Introduction to Computer Systems
15-213/18-243, spring 2009
16th Lecture, Mar. 17th

Instructors:
Gregory Kesden and Markus Püschel
Signals

- Kernel $\rightarrow$ Process
- Process $\rightarrow$ Process (using `kill`)
- 32 types of signals
- Sent by updating bit in `pending` vector
- You can write your own signal handlers

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., <code>ctl-c</code> from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Signal Handlers as Concurrent Flows

Process A

Signal delivered

\[ I_{\text{curr}} \]

user code (main)

kernel code

user code (main)

kernel code

user code (handler)

kernel code

user code (main)

Signal received

\[ I_{\text{next}} \]

Process B

\}\text{context switch}

\}\text{context switch}
Today

- More on signals
- Long jumps
- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
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

```
<table>
<thead>
<tr>
<th>Process Group</th>
<th>PID</th>
<th>PGID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreground</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Background #1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Background #2</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>
```

**Diagram:**
- Shell
  - Foreground job
    - Child
      - `pid=21` `pgid=20`
    - Child
      - `pid=22` `pgid=20`
  - Background job #1
    - `pid=32` `pgid=32`
  - Background job #2
    - `pid=40` `pgid=40`
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
./forks 17
<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
```
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

```c
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 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); /* deschedule child */
            exit(0); /* Child: Exit */
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
```
Living With Nonqueueing Signals

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

```c
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);
    . . .
}
```
Signal Handler Funkiness (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
A Program That Reacts to Externally Generated Events (Ctrl-c)

```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 ctrl-c handler */
    while(1) {
    }
}
```
A Program That Reacts to Internally Generated Events

```c
#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);
    }
}
```

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

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

`Linux> a.out`
A Program That Reacts to Internally Generated Events

```c
#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 to process in 1 second */
    while (1) {
        /* handler returns here */
    }
}
```

```bash
linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```
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
Today

- More on signals
- Long jumps
- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
Nonlocal Jumps: setjmp/longjmp

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

- `int setjmp(jmp_buf buf)`
  - 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
void longjmp(jmp_buf buf, int i)

- Meaning:
  - return from the `setjmp` remembered by jump buffer `buf` again ...
  - ... this time returning `i` instead of 0
- Called after `setjmp`
- Called once, but never returns

**longjmp Implementation:**

- Restore register context (stack pointer, base pointer, PC value) from jump buffer `buf`
- Set `%eax` (the return value) to `i`
- Jump to the location indicated by the PC stored in jump buf `buf`
setjmp/longjmp Example

```c
#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)
}
```
Limitations of Nonlocal Jumps

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    if (setjmp(buf)) {
        /* long jump to here */
    } else {
        P2();
    }
}

P2()
{
    . . . P2(); . . . P3(); }

P3()
{
    longjmp(buf, 1);
}
```

Diagram:
- Before longjmp:
  - P1
  - P2
  - P2
  - P3

- After longjmp:
  - P1
  - P2
  - P2
  - P3
Limitations of Long Jumps (cont.)

- Works within stack discipline
  - Can only long jump to environment of function that has been called but not yet completed

```c
jmp_buf env;

P1()
{
    P2(); P3();
}

P2()
{
    if (setjmp(buf)) {
        /* long jump to here */
    }
}

P3()
{
    longjmp(buf, 1);
}
```
Putting It All Together: A Program That Restarts Itself When `ctrl-c’d`

```c
#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...
restarting
processing...
restarting
processing...

Ctrl-c

Ctrl-c

Ctrl-c
Today

- More on signals
- Long jumps
- Virtual memory (VM)
  - Overview and motivation
  - VM as tool for caching
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
Virtual Memory (Previous Lectures)

- Programs refer to virtual memory addresses
  - `movl (%ecx), %eax`
  - Conceptually very large array of bytes
  - Each byte has its own address
  - Actually implemented with hierarchy of different memory types
  - System provides address space private to particular “process”

- Allocation: Compiler and run-time system
  - Where different program objects should be stored
  - All allocation within single virtual address space

- But why virtual memory?
- Why not physical memory?
Problem 1: How Does Everything Fit?

64-bit addresses:
16 Exabyte

Physical main memory:
Few Gigabytes

And there are many processes....
Problem 2: Memory Management

Physical main memory

Process 1
Process 2
Process 3
...
Process n

stack
heap
.text
data
...

What goes where?
Problem 3: How To Protect

Problem 4: How To Share?
Solution: Level Of Indirection

- Each process gets its own private memory space
- Solves the previous problems
Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  \[ \{0, 1, 2, 3 \ldots \} \]

- **Virtual address space:** Set of \( N = 2^n \) virtual addresses
  \[ \{0, 1, 2, 3, \ldots, N-1\} \]

- **Physical address space:** Set of \( M = 2^m \) physical addresses
  \[ \{0, 1, 2, 3, \ldots, M-1\} \]

- Clean distinction between data (bytes) and their attributes (addresses)
- Each object can now have multiple addresses
- Every byte in main memory:
  one physical address, one (or more) virtual addresses
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

- Used in all modern desktops, laptops, workstations
- One of the great ideas in computer science
- **MMU checks the cache**
Why Virtual Memory (VM)?

- **Efficient use of limited main memory (RAM)**
  - Use RAM as a cache for the parts of a virtual address space
    - some non-cached parts stored on disk
    - some (unallocated) non-cached parts stored nowhere
  - Keep only active areas of virtual address space in memory
    - transfer data back and forth as needed

- **Simplifies memory management for programmers**
  - Each process gets the same full, private linear address space

- **Isolates address spaces**
  - One process can’t interfere with another’s memory
    - because they operate in different address spaces
  - User process cannot access privileged information
    - different sections of address spaces have different permissions
Today

- More on signals
- Long jumps
- **Virtual memory (VM)**
  - Overview and motivation
  - **VM as tool for caching**
  - VM as tool for memory management
  - VM as tool for memory protection
  - Address translation
VM as a Tool for Caching

- **Virtual memory**: array of $N = 2^n$ contiguous bytes
  - think of the array (allocated part) as being stored on disk
- Physical main memory (DRAM) = cache for allocated virtual memory
- Blocks are called pages; size = $2^p$

Virtual pages (VP's) stored on disk

Physical pages (PP's) cached in DRAM
Memory Hierarchy: Core 2 Duo

L1/L2 cache: 64 B blocks

Throughput: 16 B/cycle
Latency: 3 cycles

Throughput: 8 B/cycle
Latency: 14 cycles

Throughput: 2 B/cycle
Latency: 100 cycles

Throughput: 1 B/30 cycles
Latency: millions

Miss penalty (latency): 30x

Miss penalty (latency): 10,000x

Not drawn to scale
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about $10x$ slower than SRAM
  - Disk is about $10,000x$ slower than DRAM
    - For first byte, faster for next byte

- Consequences
  - Large page (block) size: typically 4-8 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from CPU caches
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through