Semaphores: View of Process

Semaphore contains 0 $\rightsquigarrow$

I want to read...
(P–Operation)

I am allowed to read...
(someone did V–Operation)
Semaphore contained ≥ 1

Theory of Semaphores

invented by Dijkstra 1968

`http://www.cs.utexas.edu/~EWD/transcriptions/EWD01xx/EWD123.html`

critical section: only one process is allowed to enter CS

P–Operation: (dutch ,,passeren")

- process wants to enter CS,
- but is blocked if some other process in CS
- in CS, process allocated the resource

V–Operation: (dutch ,,vrijgeven")

- process leaves CS,
- releases resource

UNIX: Semaphore Set

a vector of n semaphores comprise a semaphore set

semaphore: (semaphore ID, semaphore number)

obtain a semaphore set by `semget()`

operations on semaphore set by `semop()` : **P**, **V**

remove semaphore set by `semctl()`

Semaphore Semantics

- semaphore has integer values
- normal P–Operation corresponds to –1
  (which is blocked if semaphore value = 0)
- normal V–Operation corresponds to $+1$

can use other values than ± 1

P–Operation can be made non–blocking

Code Example: new semaphore set

```
/* create new semaphore set with n semaphores, return semid */
int new_sem(int n)
{
         return semget(IPC_PRIVATE, n, SEM_A | SEM_R );
}
```


Code Example: operation on semaphore set

```
int operation_p(int semid) /* enter critical region */
{
        struct sembuf sb;

        sb.sem_num   = 0;
        sb.sem_flg   = 0;
        sb.sem_op    = -1;
        if (semop(semid, &sb, 1) < 0) /* 1 operation */
        {
                perror("semop() in operation_p()");
                return 0; /* false, error */
        }
        return 1; /* true, success */
}
```


Code Example: delete semaphore set

```
/* delete semaphore set semid */
int delete_sem(int semid)
{
        if (semctl(semid, 0, IPC_RMID) < 0)
        {
                        perror("semctl(sem, 0, IPC_RMID, 0)");
                        return 0; /* error removing semaphore */
        }
        return 1; /* success */
}
```


UNIX: Semaphore Special Features

- semaphores exist after process is terminated
  use `ipcrm` or `semctl()`
- access rights user/group/other for read/alter
- more than 1 semaphores in 1 operation
- counting semaphore instead of binary semaphore
- can UNDO operations if process is terminated

UNIX: Shared Memory Segment

a shared memory segment shared memory ID

allocate a shared memory segment by `shmget()`

obtain the pointer to segment by `shmat()`

perform operations on this segment by using that pointer

remove shared memory segment `shmctl()`

UNIX: SEM, SHM, MSGQ admin commands

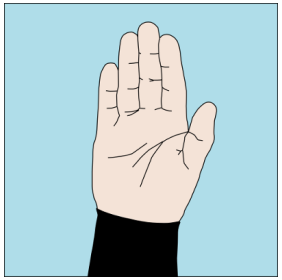
`ipcs` shows these objects (IPC status)

`ipcrm` removes these objects

Signals

A *signal* is a reporting method for exceptional events.

A signal my be viewed as an asynchronous input to a process.



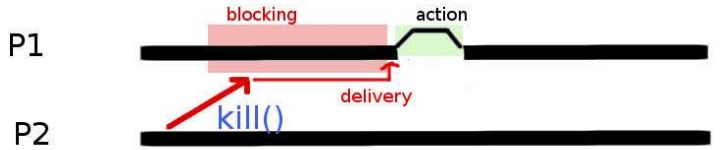
A signal is raised by ...

- an error (by OS kernel)
- an external event (by OS kernel)
- an explicit request (by a process)

$\rightsquigarrow$time of receiving a signal is unpredictable

Signal Delivery

generation of signal by process P_2 with destination P_1



$\longrightarrow$ most signals (may be) blocked

$\longrightarrow$ pending

$\longrightarrow$ delivery (on system call/page fault/clock interrupt)

$\longrightarrow$ action

Signal Action

- accept default action
  - ignore
  - stop
  - terminate
- ignore signal
- install signal handler

`see signal(3)`

Signal Examples

- division by zero
- accessing memory not allocated by the process
  - segmentation fault (invalid access to valid memory)
  - bus error (access to an invalid address)
- I/O errors (reading from pipe which has no writer)
- child exit or stop
- timer expires
- process termination/stopping by user (Strg+c,Strg+z)
- hangup (user shell terminates, notifies all processes)

Signals for the Shell Programmer

avoid hangup signals by starting processes with `nohup`

```
nohup ./long_running_process &
```

catch signals with `trap`

```
trap "rm $TEMP_FILE; exit" SIGHUP SIGINT SIGTERM
```


Sending a Signal

Shell command `kill`.

System call `kill()`.

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

Example:

```
kill -1 9518
kill -HUP 9518
```

send both the *hangup signal* to process 9518

Note: `/bin/kill` is the original – maybe shell built-in command

Signal Types (1)

```
No    Name         Default Action       Description
 1    SIGHUP       terminate process    terminal line hangup
 commonly used for causing servers to reread configuration

 2    SIGINT       terminate process    interrupt program
 STRG+C to terminate process

 3    SIGQUIT      create core image    quit program
 tell process to shutdown gracefully

 4    SIGILL       create core image    illegal instruction

 5    SIGTRAP      create core image    trace trap
 process being debugged has reached a break point
```


Signal Types (2)

```
 6    SIGABRT      create core image    abort program (formerly SIGIOT)
 used when calling abort()

 7    SIGEMT       create core image    emulate instruction executed
 historical reasons, seldom used, meaning varies

 8    SIGFPE       create core image    floating-point exception

 9   *SIGKILL*     terminate process    kill program
 cannot be caught/ignored

10    SIGBUS       create core image    bus error
 CPU detects error on data bus (invalid address)
```


Signal Types (3)

```
11    SIGSEGV      create core image    segmentation violation
 process tries to access a protected memory location

12    SIGSYS       create core image    non-existent system call invoked

13    SIGPIPE      terminate process    write on a pipe with no reader

14    SIGALRM      terminate process    real-time timer expired

15    SIGTERM      terminate process    software termination signal
 tell process to clean up and terminate, default signal of kill command
```


Signal Types (4)

```
16    SIGURG       discard signal       urgent condition on socket
 urgent data on socket (see TCP segment format)

17   *SIGSTOP*     stop process         stop
 cannot be caught/ignored, process waits for SIGCONT

18    SIGTSTP      stop process         stop signal (keyboard)
 STRG+Z on keyboard, process waits for SIGCONT

19    SIGCONT      discard signal       continue after stop
 cannot be ignored but can be caught

20    SIGCHLD      discard signal       child status has changed
 child has stopped or exited
```

Signal Types (5)

```
21    SIGTTIN      stop process         background read attempted
 process waits for SIGCONT

22    SIGTTOU      stop process         background write attempted
 stop only if tty has TOSTOP attribute, process waits for SIGCONT

23    SIGIO        discard signal       I/O is possible on a descriptor
 enabled with fcntl()

24    SIGXCPU      terminate process    cpu time limit exceeded

25    SIGXFSZ      terminate process    file size limit exceeded
```


Signal Types (6)

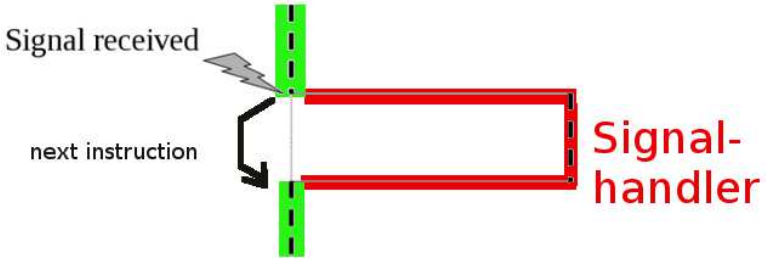
```
26    SIGVTALRM    terminate process    virtual time alarm
 "CPU user time" alarm

27    SIGPROF      terminate process    profiling timer alarm
 "CPU user+system time" alarm

28    SIGWINCH     discard signal       Window size change
 columns or rows of terminal are adjusted

29    SIGUSR1      terminate process    User defined signal 1
30    SIGUSR2      terminate process    User defined signal 2
```


Signal Handler (what is it)



In order to handle a signal, a *signal handler* is needed.

This is a C function with prototype

```
void handler(int sig);
```

The parameter sig contains the number of the signal.

Signal Handler (how to install)

`signal()` or `sigaction()` function.

```
signal(SIGTERM, handler); /* use the handler */
```

install **default** action or **ignore** signal

```
signal(SIGTERM, SIG_DFL); /* set the default action */
signal(SIGTERM, SIG_IGN); /* ignore this signal */
```

Signal Handler (what happens)

when a signal is generated for a process

further occurrences of *this* signal are blocked

after return from the handler() the handled signal is unblocked

the process continues from where it left off when the signal occurred

exception: some system calls are restarted

`open(2), read(2), write(2), sendto(2), recvfrom(2), sendmsg(2), recvmsg(2), ioctl(2), wait(2)`

if data already *transferred*, then they return *partial success*

change system call behaviour with `siginterrupt()`

Signal Handler (child processes)

the child inherits after `fork()` the installed signal handlers

the child resets the *handled* signals after `execve()`

the child ignores signals that are ignored by the parent

if a child exits the parent is sent a `SIGCHLD`

if a process ignores `SIGCHLD`, no zombies will be created