Concepts: Sending a Signal

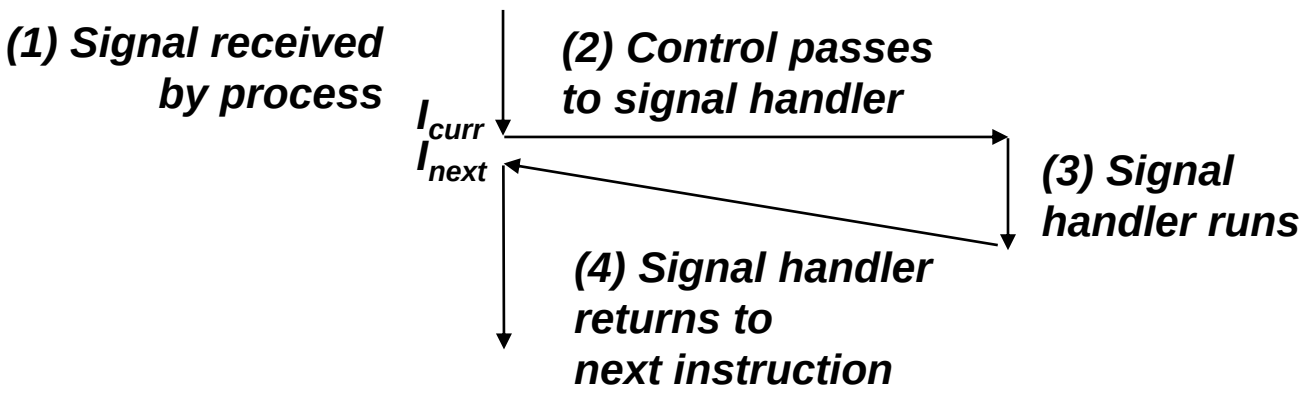
- **Kernel *sends* (delivers) a signal to a *destination process***

- **Kernel sends a signal for one of the following reasons:**
  - Kernel has detected a system event such as divide-by-zero (SIGFPE) or 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

Signal Concepts: Receiving a Signal

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

- **Some 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***



Signal Concepts: Pending and Blocked Signals

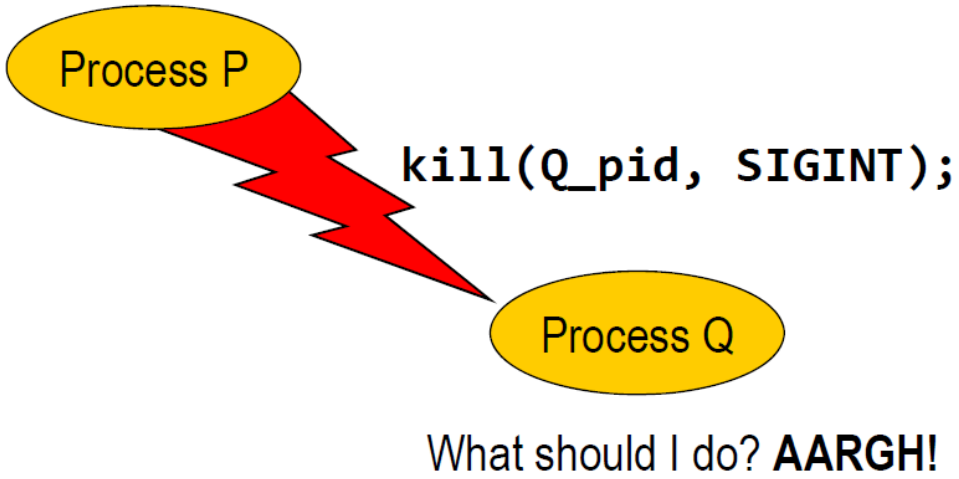
- **A signal is *pending* if sent but not yet received**
  - There can be at most one pending signal of any particula type
  - Important: Signals are not queued (Except Sigrt ones)
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- **A process can *block* the receipt of certain signals**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

Sending a signal

- A process can invoke kill() to send a signal to another process
  - kill() has two arguments
    - the ***process id*** of the receiving process
    - a ***signal name*** or a ***signal number***
- **#include <signal.h>**

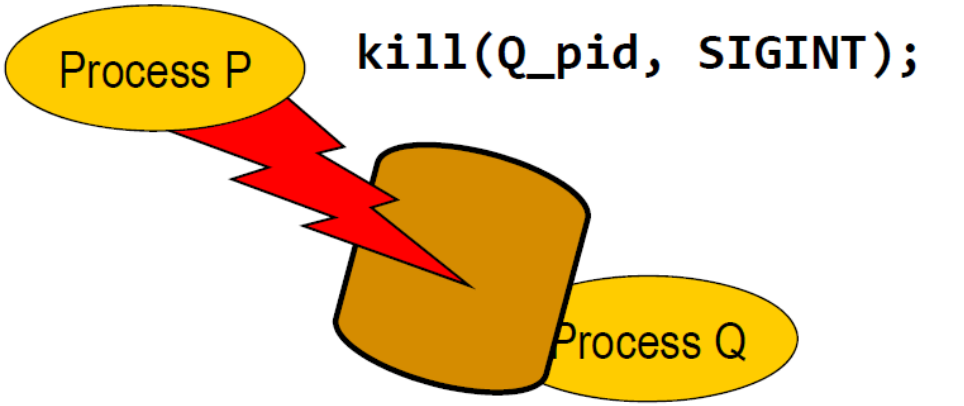
  kill(this_pid, this_signal);
- Process receiving the signal will terminate if default action is terminate and signal is not handled



Catching a signal

- The process receiving signal can ***catch*** it by using **signal()**

- **signal(a_signal, catch_it);**
  - where catch_it points to a function that will be called whenever a signal a_signal is received.
- The 9th and 19th signals, **SIGKILL and SIGSTOP**, cannot be caught or blocked.



Process is now **shielded** by **signal()** call

Sending Signals with `/bin/kill` Program

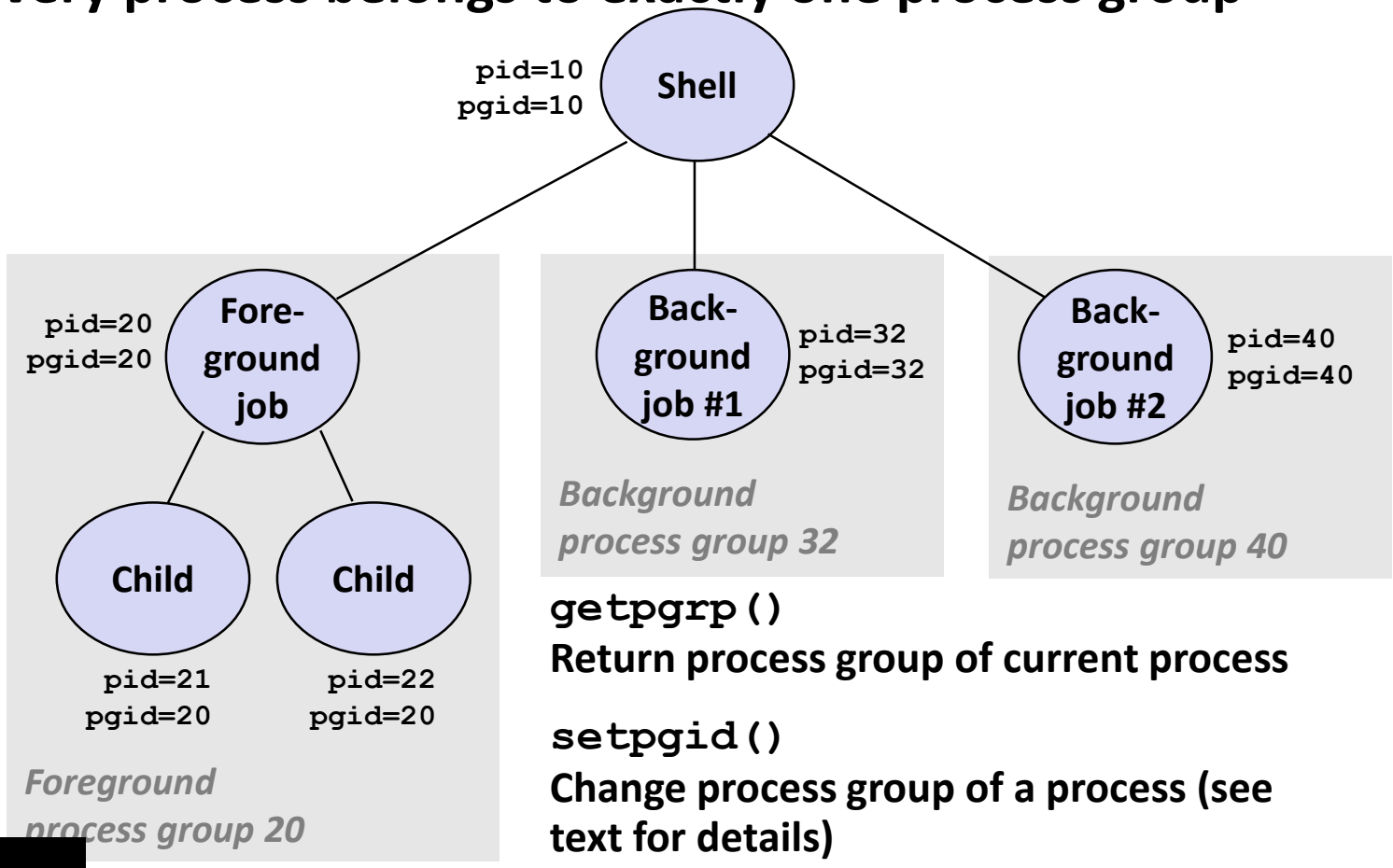
- **`/bin/kill` program sends arbitrary signal to a process or process group**

- **Examples**
  - **`/bin/kill -9 24818`**
    Send SIGKILL to process 24818
  - **`/bin/kill -9 -24817`**
    Send SIGKILL to every process in process group 24817
  - **`/bin/kill -9 0`**
    Send SIGKILL to every process in current processes group(including shell)

```
linux> ./forks 16
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> /bin/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 to Process Groups

- **Every process belongs to exactly one process group**



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process (see text for details)

```
ps -fj
UID          PID    PPID    PGID     SID  C STIME TTY          TIME CMD
sonmeza   117289  117288  117289  117289  0 13:07 pts/20   00:00:00 -bash
sonmeza   120926  117289  120926  117289  0 13:43 pts/20   00:00:00 ps -fj
```

Process group id

- The getpgrp function returns the process group ID of the current process.

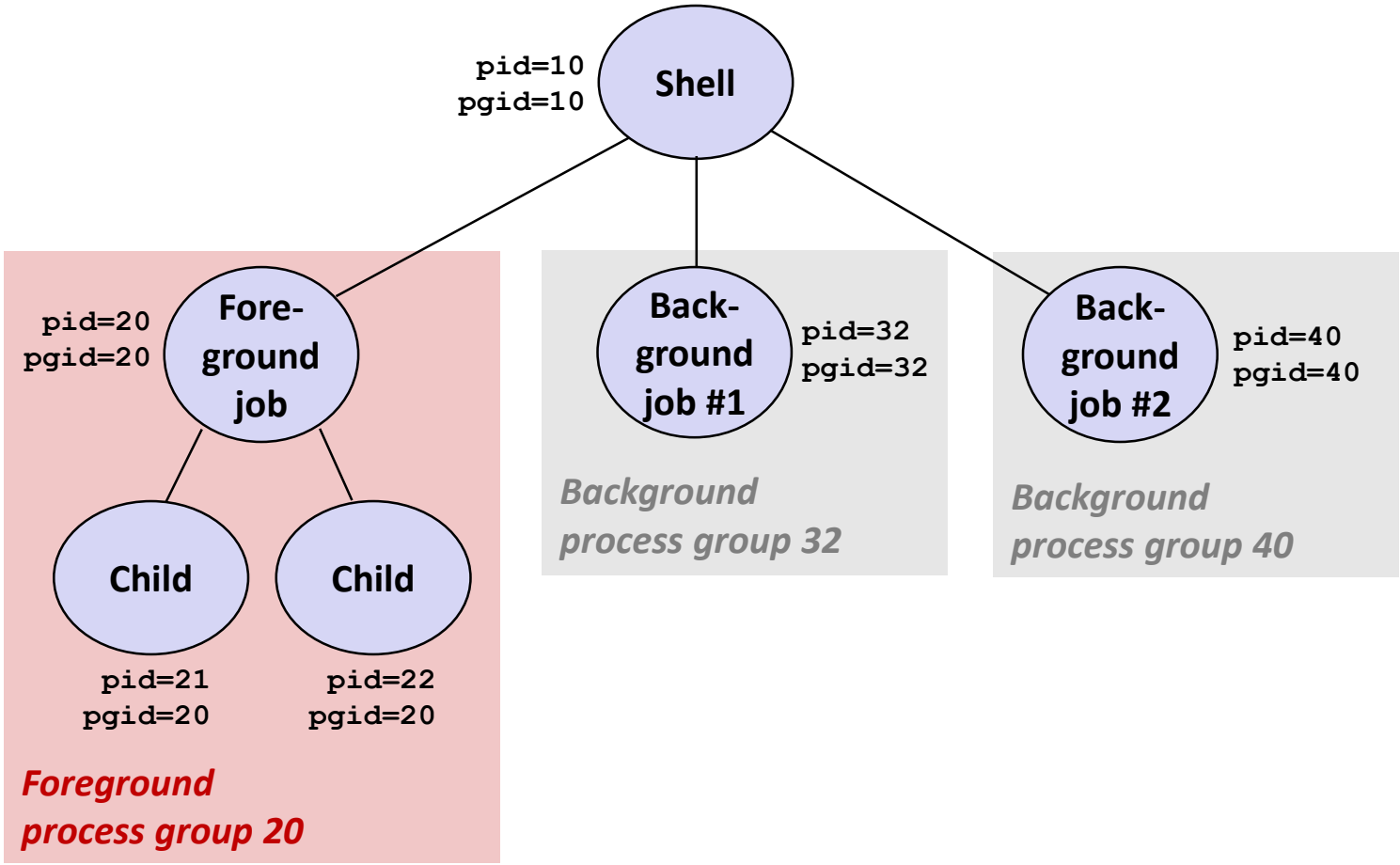
```
#include <unistd.h>

pid_t getpgrp(void);
```

- By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the setpgid function:

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Sending Signals from the Keyboard

- **Typing ctrl-c (ctrl-z) causes the kernel to send a SIGINT (SIGTSTP) to every job in the foreground process group.**
  - SIGINT (Signal 2) – default action is to terminate each process
  - SIGTSTP (Signal 20)– 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
<types ctrl-z>
Suspended
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

<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
```

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

fg will run the current job in the foreground

Sending SIGCONT after stopping

```
[sonmeza@cmsc257 ~]$ ./infinite
^Z
[1]+  Stopped                 ./infinite
[sonmeza@cmsc257 ~]$ ps w
   PID TTY      STAT   TIME COMMAND
 51464 pts/0    Ss     0:00 -bash
 52037 pts/0    T      0:02 ./infinite
 52052 pts/0    R+     0:00 ps w
[sonmeza@cmsc257 ~]$ kill -18 52037
[sonmeza@cmsc257 ~]$ ps w
   PID TTY      STAT   TIME COMMAND
 51464 pts/0    Ss     0:00 -bash
 52037 pts/0    R      0:05 ./infinite
 52109 pts/0    R+     0:00 ps w
[sonmeza@cmsc257 ~]$
```

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 18 is the SIGCONT signal

kill function

Processes send signals to other processes (including themselves) by calling the kill function.

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

int kill(pid_t pid, int sig);
```

```
1   #include "csapp.h"
2
3   int main()
4   {
5       pid_t pid;
6
7       /* Child sleeps until SIGKILL signal received, then dies */
8       if ((pid = Fork()) == 0) {
9           Pause();  /* Wait for a signal to arrive */
10          printf("control should never reach here!\n");
11          exit(0);
12      }
13
14      /* Parent sends a SIGKILL signal to a child */
15      Kill(pid, SIGKILL);
16      exit(0);
17  }
```

Kill is wrapper of kill in csapp.c

pause() causes the calling process (or thread) to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function.(From linux man pages)

http://csapp.cs.cmu.edu/3e/code.html

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 user-level ***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

Signal Handling Example

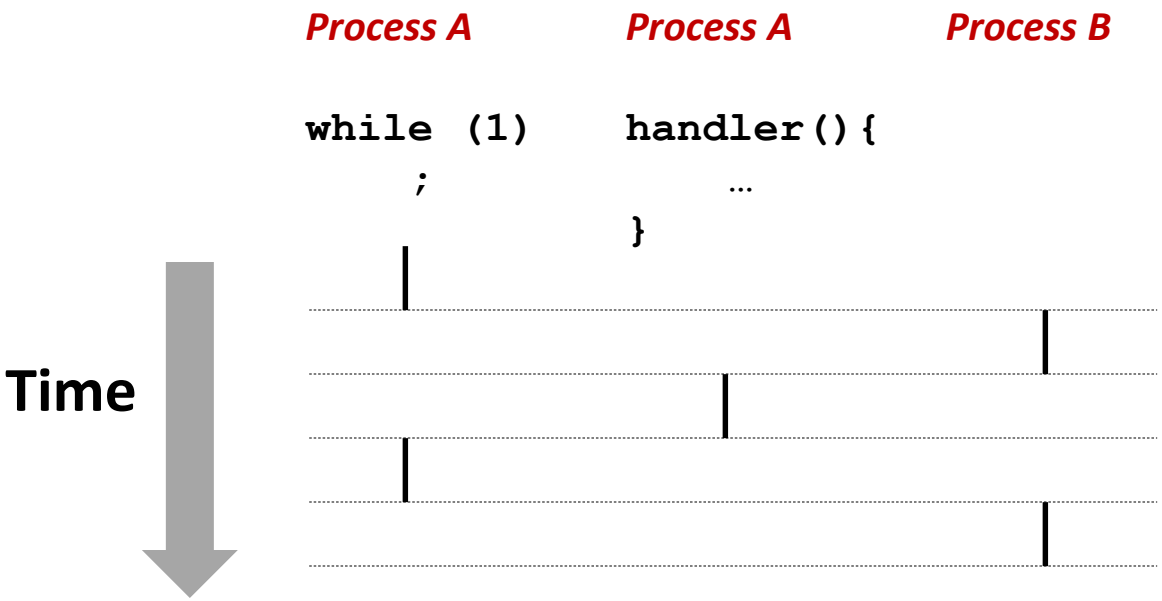
```
void sigint_handler(int sig) /* SIGINT handler */
{
   printf("So you think you can stop the bomb with ctrl-c, do you?\n");
   sleep(2);
   printf("Well...");
   fflush(stdout);
   sleep(1);
   printf("OK. :-)\n");
   exit(0);
}

int main()
{
   /* Install the SIGINT handler */
   if (signal(SIGINT, sigint_handler) == SIG_ERR)
      unix_error("signal error");

   /* Wait for the receipt of a signal */
   pause();

   return 0;
}
```

```
[sonmeza@cmsc257 ecf]$ ./sigint
^CSo you think you can stop the bomb with ctrl-c, do you?
Well..OK. :)
[sonmeza@cmsc257 ecf]$
```

sleep() causes the calling thread to sleep either until the number of real-time seconds specified in *seconds* have elapsed or until a signal arrives which is not ignored. (Linux man pages)

Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program



Safe Signal Handling

- **Handlers are tricky because they are concurrent with main program and share the same global data structures.**
  - Shared data structures can become corrupted.
- **Concurrency comes with issues.**
- **For now here are some guidelines to help you avoid trouble.**

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., Handler might simply set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - `printf`, `sprintf`, `malloc`, and `exit` are not safe!
- **Refer to the following link for async-signal-safe functions**

http://man7.org/linux/man-pages/man7/signal-safety.7.html

Safely Generating Formatted Output

- **Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.**
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)   /* Put long */`
  - `void sio_error(char s[])   /* Put msg & exit */`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
   Sio_puts("So you think you can stop the bomb with ctrl-c, do you?\n");
   sleep(2);
   Sio_puts("Well...");
   sleep(1);
   Sio_puts("OK. :-)\n");
   _exit(0);
}
```

sigintsafe.c

Async-Signal-Safety

- **Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.**
- **Posix guarantees 117 functions to be async-signal-safe**
  - Source: "`man 7 signal`" http://man7.org/linux/man-pages/man7/signal-safety.7.html
  - Popular functions on the list:
    - `write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** on the list:
    - `printf, sprintf, malloc, exit`
    - **Unfortunate fact: `write` is the only async-signal-safe output function**

Guidelines for Writing Safe Handlers

- **G2: Save and restore `errno` on entry and exit**
  - So that other handlers don't overwrite your value of `errno`
  - Asnyc-signal safe functions may change the value of errno. If other parts of the program rely on errno, this will cause problem.

```
void handler(int sig)
{
   int olderrno = errno;

  // Some handling operation

   errno = olderrno;
}
```

- **G3: Beware that signals are not queued**
  - There can be only one signal of same type pending, after receiving one

Wrong Signal Handling

- **Pending signals are not queued**
  - For each signal type, one bit indicates whether or not signal is pending…
  - …thus at most one pending signal of any particular type.
  - First signal will be handled, second one will be pending first then will be handled, third will not be received
- **You can't use signals to count events, such as children terminating.**

```c
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}
```

```c
void fork14() {
    pid_t pid[3];
    int i;
    ccount = 3;
    signal(SIGCHLD, child_handler);

    for (i = 0; i < 3; i++) {
        if ((pid[i] = Fork()) == 0) {
            printf("Hello From Child %d\n", (int) getpid());
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) /* Parent spins */ ;
}
```

```
linux> ./signal1
Hello from child 10320
Hello from child 10321
Hello from child 10322
Handler reaped child 10320
Handler reaped child 10322
```

```
<ctrl-z>
Suspended
linux> ps
PID TTY STAT TIME COMMAND
...
10319  p5 T    0:03 signal1
10321  p5 Z    0:00 signal1 <defunct>
10323  p5 R    0:00 ps
```

Better Signal Handling

- **Must wait for all terminated child processes**
  - Put `wait` in a loop to reap all terminated children

```
void child_handler2(int sig)
{
   int olderrno = errno;
   pid_t pid;
   while ((pid = wait(NULL)) > 0) {
      ccount--;
      Sio_puts("Handler reaped child ");
      Sio_putl((long)pid);
      Sio_puts(" \n");
   }
   if (errno != ECHILD)
      Sio_error("wait error");
   errno = olderrno;
}
```

```
linux> ./signal2
Hello from child 10378
Hello from child 10379
Hello from child 10380
Handler reaped child 10379
Handler reaped child 10378
Handler reaped child 10380
```

Portable Signal Handling

- **Ugh! Different versions of Unix can have different signal handling semantics**
  - Some older systems restore action to default after catching signal
  - Some system calls might be interrupted, may return instead of resuming in handler
  - Some systems don't block signals of the type being handled
- **Solution: Posix library standard `sigaction` function**
  - It is standard but complicated

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act,
                          struct sigaction *oldact);
                                Returns: 0 if OK, −1 on error
```

Portable Signal Handling

- **Textbook provides a wrapper within csapp.c**
  - Which is called the same way as signal

```
handler_t *Signal(int signum, handler_t *handler)
{
   struct sigaction action, old_action;

   action.sa_handler = handler;
   sigemptyset(&action.sa_mask); /* Block sigs of type being handled */
   action.sa_flags = SA_RESTART; /* Restart functions if interrupted by handler*/

   if (sigaction(signum, &action, &old_action) < 0)
      unix_error("Signal error");
   return (old_action.sa_handler);
}
```

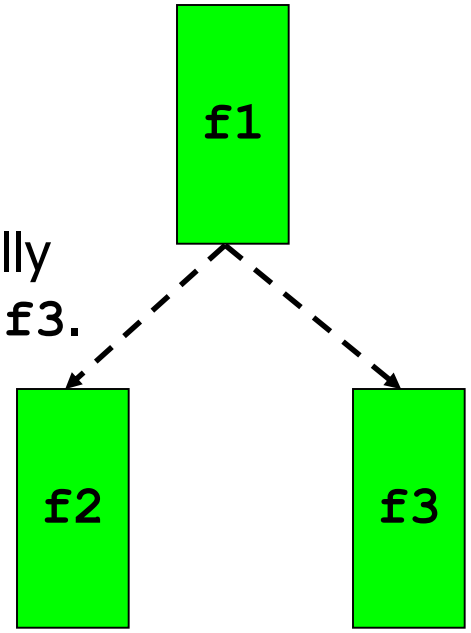
Today

- System call error handling
- More on fork System Call and Process Graphs
- wait and waitpid system calls
- exec family system calls
- Signals
- **Non-Local Jumps**

Other Types of Exceptional Control Flow

- **Non-local Jumps**
  - C mechanism to transfer control to any program point higher in the current stack

`f1` eventually calls `f2` and `f3`.



When can non-local jumps be used:

- **Yes: f2 to f1**
- **Yes: f3 to f1**

- **No: f1 to either f2 or f3**
- **No: f2 to f3, or vice versa**

Non-local Jumps

- **`setjmp()`**
  - Identify the current program point as a place to jump to
- **`longjmp()`**
  - Jump to a point previously identified by `setjmp()`

Nonlocal Jumps: `setjmp/longjmp`

- **Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location**
  - Controlled way to break the procedure call / return discipline
  - Useful for error recovery and signal handling

- **`int setjmp(jmp_buf j)`**
  - Must be called before longjmp
  - Identifies a return site for a subsequent longjmp
  - Called **once**, returns **one or more** times

- **Implementation:**
  - Remember where you are by storing the current ***register context***, ***stack pointer***, and ***PC value*** in **`jmp_buf`**
  - Return 0

`setjmp/longjmp` (cont)

- **`void longjmp(jmp_buf j, int i)`**
  - Meaning:
    - return from the **`setjmp`** remembered by jump buffer **`j`** again ...
    - … this time setjmp will return **`i`** instead of 0 at return
  - Called after **`setjmp`**
  - Called **once**, but **never** returns

- **`longjmp` Implementation:**
  - Restore register context (stack pointer, base pointer, PC value) from jump buffer **`j`**
  - Set the return value of setjmp to **`i`**
  - Jump to the location indicated by the PC stored in jump buf **`j`**

Non-local Jumps

- **From the UNIX `man` pages:**

longjmp() and siglongjmp() make programs hard to understand and maintain. If possible an alternative should be used.

sig versions are signal versions

Non-local Jumps: Example

```
#include <setjmp.h>

jmp_buf  buf;

int main(void)
{
   if (setjmp(buf) == 0)
       printf("First time through.\n");
   else
       printf("Back in main() again.\n");

   f1();
}
```

```
f1()
{
   …
   f2();
   …
}

f2()
{
   …
   longjmp(buf, 1);
   …
}
```

Putting It All Together: A Program That Restarts Itself When `ctrl-c`'d

```
#include "csapp.h"

sigjmp_buf buf;

void handler(int sig)
{
   siglongjmp(buf, 1);
}

int main()
{
   if (!sigsetjmp(buf, 1)) {
      Signal(SIGINT, handler);
      Sio_puts("starting\n");
   }
   else
      Sio_puts("restarting\n");

   while(1) {
                Sleep(1);
                Sio_puts("processing...\n");
   }
   exit(0); /* Control never reaches here */
}
```

Returns 0 at first call, returns 1 at subsequent calls

```
greatwhite> ./restart
starting
processing...
processing...
processing...
restarting
processing...
processing...
restarting
processing...
processing...
processing...
```

Ctrl-c

Ctrl-c

restart.c

Try catch in C

- A sample implementation

- http://www.di.unipi.it/~nids/docs/longjump_try_trow_catch.html

Summary: Signals & Jumps

- **Signals – 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
    - Don't have queues
      - Just one bit for each pending signal type
- **Non-local jumps – exceptional control flow within process**
  - Within constraints of stack discipline

The good and bad news

- **Good news:**
  - Server can now handle several user requests in parallel
- **Bad news:**
  - fork() is a **very expensive** system call
  - Communicating between processes is costly because most communication goes through the OS
- **OS trick**
  - OS have **Copy on Write** mechanism to reduce the cost, but still costly

Copy on Write

- **Inefficient to physically copy memory from parent to child**
  - Code (text section) remains identical after fork
  - Portions of data section, heap, and stack may remain identical after fork
- **Copy-On-Write**
  - OS memory management policy to lazily copy pages only when they are modified
  - Initially map same physical page to child virtual memory space (but in read mode)
  - Write to child virtual page triggers page protection violation (exception)
  - OS handles exception by making physical copy of page and remapping child virtual page to that page

A more efficient solution

- **Provide a faster mechanism for creating cheaper processes:**
  - ***Threads***

- ***How?***
  - ***Threads share the address space of their parent***
  - ***No need to create a new address space as in fork() system call***