CSE 320 System Fundamentals II Signals

YOUNGMIN KWON / TONY MIONE

Signals

A signal is a small message that notifies a process that an event of some type has occurred in the system

Low-level hardware exceptions are processed by the kernel's exception handlers

- Signals provide a mechanism for exposing them to user processes
  - SIGFPE (division by zero)
  - SIGILL (illegal instruction)
  - SIGSEGV (illegal memory reference)
  - SIGINT (Ctrl-C)
  - SIGCHLD (stop/termination of child process)

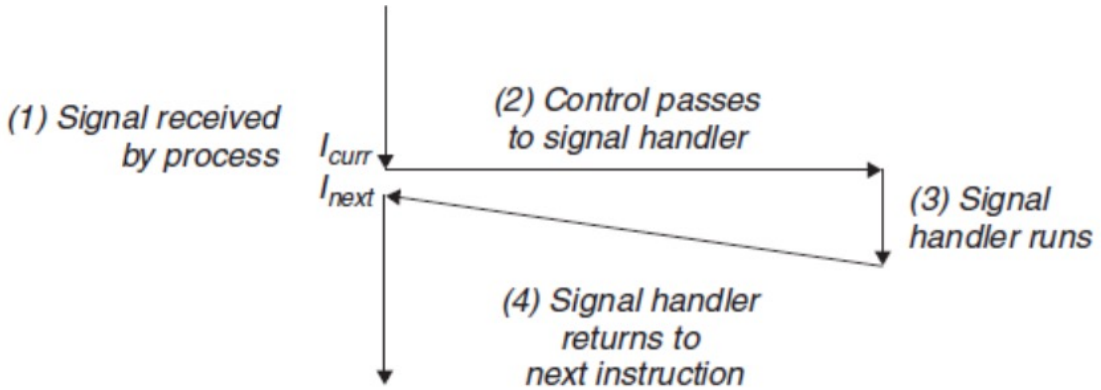
Signals

Sending a signal

- The kernel sends (delivers) a signal to a process by updating some state of the process
- Invoked by a system event or by the `kill` function

Receiving a signal

- A process receives a signal when it is forced by the kernel to react to the delivery of the signal



Sending Signals

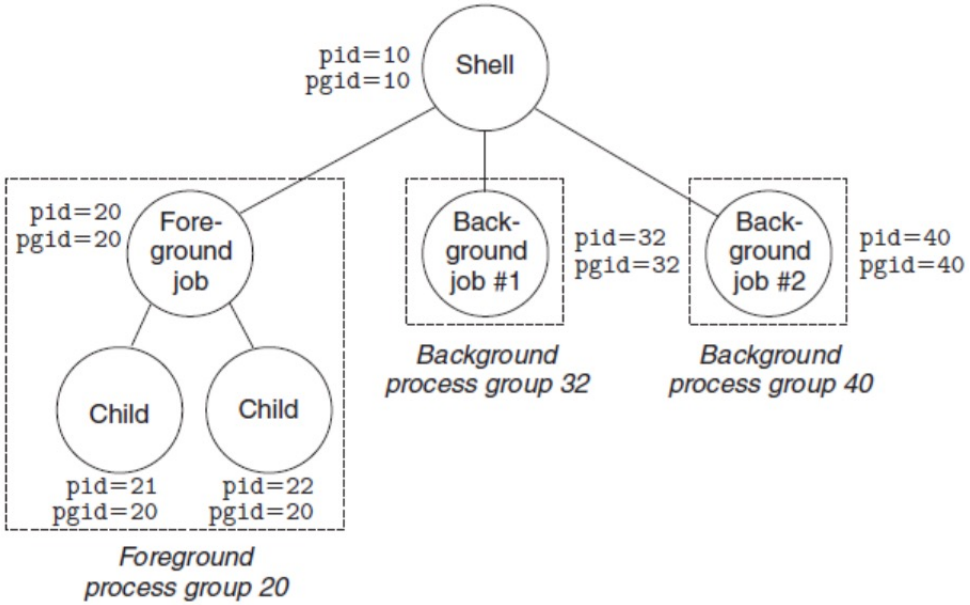
Process Group

- Each process belongs to one process group
- `pid_t getpgrp(void)` returns the process group id
- `int setpgid(pid_t pid, pid_t pgid)`sets the process group id of a process
- Calls require including:
  - `<sys/types.h>`
  - `<unistd.h>`

Sending Signals

/bin/kill sends an arbitrary signal to another process

- > /bin/kill -9 1234 sends 9 (SIGKILL) to process 1234
- > /bin/kill -9 -1234 sends 9 to all processes in process group 1234



Sending Signals

A process can send signals to other processes by calling

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

- If pid > 0: send sig to the process pid
- If pid = 0: send sig to every process in the calling process' process group
- If pid < 0: send sig to every process in the process group |pid|

```
unsigned int alarm(unsigned int secs);
```

- Send SIGALRM to itself after secs second

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

Sending Signals

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
int main() {
    pid_t pid = fork();
    if(pid == 0) { // Child
        printf("before pause\n");
        pause();
        printf("after pause\n");
    }
    else {          // Parent
        sleep(1);
        printf("killing child\n");
        kill(pid, SIGKILL);
    }
    return 0;
}
```

Receiving Signals

When the kernel switches from kernel mode to user mode

- It checks the set of unblocked pending signals (`pending & ~blocked`)
- If this set is not empty the kernel forces the process to receive the signal
- Once an action for the signal is complete the control passes back to the next instruction (I_{next})

Receiving Signals

Default actions

- Terminate the process
- Terminate and dump core
- Stop (suspend) until restarted by a SIGCONT signal
- Ignores the signal

Receiving Signals

signal function can change the action associated with a signal signum

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- If handler is SIG_IGN, the signal is ignored
- If handler is SIG_DFL, the default action for the signal
- Otherwise, the handler is the address of a user-defined function called signal handler

Receiving Signals

```c
#include <stdio.h>
#include <signal.h>
int sigStrlen(char *str) {
    int i;
    for(i = 0; str[i]; i++) ;
    return i;
}
void sigPuts(char *str) { write(1, str, sigStrlen(str)); }
void handler(int sig) { sigPuts("Caught SIGINT!\n"); }
int main() {
    if(signal(SIGINT, handler) == SIG_ERR) {
        printf("error in %s at %d\n", __FUNCTION__, __LINE__);
        return 0;
    }
    printf("before pause\n");
    pause();
    printf("after pause\n");
    return 0;
}
```

Blocking and Unblocking Signals

Applications can explicitly block and unblock selected signals using sigprocmask

```
int sigprocmask(int how,
                const sigset_t *set,
                sigset_t *oldset);
```

- how:
  SIG_BLOCK add signals in set to be blocked,
  SIG_UNBLOCK remove signals in set from blocked,
  SIG_SETMASK make blocked equal to set

- oldset: if not NULL, the previous set of blocked signals is returned

Blocking and Unblocking Signals

Helper functions

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

Blocking Signals : Example

```c
#include <stdio.h>
#include <signal.h>
void handler(int sig) { sigPuts("Caught SIGINT!\n"); }

int main() {
    signal(SIGINT, handler); //register handler

    sigset_t mask, oldmask;  // prepare mask set
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT);

    //Try Ctrl-C with and without this line
    sigprocmask(SIG_BLOCK, &mask, &oldmask);

    printf("Before sleep\n");
    sleep(3);
    printf("After sleep\n");

    sigprocmask(SIG_SETMASK, &oldmask, NULL);
    return 0;
}
```

Safe Signal Handling

G0: Keep handlers simple

- Handlers set global flags and let main program check the flags periodically

G1: Call only async-signal-safe functions in your handler

- printf, sprintf, malloc, exit are **not safe**
- write, _exit are safe

G2: Save and restore errno

- On entering a handler save errno to a local variable and restore it before returning from the handler

Safe Signal Handling

G3: Protect access to shared global data structures

- Temporarily block signals while accessing the data structures

G4: Declare global variables with volatile

```
volatile int g;
```

- Compiler optimization may cache variables in registers and not reading them from the memory
- Updates from handlers may be ignored

G5: Declare flags with sig_atomic_t

```
sig_atomic_t flag;
```

- Reads and writes are guaranteed to be atomic (uninterruptable)

Correct Signal Handling
: Signals are not queued

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
void prnQuit(char *msg) {
    sigPuts(msg);
    _exit(0);
}
void handler(int sig) {
    int olderrno = errno;

    if(waitpid(-1, NULL, 0) < 0)
        sigPuts("waitpid error\n");
    sigPuts("Reaped child\n");

    sleep(1);
    errno = olderrno;
}
```

```c
int main() {
    int i;
    char buf[100];

    if(signal(SIGCHLD, handler)
                            == SIG_ERR)
        prnQuit("signal error\n");

    for(i = 0; i < 3; i++)
        if(fork() == 0)
            prnQuit("child\n");

    if(read(0, buf, sizeof(buf)) < 0)
        prnQuit("read error\n");

    return 0;
}
```

Correct Signal Handling
: Signals are not queued

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

    while(waitpid(-1, NULL, 0) > 0)
        sigPuts("Reaped child\n");

    sleep(1);
    errno = olderrno;
}
```

Synchronizing Flows

```
static int num_jobs = 0;
void add_to_queue(pid_t pid) {
    num_jobs++; // only count the # of entries in queue
}
void delete_from_queue(pid_t pid) {
    if(num_jobs > 0)
        num_jobs--; // only count the # of entries in queue
    else
        prnQuit("nonexistent job\n");
}
void handler(int sig) {
    int olderrno = errno;
    pid_t pid;

    // reap the child and delete it from the job queue
    while((pid = waitpid(-1, NULL, 0)) > 0)
        delete_from_queue(pid);

    errno = olderrno;
}
```

Synchronizing Flows

```
int main() {
    char *argv[] = { "/bin/date", NULL };
    pid_t pid;
    sigset_t mask, prev;
    sigfillset(&mask);
    signal(SIGCHLD, handler);
    while(1) {
        if((pid = fork()) == 0)
            execve(argv[0], argv, NULL);

        //block for the exclusive access to the job queue
        sigprocmask(SIG_BLOCK, &mask, &prev);

        //BUG: if the child exits before the parent adds pid
        //to the queue, delete_from_queue cannot find/remove
        //the pid and it will remain forever in the queue
        add_to_queue(pid);
        sigprocmask(SIG_SETMASK, &prev, NULL);
    }
    return 0;
}
```

Synchronizing Flows

```
int main() {
    char *argv[] = { "/bin/date", NULL };
    pid_t pid;
    sigset_t mask, prev;
    sigfillset(&mask);
    signal(SIGCHLD, handler);
    while(1) {
        //block before fork for exclusive access to the job queue
        sigprocmask(SIG_BLOCK, &mask, &prev);
        if((pid = fork()) == 0) {
            //unblock signals for child
            sigprocmask(SIG_SETMASK, &prev, NULL);
            execve(argv[0], argv, NULL);
        }

        //unblock for parent after adding pid to queue
        add_to_queue(pid);
        sigprocmask(SIG_SETMASK, &prev, NULL);
    }
    return 0;
}
```

Explicit Waiting for Signals

```
Static volatile sig_atomic_t pid;

void sigchld_handler(int sig) {
    int olderrno = errno;

    // main will wait until pid is set to a nonzero value
    pid = waitpid(-1, NULL, 0);

    errno = olderrno;
}

void sigint_handler(int sig) {
    _exit(0);
}
```

```c
int main() {
    char *argv[] = {"/bin/date", NULL};
    sigset_t mask, prev;
    sigfillset(&mask);
    signal(SIGINT,  sigint_handler);
    signal(SIGCHLD, sigchld_handler);
    while(1) {
        sigprocmask(SIG_BLOCK, &mask, &prev);
        if((pid = fork()) == 0) {
            sigprocmask(SIG_SETMASK, &prev, NULL);
            execve(argv[0], argv, NULL);
        }

        pid = 0;
        sigprocmask(SIG_SETMASK, &prev, NULL);
        while(!pid) ;           //correct but wasteful spinlock
        //while(!pid)           //BUG: SIGCHLD can be received after
        //    pause();          //     !pid check but before pause()
        //while(!pid)           //correct but too slow
        //    sleep(1);
        printf("child pid: %d\n", pid);
    }
    return 0;
}
```

```
int main() {
    //This part is the same as the previous main

    while(1) {
        sigprocmask(SIG_BLOCK, &mask, &prev);
        if((pid = fork()) == 0) {
            sigprocmask(SIG_SETMASK, &prev, NULL);
            execve(argv[0], argv, NULL);
        }
        pid = 0;
        while(!pid)
            sigsuspend(&prev);
            //equivalent to an atomic version of
            //    sigprocmask(SIG_SETMASK, &prev, &tmp);
            //    pause();
            //    sigprocmask(SIG_SETMASK, &tmp, NULL);
        sigprocmask(SIG_SETMASK, &prev, NULL);
        printf("child pid: %d\n", pid);
    }
    return 0;
}
```

Question?