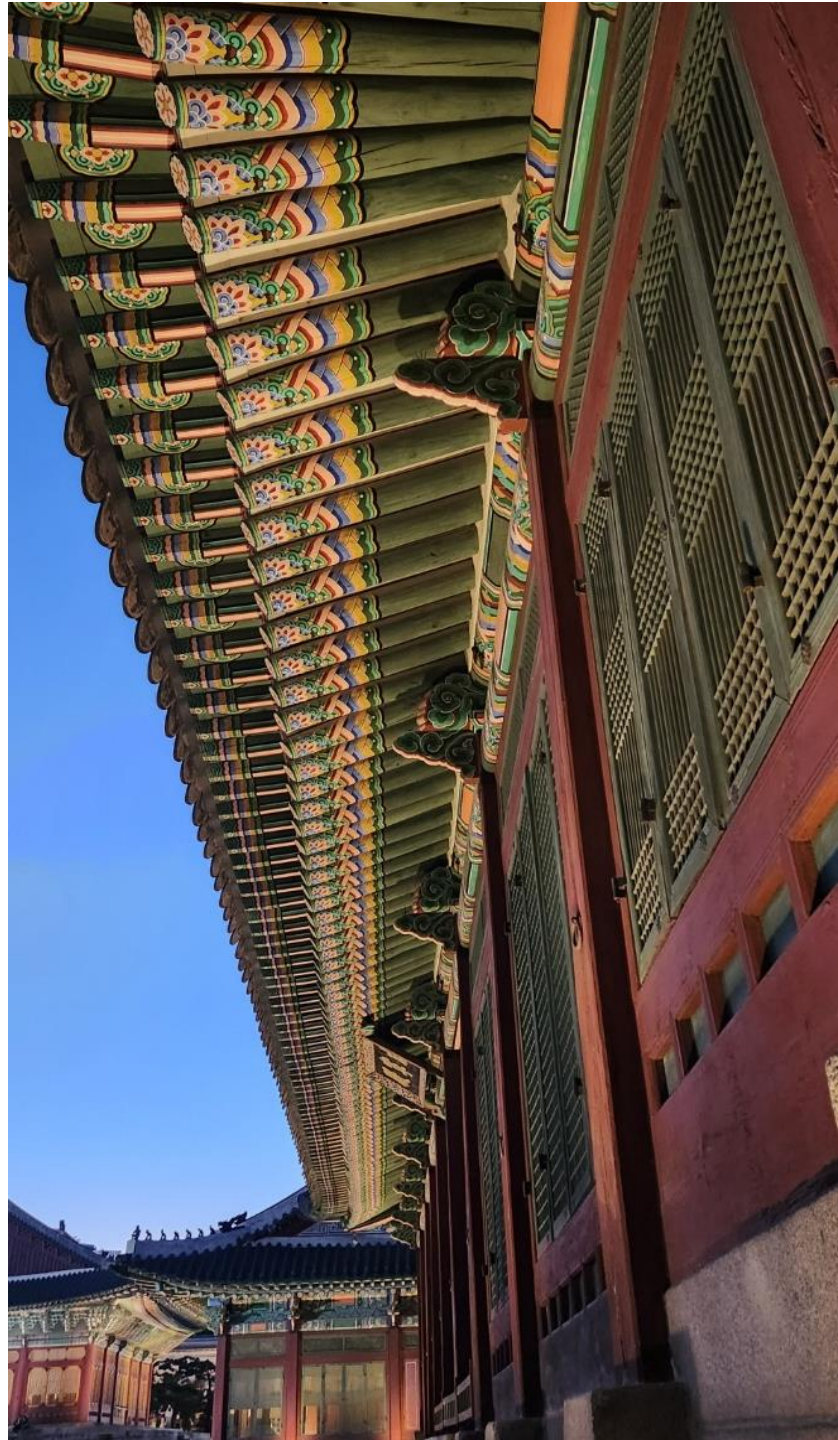
Signal & Time

HGU



Contents

- Shell Program
- Signal
- Time

Shell Programs

- A shell is an application program that runs programs on behalf of the user.
  - `sh`: Original Unix Bourne Shell
  - `csh`: BSD Unix C Shell
  - `tcsh`: Enhanced C Shell
  - `bash`: Bourne-Again Shell

Simple Shell: main() & eval()

shellex.c

```
#define MAXLINE  8192  /* Max text line length */
#define MAXARGS  128

/* Function prototypes */
void eval(char *cmdline);
int parseline(char *buf, char **argv);
int builtin_command(char **argv);

extern char **environ; /* Defined by libc */

int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps

```
/* eval */
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* Argument list execve() */
    char buf[MAXLINE];   /* Holds modified command line */
    int bg;              /* Should the job run in bg or fg? */
    pid_t pid;           /* Process id */

    strcpy(buf, cmdline);
    bg = parseline(buf, argv);
    if (argv[0] == NULL)
        return;

    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) {
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {
            int status;
            if (waitpid(pid, &status, 0) < 0)
                perror("waitfg: waitpid error");
        }
        else
            printf("%d %s", pid, cmdline);
    }
    return;
}
```

Simple Shell: `builtin_command()` & `parseline()`

shellex.c

```
/* builtin - */
int builtin_command(char **argv)
{
    if (!strcmp(argv[0], "quit"))
        exit(0);
    if (!strcmp(argv[0], "&"))
        return 1;
    return 0;
}
```

Expected results?

```
/* parseline - ? */
int parseline(char *buf, char **argv)
{
    char *delim;         /* Points to first space delimiter */
    int argc;            /* Number of args */
    int bg;              /* Background job? */

    buf[strlen(buf)-1] = ' ';
    while (*buf && (*buf == ' '))
    buf++;

    argc = 0;
    while ((delim = strchr(buf, ' '))) {
        argv[argc++] = buf;
        *delim = '\0';
        buf = delim + 1;
        while (*buf && (*buf == ' '))
            buf++;
    }
    argv[argc] = NULL;

    if (argc == 0)
        return 1;

    if ((bg = (*argv[argc-1] == '&')) != 0)
        argv[--argc] = NULL;

    return bg;
}
```

Contents

- Shell Program
- Signal
- Time

Problem with Simple Shell Example

- 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.
- Solution: Reaping background jobs requires a mechanism called a **signal**.

Signal

- 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
  - The only information in a signal is its ID and the fact that it arrived.
  - Each signal has a current disposition (action associated with a signal), which determines how the process behaves when it is delivered the signal.

Signal Types

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctrl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Igonre	Child stopped or terminated

```
yunmin@peace:~/ch10$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
```

https://man7.org/linux/man-pages/man7/signal.7.html

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.

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.
- Signal disposition: Each signal has a current disposition (i.e., default action), which determines how the process behaves when it is delivered the signal
  - Terminate the process
  - Terminate the process and dump core
  - Ignore the signal
  - Stop the process
  - Continue the process if it is currently stopped
- Also a process can change the disposition of a signal
  - Process executes a user-level function called a signal handler when the specified signal occurs.

https://man7.org/linux/man-pages/man7/signal.7.html

Signal Pending and Blocked

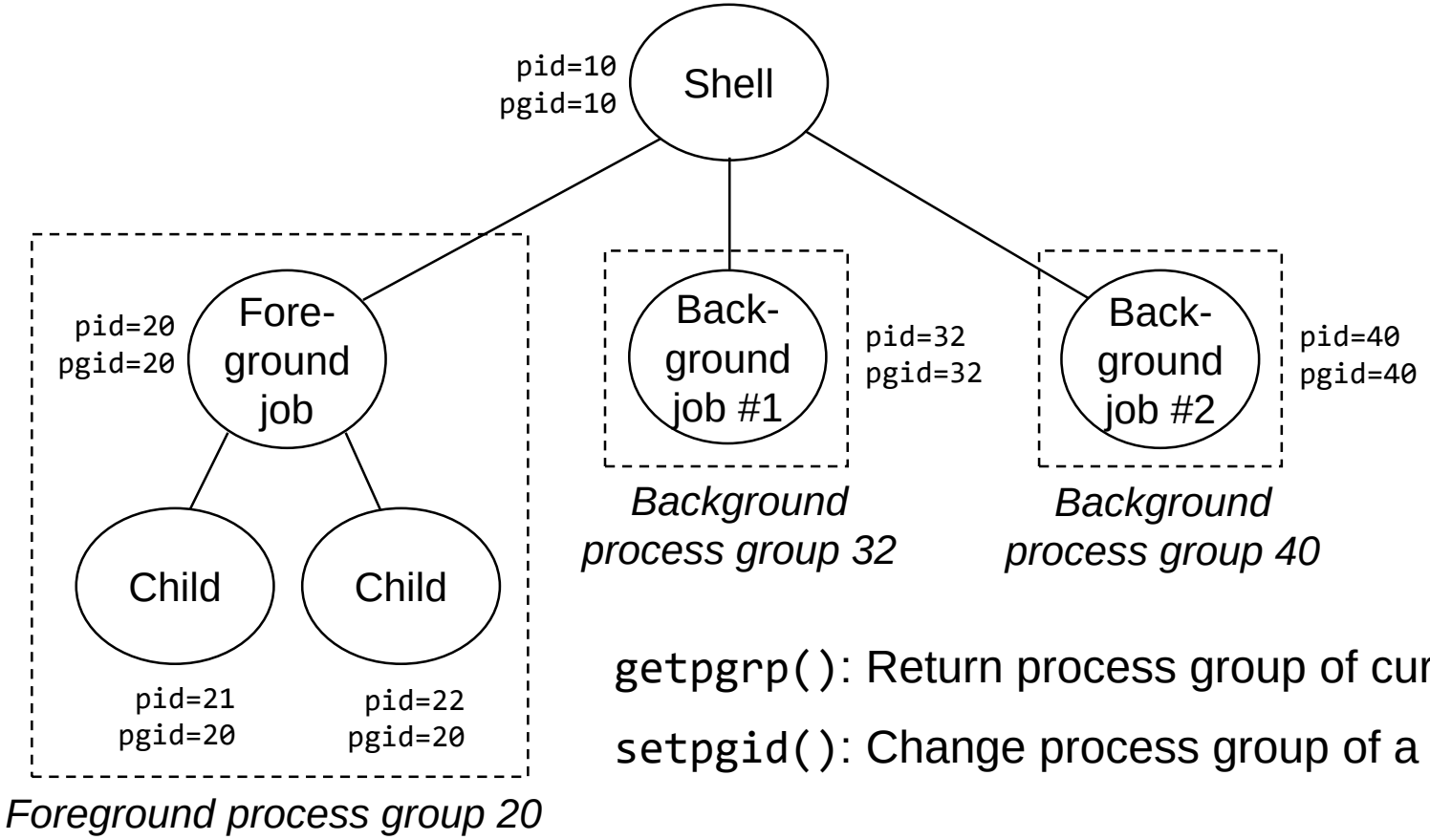
- A signal is pending if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
-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

Signal Pending and Blocked

- 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.

Process Groups

- Every process belongs to exactly one process group



`getpgrp()`: Return process group of current process

`setpgid()`: Change process group of a process

Kill Program

- kill program sends arbitrary signal to a process or process group

killproc1.c

```
int main()
{
    if (fork() == 0) {
        printf("Child1: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
        if (fork() == 0)
            printf("Child2: pid=%d pgrp=%d\n",
                    getpid(), getpgrp());
        while(1);
    }
    return 0;
}
```

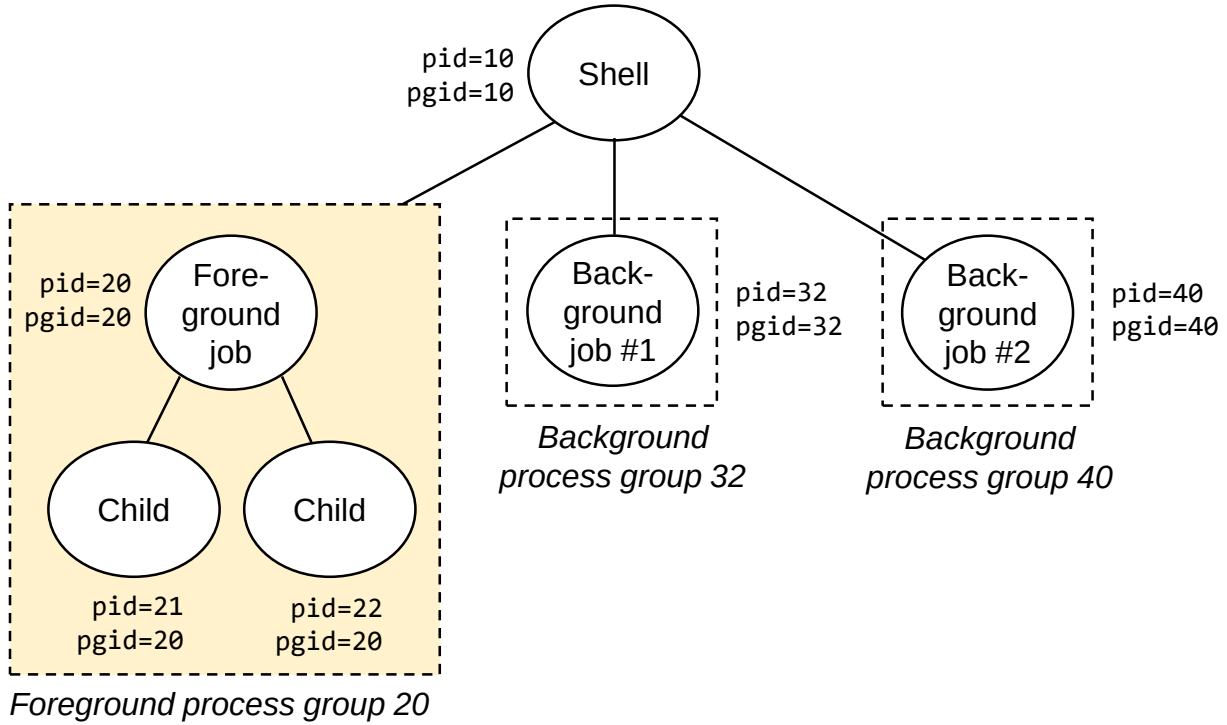
- $ kill -9 15251
  - Send SIGKILL to process 15251
- $ kill -9 -15545
  - Send SIGKILL to every process in process group 15545

```
yunmin@peace:~/ch10$ ./killproc1
Child1: pid=7238 pgrp=7237
Child2: pid=7239 pgrp=7237
yunmin@peace:~/ch10$ ps
  PID TTY          TIME CMD
 5530 pts/0    00:00:00 bash
 7238 pts/0    00:00:05 killproc1
 7239 pts/0    00:00:05 killproc1
 7242 pts/0    00:00:00 ps
yunmin@peace:~/ch10$ kill -9 7238
yunmin@peace:~/ch10$ ps
  PID TTY          TIME CMD
 5530 pts/0    00:00:00 bash
 7239 pts/0    00:00:14 killproc1
 7280 pts/0    00:00:00 ps
yunmin@peace:~/ch10$ kill -9 7239
yunmin@peace:~/ch10$ ps
  PID TTY          TIME CMD
 5530 pts/0    00:00:00 bash
 7321 pts/0    00:00:00 ps
```

```
yunmin@peace:~/ch10$ ./killproc1
Child1: pid=7380 pgrp=7379
Child2: pid=7381 pgrp=7379
yunmin@peace:~/ch10$ ps
  PID TTY          TIME CMD
 5530 pts/0    00:00:00 bash
 7380 pts/0    00:00:02 killproc1
 7381 pts/0    00:00:02 killproc1
 7396 pts/0    00:00:00 ps
yunmin@peace:~/ch10$ kill -9 -7379
yunmin@peace:~/ch10$ ps
  PID TTY          TIME CMD
 5530 pts/0    00:00:00 bash
 7446 pts/0    00:00:00 ps
```

Signals from Keyboard

- Typing `ctrl-c` (`ctrl-z`) sends a `SIGTERM` (`SIGTSTP`) to every job in the foreground process group.
  - `SIGTERM`: default action is to terminate each process
  - `SIGTSTP`: default action is to stop (suspend) each process



Signals from Keyboard: Example

- Example of `ctrl-c` and `ctrl-z`

killproc2.c

```
int main()
{
    if (fork() == 0) {
        printf("Child: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
    } else {
        printf("Parent: pid=%d pgrp=%d\n",
                getpid(), getpgrp());
    }
    while(1);
    return 0;
}
```

```
yunmin@peace:~/ch10$ ./killproc2
Parent: pid=7604 pgrp=7604
Child: pid=7605 pgrp=7604
^Z      ← Typed ctrl-z
[1]+  Stopped                 ./killproc2
yunmin@peace:~/ch10$ ps a
  PID TTY      STAT   TIME COMMAND
 2239 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
 3632 tty2     S<sl+   0:00 /usr/lib/xorg/Xorg :0 -br -once -con
 5530 pts/0    Ss     0:00 /bin/bash --init-file /home/yunmin/.v
 7604 pts/0    T      0:02 ./killproc2
 7605 pts/0    T      0:02 ./killproc2
 7650 pts/0    R+     0:00 ps a
yunmin@peace:~/ch10$ fg
./killproc2
^C      ← Typed ctrl-c
yunmin@peace:~/ch10$ ps a
  PID TTY      STAT   TIME COMMAND
 2239 tty1     Ss+    0:00 /sbin/agetty --noclear tty1 linux
 3632 tty2     S<sl+   0:00 /usr/lib/xorg/Xorg :0 -br -once -con
 5530 pts/0    Ss     0:00 /bin/bash --init-file /home/yunmin/.v
 7721 pts/0    R+     0:00 ps a
```

T: stopped by job control signal

`fg`: By default, this brings the last background job to the foreground.

`fg %job_number`: Brings a specific job (by job number) to the foreground.

https://man7.org/linux/man-pages/man1/ps.1.html

kill()

- `kill()`: Send signal to a process

```
#include <signal.h>
int kill(pid_t pid, int sig);
```

  - The `kill()` system call can be used to send any signal to any process group or process.
  - *`pid`*>0: *`sig`* is sent to the process with specified by *`pid`*
  - *`pid`*=0: *`sig`* is sent to every process in the process group of the calling process
  - *`pid`*=-1: *`sig`* is sent to every process for which the calling process has permission to send signals, except for process 1 (init)
  - *`pid`*<-1: *`sig`* is sent to every process in the process group whose ID is *`-pid`*.
  - Return value
    - Success: zero is returned
    - Error: -1 is returned

https://man7.org/linux/man-pages/man2/kill.2.html

kill(): Example

killproc3.c

```
void main()
{
    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]);
        // TODO
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = // TODO
        if ( // TODO                )
            printf("Child %d terminated with exit status %d\n",
                    wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
yunmin@peace:~/ch10$ ./killproc3
Killing process 8114
Killing process 8115
Killing process 8116
Killing process 8117
Killing process 8118
Child 8114 terminated abnormally
Child 8115 terminated abnormally
Child 8116 terminated abnormally
Child 8117 terminated abnormally
Child 8118 terminated abnormally
```

Receiving Signals

- Suppose kernel is returning from 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.
```

signal()

- `signal()`: Define the action associated with a signal

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

  - Sets the disposition of the signal *`signum`* to *`handler`*
  - *`signum`*: the name of signal
  - *`handler`*: could be one among three possible reaction to the signal *`signum`*
    - `SIG_IGN`: ignore the signals
    - `SIG_DFL`: use default action
    - `Address of signal handler`: use a programmer-defined function
      → Installing the handler

- Return value
  - Success: the previous value of the signal handler
  - Error: SIG_ERR

https://man7.org/linux/man-pages/man7/signal.7.html

signal(): Example

signal1.c

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

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

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

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    /* Parent reaps terminated children */
    ...
```

```
yunmin@peace:~/ch10$ ./signal1
Killing process 8319
Killing process 8320
Killing process 8321
Killing process 8322
Killing process 8323
Process 8319 received signal 2
Process 8320 received signal 2
Process 8321 received signal 2
Process 8323 received signal 2
Process 8322 received signal 2
Child 8319 terminated with exit status 0
Child 8320 terminated with exit status 0
Child 8321 terminated with exit status 0
Child 8322 terminated with exit status 0
Child 8323 terminated with exit status 0
```

Signal Handler Funkiness

signal2.c

```
int ccount = 0;

// SIGCHLD handler that reaps one terminated child
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);
}

// Signal funkiness: Pending signals are not queued
void main()
{
    pid_t pid[N];
    int i;
    ccount = N;

    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            exit(0);   /* Child: Exit */
        }
    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

Living With Nonqueuing Signals

signal3.c

```
int ccount = 0;

// SIGCHLD handler that reaps all terminated children
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = wait(&child_status)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}

// Using a handler that reaps multiple children
void main()
{
    pid_t pid[N];
    int i;
    ccount = N;

    signal(SIGCHLD, child_handler2);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1);
            exit(0); /* Child exits */
        }
    while (ccount > 0);  /* Parent spins */
}
```

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

Example for SIGINT

- A program that reacts to externally generated events (ctrl-c)

sigint.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.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);
}

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

Typed ctrl-c

```
yumin@peace:~/ch10$ ./sigint
^CYou think hitting ctrl-c will stop the bomb?
Well...OK
```

Example for SIGALRM

- A program that reacts to internally generated events

sigalrm.c

```
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);
    }
}

void main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */

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

```
yunmin@peace:~/ch10$ ./sigalrm
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
```

sigaction()

- `sigaction()`: Examine and change a signal action

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

  - Used to change the action taken by a process on receipt of a specific signal.
  - *signum*: the name of signal
  - *act*: the new action
  - *oldact*: the previous actions
  - Return value
    - Success: 0
    - Error: -1

  - struct sigaction

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int      sa_flags;
    void     (*sa_restorer)(void);
};
```

    - sa_handler: the address of a function to be called when the signal occurs
    - sa_mask: specifies a set of signals that will be blocked when the signal handler is called
    - sa_flags: options. Basically 0

https://man7.org/linux/man-pages/man7/signal.7.html

Example: sigaction()

sigaction.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// Signal handler for timeout
void timeout(int sig)
{
    if (sig == SIGALRM)
        puts("Time out!");

    // generate SIGALRM after 2 seconds
    alarm(2);
}
```

```c
int main(int argc, char *argv[])
{
    int i;
    struct sigaction act;
    act.sa_handler = timeout;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;

    // specify signal type and signal handler
    sigaction(SIGALRM, &act, 0);

    alarm(2);

    for (i = 0; i < 3; i++)
    {
        puts("wait...");
        sleep(10);
    }
    return 0;
}
```

```
yunmin@peace:~/ch10$ ./sigaction
wait...
Time out!
wait...
Time out!
wait...
Time out!
```

Contents

- Shell Program
- Signal
- Time

POSIX Clocks

- POSIX clocks are standardized ways to represent and retrieve various time sources in Unix-like operating systems
- They enable consistent time measurements across processes and applications.
- They are designed to meet different timing needs, such as tracking real-world time, measuring time elapsed since system boot, or counting CPU time consumed by processes.

Types of POSIX Clock

Type	Description
CLOCK_REALTIME	• A settable system-wide clock that measures real time (wall clock time in UTC) • Setting this clock requires appropriate privileges • Use cases include getting the current date and time or setting timestamps for real-world events
CLOCK_MONOTONIC	• A nonsettable system-wide clock that represents monotonic time since the system was booted • Useful for measuring durations, such as profiling code execution or timeout calculations in programs.
CLOCK_MONOTONIC_RAW	• Similar to CLOCK_MONOTONIC, but provides access to a raw hardware-based time • Useful in high-precision timing where even small adjustments are undesirable
CLOCK_PROCESS_CPUTIME_ID	• This is a clock that measures CPU time consumed by this process • Suitable for tracking process resource usage or performance profiling
CLOCK_THREAD_CPUTIME_ID	• This is a clock that measures CPU time consumed by this thread • Useful in multi-threaded applications where tracking CPU time per thread is needed

gettimeofday()

- `gettimeofday()`: get time ⟷ `settimeofday()`: set time

```
#include <sys/time.h>
int settimeofday(const struct timeval *tv,
                 const struct timezone *_Nullable tz);
```

- Used for obtaining the current time by offering microsecond resolution
- The timezone structure tz is no longer used in Linux, so tz is always passed as NULL.

- *tv*: gives the number of seconds and microseconds

```
struct timeval {
  time_t      tv_sec;  /* seconds */
  suseconds_t tv_usec; /* microseconds */
};
```

- Return value
  - Success: zero is returned
  - Error: -1 is returned

https://man7.org/linux/man-pages/man2/settimeofday.2.html

gettimeofday(): Example

gettime.c

```c
#include <stdio.h>
#include <sys/time.h>

int main() {
    struct timeval start, end;
    double elapsed = 0;

    gettimeofday(&start, NULL);

    for (long i = 0; i < 10000000; i++);

    gettimeofday(&end, NULL);

    printf("Start: seconds=%ld useconds=%ld\n", (long)start.tv_sec, (long)start.tv_usec);
    printf("End: seconds=%ld useconds=%ld\n", (long)start.tv_sec, (long)start.tv_usec);

    elapsed = (end.tv_sec - start.tv_sec) * 1000.0;
    elapsed += (end.tv_usec - start.tv_usec) / 1000.0;
    printf("Elapsed: %.6f ms\n", elapsed);

    return 0;
}
```

```
yunmin@peace:~/ch10$ ./gettime
Start: seconds=1731299291 useconds=867700
End: seconds=1731299291 useconds=867700
Elapsed: 33.784000 ms
```

clock_gettime()

- `clock_gettime()`: get time of the specified clock ⟷ `clock_settime()`

```
#include <time.h>
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

  - Retrieve the time of the specified clock *clk_id*
  - *clk_id*: the identifier of the particular clock on which to act. A clock may be system-wide and hence visible for all processes, or per-process if it measures time only within a single process.

- *tp*: gives the number of seconds and nanoseconds

```
struct timespec {
  time_t   tv_sec;  /* seconds */
  long     tv_usec; /* nanoseconds */
};
```

- Return value
  - Success: zero is returned
  - Error: -1 is returned

https://man7.org/linux/man-pages/man3/clock_gettime.3.html

clock_gettime(): Example #1

clkgettime1.c

```c
#include <stdio.h>
#include <time.h>

int main() {
    struct timespec start, end;
    double elapsed = 0;

    clock_gettime(CLOCK_MONOTONIC, &start);

    for (long i = 0; i < 10000000; i++);

    clock_gettime(CLOCK_MONOTONIC, &end);

    printf("Start: seconds=%ld nanoseconds=%ld\n", (long)start.tv_sec, start.tv_nsec);
    printf("End: seconds=%ld nanoseconds=%ld\n", (long)end.tv_sec, end.tv_nsec);

    elapsed = (end.tv_sec - start.tv_sec) * 1000.0;
    elapsed += (end.tv_nsec - start.tv_nsec) / 1000000.0;
    printf("Elapsed: %.6f ms\n", elapsed);

    return 0;
}
```

```
yunmin@peace:~/ch10$ ./gettime
Start: seconds=1731299474 useconds=742166
End: seconds=1731299474 useconds=742166
Elapsed: 34.442000 ms
yunmin@peace:~/ch10$ ./clkgettime1
Start: seconds=848619 nanoseconds=900973055
End: seconds=848619 nanoseconds=935310756
Elapsed: 34.337701 ms
```

clock_gettime(): Example #2

clkgettime2.c

```c
#include <stdio.h>
#include <time.h>

int main() {
    clockid_t clocks[] = {
        CLOCK_REALTIME,
        CLOCK_MONOTONIC,
        CLOCK_MONOTONIC_RAW,
        CLOCK_PROCESS_CPUTIME_ID,
        CLOCK_THREAD_CPUTIME_ID,
    };
    int i;

    for (i = 0; i < 5; i++) {
        struct timespec ts;
        int ret;

        ret = clock_gettime(clocks[i], &ts);
        if (ret)
            perror("clock_gettime");
        else
            printf("clock=%d sec=%ld nsec=%ld\n",
                    clocks[i], ts.tv_sec, ts.tv_nsec);
    }
}
```

```
yunmin@peace:~/ch10$ ./clkgettime2
clock=0 sec=1731299598 nsec=521924244
clock=1 sec=848741 nsec=488663422
clock=4 sec=848743 nsec=387681368
clock=2 sec=0 nsec=1299471
clock=3 sec=0 nsec=1305111
```

sleep()

- `sleep()`: sleep for a specified number of seconds

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

  - Causes the calling thread to sleep either until the number of real-time seconds specified in *seconds* have elapsed or until a signal arrives which is not ignored.
  - Return value
    - Zero if the requested time has elapsed
    - The number of seconds left to sleep, if the call was interrupted by a signal handler

https://man7.org/linux/man-pages/man3/sleep.3.html

usleep()

- `usleep()`: suspend execution for microsecond intervals

```
#include <unistd.h>
int usleep(useconds_t usec);
```

  - Suspends execution of the calling thread for (at least) *usec* microseconds.
  - The sleep may be lengthened slightly by any system activity or by the time spent processing the call or by the granularity of system timers.

  - Return value
    - Success: zero is returned
    - Error: -1 is returned

https://man7.org/linux/man-pages/man3/usleep.3.html

alarm()

- usleep(): set an alarm clock for delivery of a signal

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

  - Arranges for a SIGALRM signal to be delivered to the calling process in *seconds* seconds.
  - If *seconds* is zero, any pending alarm is canceled.
  - In any event any previously set *alarm()* is canceled.
  - Return value
    - The number of seconds remaining until any previously scheduled alarm was due to be delivered
    - Zero if there was no previously scheduled alarm

https://man7.org/linux/man-pages/man2/alarm.2.html

alarm(): Example

alarm.c

```c
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

// Signal handler for timeout
void timeout(int sig)
{
    if (sig == SIGALRM)
        puts("Time out!");

    // generate SIGALRM after 2 seconds
    alarm(2);
}

// Signal handler for key control
void keycontrol(int sig)
{
    if (sig == SIGINT)
        puts("CTRL+C pressed");
}
```

```c
int main(int argc, char *argv[])
{
    int i;
    signal(SIGALRM, timeout);   // for timeout
    signal(SIGINT, keycontrol); // for key control
    alarm(2);      // generate SIGALRM after 2 seconds

    for (i = 0; i < 3; i++)
    {
        puts("wait...");
        sleep(10); // sleep for 10 seconds
    }
    return 0;
}
```

```
yunmin@peace:~/ch10$ ./alarm
wait...
Time out!
wait...
Time out!
wait...
Time out!
yunmin@peace:~/ch10$ ./alarm
wait...
^CCTRL+C pressed
wait...
Time out!
wait...
^CCTRL+C pressed
```

timer_create()

- `timer_create()`: create a POSIX per-process timer

```
#include <signal.h>
#include <time.h>
int timer_create(clockid_t clockid,
                 struct sigevent *_Nullable restrict sevp,
                        timer_t *restrict timerid);
```

  - Creates a new per-process interval timer.
  - *`clockid`*: specifies the clock that the new timer uses to measure time.
  - *`sevp`*: points to a `sigevent` structure that specifies how the caller should be notified when the timer expires
  - *`timerid`*: the ID of the new timer is returned in the buffer pointed to by *`timerid`*.
  - The new timer is initially disarmed.
  - Return value
    - Success: zero and the ID of the new timer is placed in *`*timerid`*
    - Failure: -1

https://man7.org/linux/man-pages/man2/timer_create.2.html

timer_create()

```
#include <signal.h>
struct sigevent {
    int              sigev_notify;  /* Notification type */
    int              sigev_signo;   /* Signal number */
    union sigval     sigev_value;   /* Data passed with notification */

    void           (*sigev_notify_function)(union sigval);
                                    /* Notification function (SIGEV_THREAD) */
    pthread_attr_t *sigev_notify_attributes;
                                    /* Notification attributes */

    /* Linux only: */
    pid_t            sigev_notify_thread_id;
                                    /* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

```
union sigval {                      /* Data passed with notification */
    int              sival_int;     /* Integer value */
    void            *sival_ptr;     /* Pointer value */
};
```

https://man7.org/linux/man-pages/man3/sigevent.3type.html

`timer_create()`

- The `sigevent` structure is used by various APIs to describe the way a process is to be notified about an event (e.g., completion of an asynchronous request, expiration of a timer, or the arrival of a message).
- The `sigev_notify` field specifies how notification is to be performed.
  - SIGEV_NONE: A null notification: don't do anything when the event occurs.
  - SIGEV_SIGNAL: Notify the process by sending the signal specified in *`sigev_signo`*. If the sigal is caught with a signal handler that was registered using the `sigaction` SA_SIGINFO flag
  - SIGEV_THREAD: Notify the process by invoking *`sigev_notify_function`* "as if" it were the start function of a new thread.

https://man7.org/linux/man-pages/man3/sigevent.3type.html

timer_settime()

- `timer_settime()`: arm/disarm state of POSIX per-process timer

```
#include <time.h>
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict new_value,
                  struct itimerspec *_Nullable restrict old_value);
```

  - Arms or disarms the timer identified by *timerid*
  - *new_value*: points to an *itimerspec* structure that specifies the new initial value and the new interval for the timer
  - Each of substructure of the *itimerspec* structure is a *timespec* structure
  - *old_value*: If *old_value* is not NULL, then it points to a buffer that is used to return the previous interval of the timer
  - Return value
    - Success: zero
    - Failure: -1

https://man7.org/linux/man-pages/man2/timer_create.2.html

timer_gettime()

- `timer_gettime()`: fetch state of POSIX per-process timer

```
#include <time.h>
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
```

  - Returns the time until next expiration, and the interval, for the timer specified by *`timerid`*, in the buffer pointed to by *`curr_value`*
  - The time remaining until the next timer expiration is returned in *`curr_value->it_value`*; this is always a relative value
  - If the value returned in *`curr_value->it_value`* is zero, then the timer is currently disarmed
  - The timer interval is returned in *`curr_value->it_interval`*.
  - Return value
    - Success: zero
    - Failure: -1

https://man7.org/linux/man-pages/man2/timer_create.2.html

timer_create(): Example

posix_timer.c

```c
// Signal handler for timer expiration
void timer_handler(int sig) {
    if (sig == SIGALRM) {
        printf("Timer expired!\n");
    }
}

int main() {
    struct sigevent sev;
    struct itimerspec its;
    timer_t timerid;
    struct sigaction sa;

    // Set up the signal handler
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = timer_handler;
    sigemptyset(&sa.sa_mask);
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }

    // Create the timer
    sev.sigev_notify = SIGEV_SIGNAL;
    sev.sigev_signo = SIGALRM;
    sev.sigev_value.sival_ptr = &timerid;
    if (timer_create(CLOCK_REALTIME, &sev, &timerid) == -1) {
        perror("timer_create");
        exit(EXIT_FAILURE);
    }

    // Set the timer
    its.it_value.tv_sec = 3;    // First expiration in 3 seconds
    its.it_value.tv_nsec = 0;
    its.it_interval.tv_sec = 2; // Interval of 2 seconds after that
    its.it_interval.tv_nsec = 0;

    if (timer_settime(timerid, 0, &its, NULL) == -1) {
        perror("timer_settime");
        exit(EXIT_FAILURE);
    }

    printf("Timer set. First 3 seconds, then every 2 seconds.\n");

    // Infinite loop waiting for the timer to expire
    while (1) {
        pause();  // Wait until a signal is received
    }

    // Delete the timer (won't actually be reached in this example)
    if (timer_delete(timerid) == -1) {
    ...
```