CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 9. 시그널
교재 8.5
Nov. 9, 2010
http://eslab.cnu.ac.kr

폭탄 Lab - Hall of Fame

- 2006 년도
- 유병성, 이경준, 나경호, 주은종, 송영진, 이기정, 조운형, 전준희
- 2007 년도
- 박진수, 오준호, 정훈민, 조으뜸, 정광운, 장동현, 백인철, 박종현
- 2009 년도
- 임형빈, 전용환, 양윤서, 이동규
- 손정규, 이호진

오늘의 주제

- Signals
  - 커널 소프트웨어
- Long jumps
  - 응용 프로그램
- More on signals

Background Job

- Users generally run one command at a time
  - Type command, read output, type another command
- Some programs run “for a long time”
  - Example: “delete this file in two hours”
  - % sleep 7200; rm/tmp/junk # shell stuck for 2 hours
- A “background” job is a process we don't want to wait for
  - % (sleep 7200 ; rm/tmp/junk) &
  - [1] 907
  - % # ready for next command

간단한 쉘 구현의 문제점

- Shell correctly waits for and reaps foreground jobs.
- But what about background jobs?
  - **Will become zombies when they terminate.**
  - **Will never be reaped because shell (typically) will not terminate.**
  - **Creates a memory leak that will eventually crash the kernel when it runs out of memory.**
  - **Modern Unix: once you exceed your process quota, your shell can't run any new commands for you: fork() returns -1**
- Solution
  - **Reaping background jobs requires a mechanism called a *signal*.**

시그널 – 상위수준의 예외적 제어 흐름

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - **Kernel abstraction for exceptions and interrupts.**
  - **Sent from the kernel (sometimes at the request of another process) to a process.**
  - **Different signals are identified by small integer ID's (1-30)**
  - **The only information in a signal is its ID and the fact that it arrived.**

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
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

시그널의 개념

Sending a signal

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

시그널의 개념(continued)

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.**
- **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 a signal handler.
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt.
    - 여기서 질문 하나. 인터럽트 발생시에 CPU가 어떻게 동작하는가?

시그널의 개념(continued)

Signal semantics

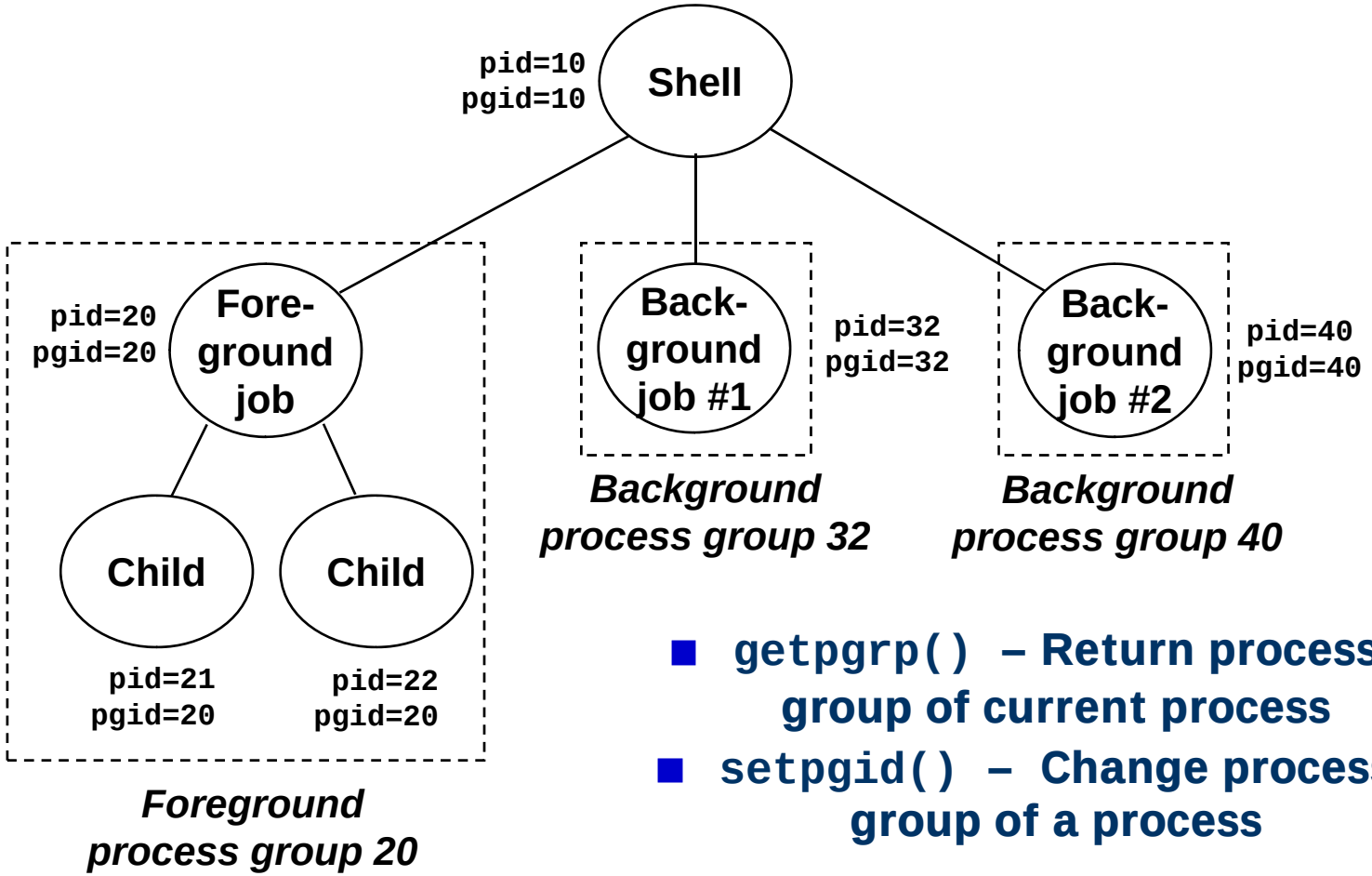
- **A signal is *pending* if it has been sent but not yet received.**
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- **A process can *block* the receipt of certain signals.**
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
  - There is one signal that can not be blocked by the process. (SIGKILL)
- **A pending signal is received at most once.**
  - Kernel uses a bit vector for indicating pending signals.

시그널의 개념- Implementation

- Kernel maintains `pending` and `blocked` bit vectors in the context of each process.
  - **`pending` – represents the set of pending signals**
    - Kernel sets bit k in `pending` whenever a signal of type k is delivered.
    - Kernel clears bit k in `pending` whenever a signal of type k is received
  - **`blocked` – represents the set of blocked signals**
    - Can be set and cleared by the application using the `sigprocmask` function.

프로세스 그룹

- Every process belongs to exactly one process group identified by process group ID



- `getpgrp()` – Return process group of current process
- `setpgid()` – Change process group of a process

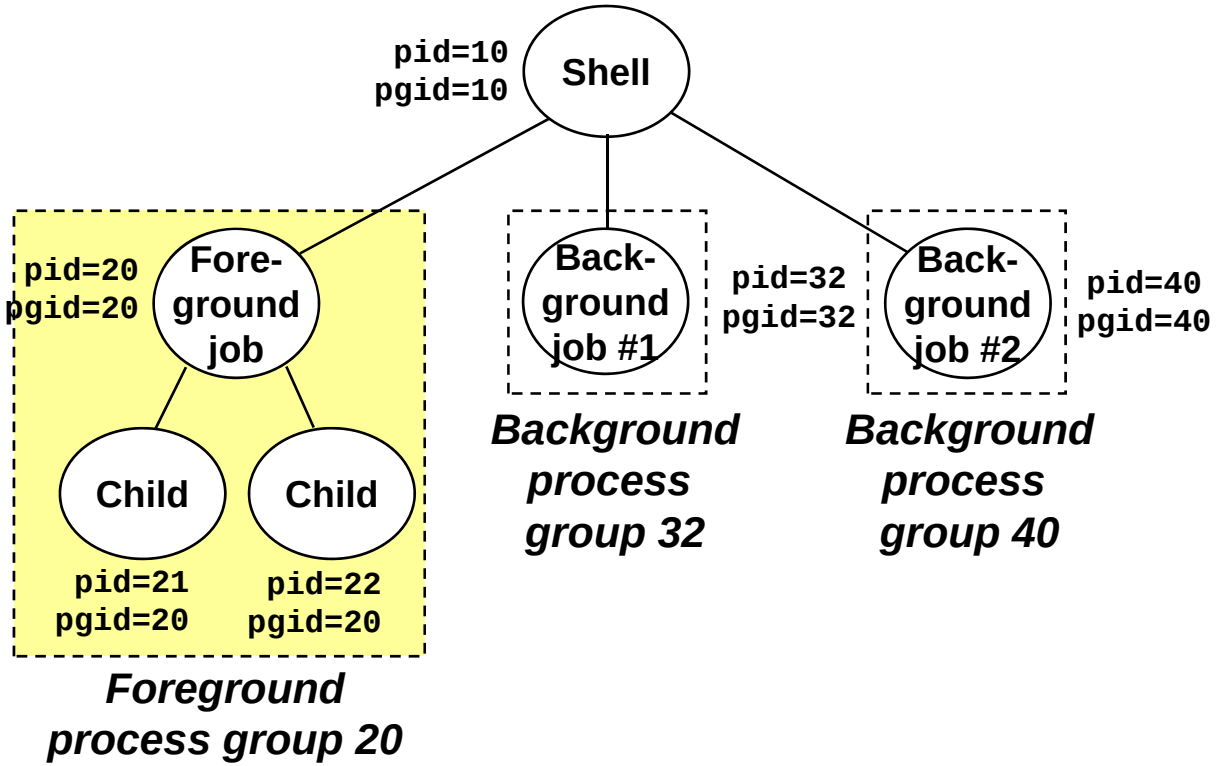
kill 프로그램을 이용한 시그널 보내기

- kill program sends arbitrary signal to a process or process group
- Examples
  - **kill –9 24818**
    - Send SIGKILL to process 24818
  - **kill –9 –24817**
    - Send SIGKILL to every process in process group 24817.

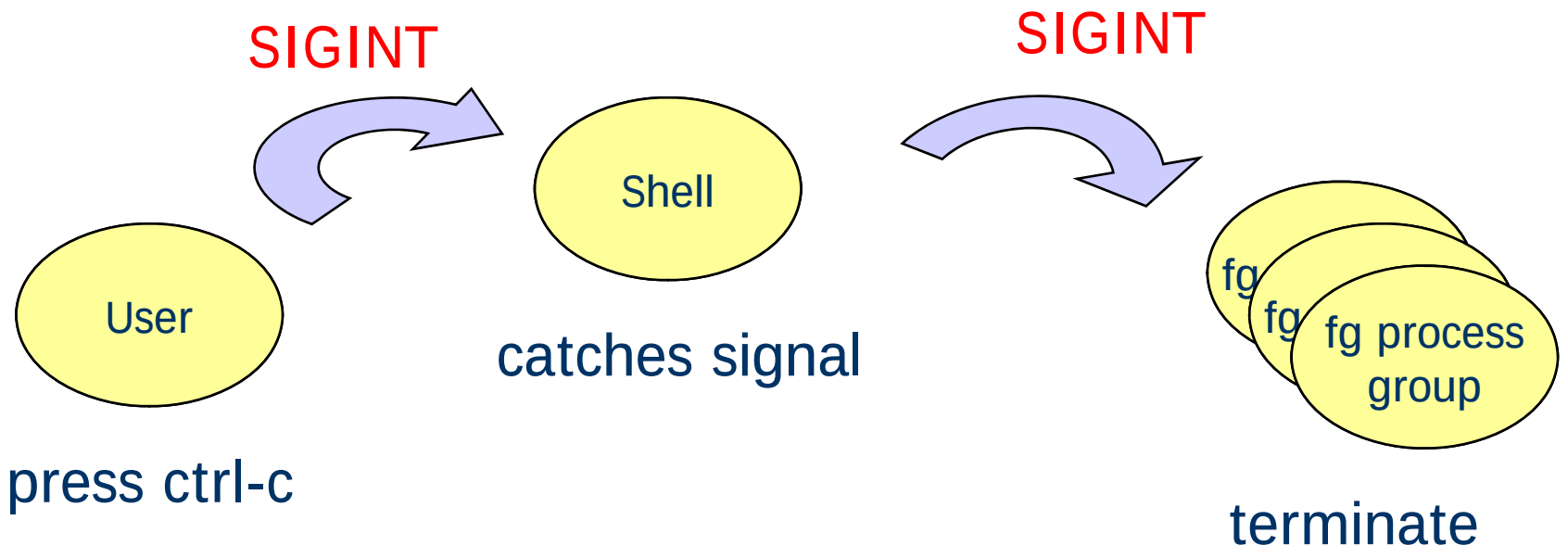
```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817

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>
```

키보드로부터 시그널 보내기

- 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**



키보드에서 CTRL-C 의 처리



Example of ctrl-c and ctrl-z

```
linux> ./forks 17
Child: pid=24868 pgrp=24867
Parent: pid=24867 pgrp=24867
<typed ctrl-z>
Suspended
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24867 pts/2    T      0:01 ./forks 17
24868 pts/2    T      0:01 ./forks 17
24869 pts/2    R      0:00 ps a
bass> fg
./forks 17
<typed ctrl-c>
linux> ps a
  PID TTY      STAT   TIME COMMAND
24788 pts/2    S      0:00 -usr/local/bin/tcsh -i
24870 pts/2    R      0:00 ps a
```

kill 함수를 이용해서 시그널 보내기

Demo sigtest

```
void fork12()
{
    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);
    }
}
```

시그널 받기

- Suppose kernel is returning from an exception 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 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 logical flow for *p*.**

기본동작(Default Actions)

- Each signal type has a predefined *default action*, 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.**
- The default action can be modified using *signal( )* function except SIGSTOP and SIGKILL

시그널 핸들러의 설치

- 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.

시그널 핸들러 예제

```
void int_handler(int sig)
{
    printf("Process %d received signal %d\n",
            getpid(), sig);
    exit(0);
}

void fork13()
{
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);

    . . .
}
```

Demo sigtest2

```
linux> ./forks 13
Killing process 24973
Killing process 24974
Killing process 24975
Killing process 24976
Killing process 24977
Process 24977 received signal 2
Child 24977 terminated with exit status 0
Process 24976 received signal 2
Child 24976 terminated with exit status 0
Process 24975 received signal 2
Child 24975 terminated with exit status 0
Process 24974 received signal 2
Child 24974 terminated with exit status 0
Process 24973 received signal 2
Child 24973 terminated with exit status 0
linux>
```

Normal exit ?

시그널 핸들러의 이상동작

```
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);
    sleep(2);
}

void fork14()
{
    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);
            /* Child: Exit */
            exit(0);
        }
    while (ccount > 0)
        pause();/* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - **For each signal type, just have single bit indicating whether or not signal is pending**
  - **Even if multiple processes have sent this signal**

Can you see the problem ?

큐를 사용하지 않는 문제점의 해결

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

```
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 fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

wait for all child processes

return immediately with return 0 if none of the child terminated

Demo : sigtest4

시그널 핸들러의 이상동작 (Cont.)

- 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 `EINTER` 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.

외부에서 생성된 이벤트를 처리하는 프로그램 (ctrl-c)

```
#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);
}

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

내부에서 발생된 이벤트를 처리하는 프로그램

```
#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!
bass>
```

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**
  - 여기서 질문:이것은 언제 쓰는 것일까?
  - **immediate return from deep nested calls**
- `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.**
  - **First returns 0**

setjmp/longjmp (cont)

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

- `longjmp` Implementation:
  - **Restore register context from jump buffer `j`**
  - **Set %eax (the return value) to `i`**
  - **Jump to the location indicated by the PC stored in jump buf `j`.**

setjmp/longjmp Example

```
#include <setjmp.h>
jmp_buf buf;

main() {
   if (setjmp(buf) != 0)
      printf("back in main due to an error\n");
   else
      printf("first time through\n");
   p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```

Demo : nonlocal.c

배운 지식 총 적용 : ctrl-c 발생시 재시동하는 프로그램의 작성

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

sigjmp_buf buf;

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

main() {
  signal(SIGINT, handler);

  if (!sigsetjmp(buf, 1))
    printf("starting\n");
  else
    printf("restarting\n");
```

```
while(1) {
    sleep(1);
    printf("processing...\n");
  }
}
```

```
bass> a.out
starting
processing...
processing...
restarting
processing...
processing...
restarting
processing...
```

Ctrl-c

Ctrl-c

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