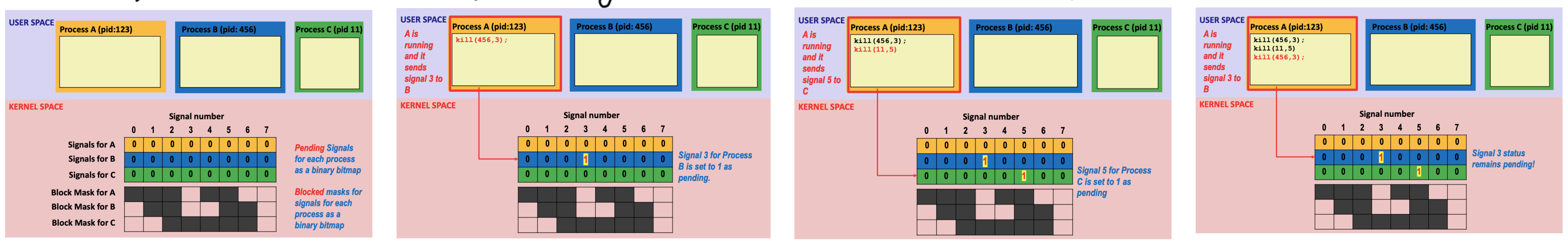
signals

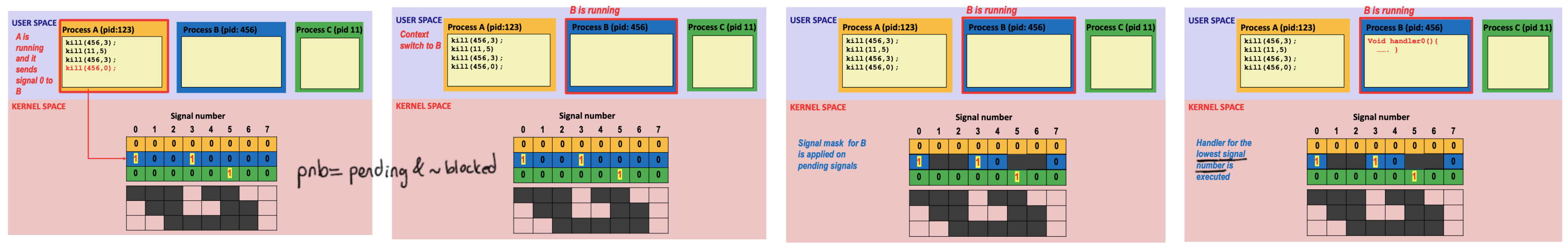
signal = small message that notifies a process about events, signal id (1-30) + arrival info

↳ software interrupts = hardware $\xrightarrow{\text{interrupt}}$ OS $\xrightarrow{\text{signal}}$ process

↳ most of them causes termination, but they can be blocked with handlers (except SIGKILL, SIGSTOP)

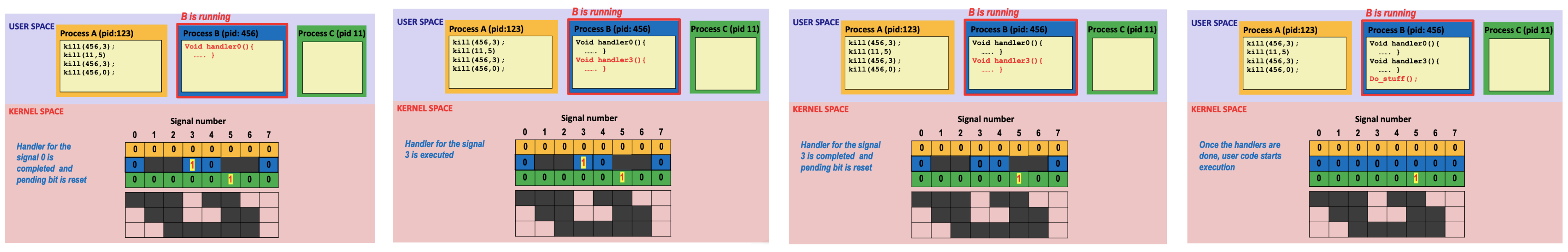


USER SPACE
Process A (pid:123)
Process B (pid: 456)
Process C (pid 11)
KERNEL SPACE
Signal number
Signals for A
Signals for B
Signals for C
Block Mask for A
Block Mask for B
Block Mask for C
Pending Signals for each process as a binary bitmap
Blocked masks for signals for each process as a binary bitmap
A is running and it sends signal 3 to B
kill(456,3);
Signal 3 for Process B is set to 1 as pending.
A is running and it sends signal 5 to C
kill(11,5)
Signal 5 for Process C is set to 1 as pending
Signal 3 status remains pending!

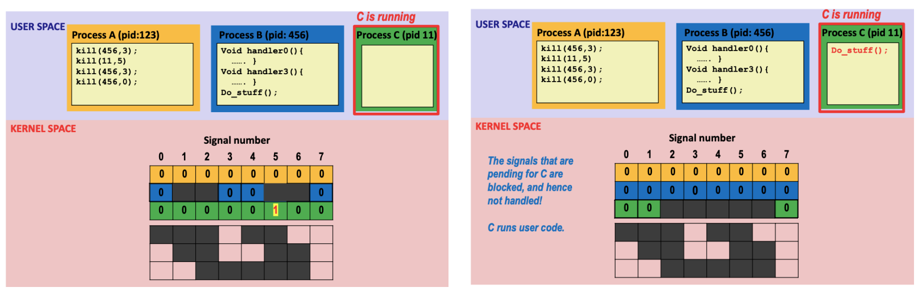


A is running and it sends signal 0 to B
kill(456,0);
Context switch to B
B is running
Signal mask for B is applied on pending signals
Handler for the lowest signal number is executed
Void handler0(){

pnb = pending & ~blocked



B is running
Handler for the signal 0 is completed and pending bit is reset
Handler for the signal 3 is executed
Void handler3(){
Handler for the signal 3 is completed and pending bit is reset
Once the handlers are done, user code starts execution
Do_stuff();



C is running
The signals that are pending for C are blocked, and hence not handled!
C runs user code.
Do_stuff();

ID	Name	Default Action	Corresponding Event
1	SIGHUP	Terminate	Terminal line close
2	SIGINT	Terminate	User typed ctrl-c
3	SIGQUIT	Terminate	Ctrl+ \
4	SIGILL	Terminate	Illegal instruction on CPU
8	SIGFPE	Terminate	Floating point exception
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate	Segmentation violation
13	SIGPIPE	Terminate	Write on a closed pipe
14	SIGALRM	Terminate	User timer
15	SIGTERM	Terminate	Terminate process (can be overwritten)
17	SIGCHLD	Ignore	Child stopped or terminated
19	SIGSTOP	Suspend	Suspend process execution
18	SIGCONT	Continue	Continue suspended process
10 12	SIGUSR1 SIGUSR2	Ignore	User defined

hardware interrupt = 8, 9, 11

OS event = 1, 13, 14, 17, user input

process request = kill() - system call

↳ kernel delivers signal to a destination process by updating some context of the destination process

pending = signal is sent to process, waiting to be delivered

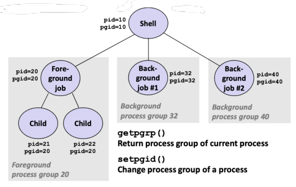
↳ there can be at most one pending signal of any particular type, not queued, first one is pending, rest discarded

↳ a pending signal is received at most once

↳ blocked signals can be delivered, but will not be received until the signal is unblocked

process groups = every process belongs to exactly one process group

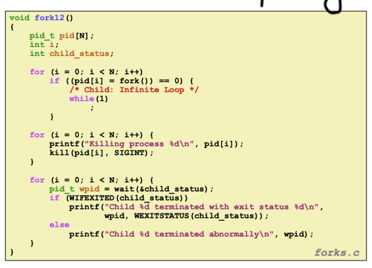
• if signal is sent to group, every member receives



`getpgrp()`
Return process group of current process

`setpgid()`
Change process group of a process

- `/bin/kill -9 24818`
  Send `SIGKILL` to process 24818
- `/bin/kill -9 -24817`
  Send `SIGKILL` to every process in process group 24817

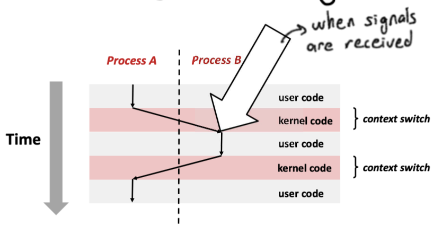


```
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1)
                ;
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    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);
    }
}
                                                    forks.c
```

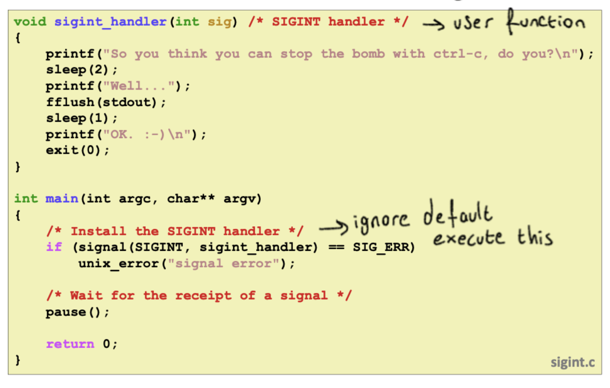


↳ reacting to signals = ignore (do nothing), terminate, catch by a signal handler

↳ each signal has predefined default action = terminate process, ignore sig, stop process until $\overset{\text{SIGCONT}}{\uparrow}$ restarted

handlers = handler_t * signal(int signal_number, handler_t * handler)

↳ ignore, revert to default, catch & handle



```
void sigint_handler(int sig) /* SIGINT handler */   → user function
{
    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(int argc, char** argv)
{
    /* Install the SIGINT handler */   → ignore default
    if (signal(SIGINT, sigint_handler) == SIG_ERR)   execute this
        unix_error("signal error");

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

    return 0;
}
                                                    sigint.c
```

• after executing user defined handler in kernel mode, returns to the next instruction

• kernel uses the same stack for handler as process's stack

implicit blocking signals = kernel blocks any pending signals of type currently being handled (ctrl+c → $\overset{\text{ignored}}{\uparrow}$ ctrl+c)

explicit un/blocking signals = sigprocmask function → can change bits in blocking bitmap