

Semaphores

an IPC mechanism

inter-process communication

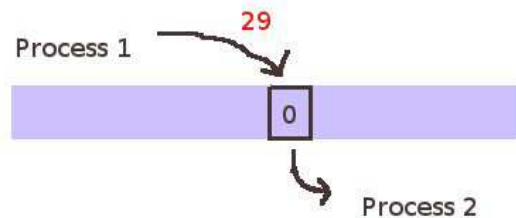
needed if two processes share a common resource, primarily memory

shared memory

Problem

- perhaps process 1 was stopped
 - perhaps process 2 was stopped
 - perhaps one of them runs at lowest priority
 - perhaps one of them delayed because of a I/O problem
 - ...
- process 2 must be stopped before reading until process 1 has written

Shared Memory Problem



assume value 0 in adress 0x10000000

Process 1 writes value 29 to address 0x10000000

Process 2 reads from address 0x10000000

when process 2 reads from 0x10000000, does it read a 0 or a 29 ?

Semaphores: View of Process

Semaphore contains 0 ~>



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



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

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()`

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 */
}
```

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

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: 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()`

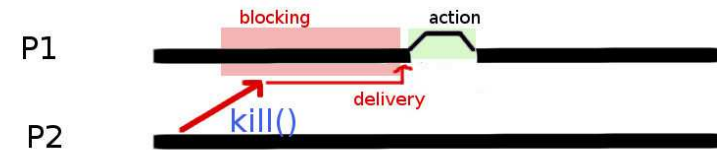
UNIX: SEM, SHM, MSGQ admin commands

`ipcs` shows these objects (IPC status)

`ipcrm` removes these objects

Signal Delivery

generation of signal by process P_2 with destination P_1



→ most signals (may be) blocked

→ pending

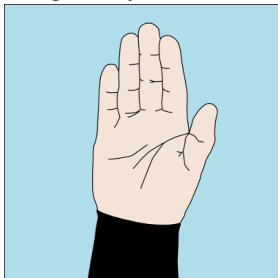
→ delivery (on system call/page fault/clock interrupt)

→ action

Signals

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

A signal may 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)

→ time of receiving a signal is unpredictable

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)

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

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
```

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 (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 (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 (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 (how to install)

signal() or sigaction() function.

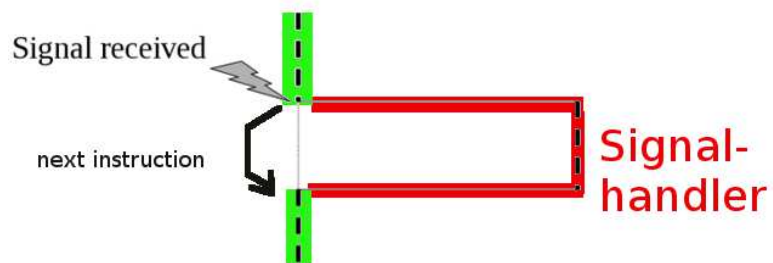
```
signal(SIGTERM, handler); /* use the handler */
```

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

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

```
signal(SIGTERM, SIG_IGN); /* ignore this signal */
```

Signal Handler (what 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 (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), recvmmsg(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

Signal Handler (sigaction)

```
int sigaction(int sig,
              const struct sigaction *act,
              struct sigaction *oact);

struct sigaction {
    union {          /* signal handler */
        void    (*__sa_handler)(int);
        void    (*__sa_sigaction)(int, siginfo_t *, void *);
    } __sigaction_u;
    sigset_t sa_mask;      /* signal mask to apply */
    int      sa_flags;     /* see signal options */
};
```

Signal Handler (why sigaction())

can restore original handling of signal

 can block other signals during execution of handler

Signal Handler (use of sigaction)

```
...
struct sigaction action;
sigset_t signal_mask;

/* all signals to be blocked during handler() */
sigfillset(&signal_mask);
/* fill action structure */
action.sa_handler = handler;
action.sa_mask = signal_mask;
action.sa_flags = 0;
/* install handler */
sigaction (SIGTERM, &action, NULL);
...
```


Signal Handler (print signal names)

```
void psignal(unsigned sig, const char *s);
print message according to signal number sig

char * strsignal(int sig);
return pointer to message according to signal number sig
```

Examples (2)

```
ftpd.c – SIGURG ~> handle urgend TCP data

223 static volatile sig_atomic_t recvurg;
...
2754 static void
2755 sigurg(int signo)
2756 {
2757
2758     recvurg = 1;
2759 }
2760
```

Examples (1)

```
ftpd.c – SIGCHLD ~> wait for child processes

3273 void
3274 reapchild(int signo)
3275 {
3276     while (waitpid(-1, NULL, WNOHANG) > 0);
3277 }
```

Examples (3)

```
ftpd.c – SIGQUIT ~> handle quit from keyboard

666 static void
667 sigquit(int signo)
668 {
669
670     syslog(LOG_ERR, "got signal %d", signo);
671     dologout(1);
672 }
```

Signal Handler (summary)

handler = *exception handling*

the handler should be...

- short: do only one thing
- indicating its use in a global variable `volatile int`
- not time-consuming
- not implementing functional features
- not continue on program bugs (SIGBUS, SIGSEGV, SIGFPE)

`sigaction()` preferred to `signal()`