Signals

Signals inform processes the occurence of asynchronous events. Processes may send each signals with the `kill` system call or the kernel may send signals internally.

Types

- Signals having to do with **termination of process** with the `signal` system call (with the death of the child parameter) or when a process *exits*.
- **Process induced exceptions**; when a process accesses an address outside its virtual space, when it attempts to write memory that is read-only (such as a program text), etc.
- **Unrecoverable conditions during a system call**, such as running out of system resources.
- **Unexpected error condition during a system call**, such as making a nonexistent system call (the process passed a system call number that is not a legal system call), writing a pipe that has no reader processes, etc
- **Signals originating from a process in user mode** (`alarm` or `kill`).
- **Related to terminal interaction** (*break* or `ctrl+c`)
- **For tracking execution of a process.**

Treatment of Signals

- How the kernel sends a signal to a process - sets a signal bit in the process table entry, if the process is asleep at the interruptible priority, the kernel awakens it.
- How the process handles a signal
- How the process control its reaction to the signals

Algorithm for recognizing signals

```
algorithm issig  /*test for receipt of signals*/
input: none
output: true, if process received signals that it does not ignore
                false, otherwise
{
        while(received signal field in process table entry not 0){
                find a signal number sent to the process;

                if(signal is death of child){
                        if(ignoring death of child signals)
                                free process table entries of zombie children;
                        else if(catching death of child signals)
```

```
                                return true;
                }

                else if(not ignoring signal)
                        return true;
                turn off signal bit in received signal field in the process table;
        }

        return false;
}
```

Handling signals

Three cases:

- The **process exits** on the receipt of a signal.
- The **process ignores** the signal
- The process executes a particular (user) function on the receipt of a signal.

Default Action: calls `exit`.
Special Action: can specify special action with the `signal` call.

The syntax of the signal function:

```
oldfunction = signal(signum, function);
```

`signum` is the signal number
`function` is the address of the user defined function. (can pass 0 or 1 instead of an address, 0 = ignore future occurences of the signal, 0 = default action).

The u area contains an array of signal-handler fields, one for each signal defined in the system.

Algorithm for handling signals

```
algorithm: psig /*handle signals after recognizing their existence*/
input: none
output: none
{
        get signal number set in the process table entry;
        clear signal number in the process table entry;
        if(user had called the signal sys call to ignore this signal)
                return; /*done*/
```

```
        if(user specified function to handle signal){
                get user virtual address of signal catcher stored in the u area;
                /*the next statement has undesirable side effects*/
                clear u area entry that stored address of signal catcher;
                modify user level context:
                                artificially create user stack frame to
mimic call to
                                signal catcher function;
                modify system level context:
                                write address of signal catcher into program
counter
                                field of user saved register context;
                return;
        }

        if(signal is type that system should dump core image of process){
                create file named "core" in the current directory;
                write contents of user level context to file "core";
        }

        invoke exit algorithm immediately;
}
```

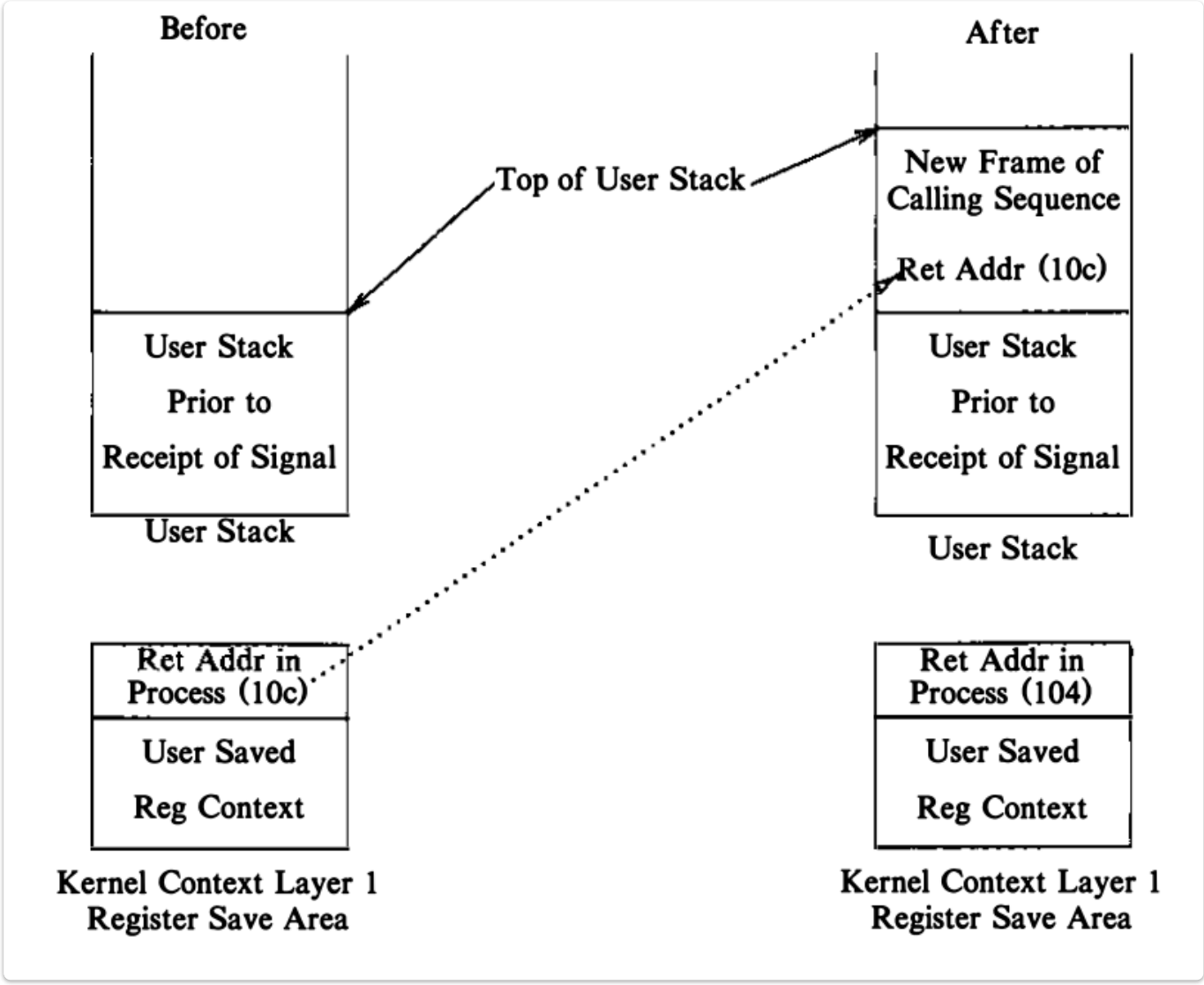
The dump is a convenience to programmers, allowing them to ascertain its causes and, thereby, to debug their programs. The kernel dumps core for signals that imply something is wrong with a process, such as when a process executes an illegal instruction or when it accesses an address outside its virtual address space. But the kernel does not dump core for signals that do not imply a program error.

Anomaly: First and most important, when a process handles a signal but before it returns to user mode, the kernel clears the field in the u area that contains the address of the user signal handling function. If the process wants to handle the signal again, it must call the signal system call again. (usually registering a signal catcher once was enough, it could handle all subsequent interrupt calls). A **race condition** results because a second instance of the signal may arrive before the process has a chance to invoke the system call. Since the process is executing in user mode, the kernel could do a context switch, increasing the chance that the process will receive the signal before resetting the signal catcher.

The kernel does the following steps:

- The kernel accesses the user saved register context, finding the program counter and the stack pointer that it had saved for return to the user process.
- It clears the signal handler field in the u area, setting it to default value.

- The kernel creates a new stack frame on the user stack, writing in the values of program counter and stack pointer it had retrieved from the user saved register context and allocating new space, if neccessary. The user stack looks as if the process had called a user-level function (the signal catcher) at the point where it had made the system call or where the kernel had interrupted.
- The kernel changes the user saved register context: It resets the value for the program counter to the address of the signal catcher function and sets the value for the stack pointer to account for the growth of the user stack.



Predefined Signals

- SIGALRM: Alarm timer timeout; generated by alarm() API
- SIGILL: Execution of an illegal machine instruction
- SIGINT: Process interruption, can be generated by `delete` or `ctrl-c` keys
- SIGSEGV: Segmentation fault
- SIGTERM: process termination (`kill` command)
- SIGCHLD: Sent to a parent process when its child process has been terminated.

Example

```
#include<signal.h>
main(){
        extern catcher();
        signal(SIGINT, catcher); // register signal catcher
        kill(0, SIGINT); // induce a SIGINT signal
}

catcher(){
}
```

Example 2

```
#include <signal.h>
sigcatcher(){
        printf("PID %d caught one\n", getpid()); /* print proc id */
        signal(SIGINT, sigcatcher);
}

main(){
        int ppid;

        signal(SIGINT, sigcatcher);

        if(fork() == 0){
                /* give enough time for both procs to set up */
                sleep(5); // delay 5sec
                ppid = getppid(); // get parent pid
                for(;;)
                        if(kill(ppid, SIGINT) == -1)
                                exit();
        }

        /* lower priority, greater chance for exhibiting race */
        nice(10);
        for(;;);

}
```

In the above example, it is possible that the following sequence may occur:

- The child sends an interrupt signal to the parent process

- The parent process catches a signal and call the signal catcher, but the kernel preempts the process and switches context before it executes `signal()` again (it will not register a signal catcher)
- The child process executes again and sends another interrupt signal to the parent process.
- The parent process receives the second interrupt signal, but it has not made arrangements to catch the signal. When it resumes execution, it *exits* (it will perform the default action: call `exit()`).

Process Groups

Identifying processes by group. For example: Processes with a common ancestor that is a login shell are generally related, and therefore all such processes receive signals.

The kernel uses the process group ID which it saves in the process table; processes in the same group have identical group IDs.

```
grp = setpgrp();
```

A child retains the process group number of its parent during `fork`.

Sending signals from processes

Processes use the `kill` system call to send signal. Syntax:

```
kill(pid, signum);
```

`pid` identifies the set of processes to receive the signal
`signum` is the signal number.

`pid > 0:` The kernel sends the signal to the process with process ID `pid`.
`pid == 0:` the kernel sends the signal to all the processes in the sender's process group.
`pid == -1:` the kernel sends the signal to all the processes with `real PID == effective PID of the sender`. If the effective PID is of the superuser, it sends to all the processes except process 0 and 1.
`pid < 0 && pid ≠ -1:` the kernel sends the signal to all processes in the process group equal to the absolute value of `pid`.

In all cases, if the sending process does not have effective user ID of superuser, or its real or effective user ID do not match the real or effective user ID of the receiving process, `kill` fails.