Signal Processing (Asynchronous Event Processing)

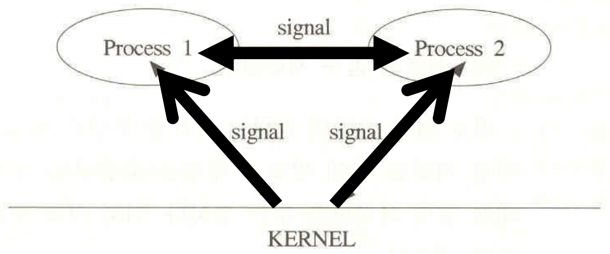
Prof. Jung Guk Kim,

HUFS, Dept. CSE

jgkim@hufs.ac.kr

Signal

- ***A user process/thread can handle an asynchronous urgent event like an interrupt handling by using signals and signal-handlers .***
  - **User's view** : user mode running → signal occurs → user program is interrupted → signal handler (user mode running) → exit or return to the interrupted user program (kind of a **S/W interrupt handling**)
  - **Kernel's view** : an event occurs(ex: **segment fault**, a special interrupt) → mark that "the signal (**SIGSEGMENT**) happened" to the relevant process's signal table in PCB → **just before returning to user mode** of the process, the signal handler will be executed in user mode -> return to the user code: **so there can be some delay (different from interrupt handling (no delay)**)
- ***Signal : kernel → user, user → user***



Signal Handling

- **Signal delivery → mark (mask set) the signal bit in the *pending signal table* of the PCB.**
- **When the process becomes running (by context switch) in the kernel, the process will eventually go to user mode. Just before return to the user mode, the signal will be processes by the signal handler.**
  - So when there is a pending signal, a process's kernel mode running can exist, however, no user mode running. (So from the user's point of view, signal handling is same as an ***S/W interrupt handling***.)
- **A signal that is not handled yet, is called "*a pending signal*".**
- **A signal handling can be temporarily blocked by a user. This is called "*signal blocking*" (similar as *interrupt_disable()*)**

Three Ways of Signal Handling

1. **Kernel defined signal handler (SIG_DFL)**
   - In general, a signal happens when there is an error.
   - So in the default handler, do “exit” or “core-dump & exit”.
2. **Signal ignore (SIG_IGN)**
   - A signal is ignored.
   - However, SIGKILL & SIGSTOP cannot be ignored.
3. **User defined signal handler**
   - A process can register it’s own signal handler.
   - Ex: ^C (kill a process). But by a user defined handler, ^C can do a shutting in a game program.

Signals (1)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGABRT	Program abort (abort())	6	Core-dump & exit
SIGALRM	Timer alarm	14	Exit
SIGBUS	Bus error	10	Core-dump & exit
SIGCHLD	Death of a child process	18	Ignore
SIGCONT	Continue a stopped process	25	Restart/Ignore
SIGEMT	Emulation Trap	7	Core-dump & exit
SIGFPE	Arithmetic exception	8	Core-dump & exit
SIGHUP	TTY disconnected	1	Exit
SIGILL	Not an instruction (an illegal jump to a data section)	4	Core-dump & exit

Signals (2)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGINT	^C	2	Exit
SIGIO	Asynchronous I/O done	22	Exit
SIGIOT	H/W fault	6	Core-dump & exit
SIGKILL	Kill request (shell or syscall: kill (9))	9	Exit
SIGPIPE	Write attempt to a pipe with no reader (broken pipe)	13	Exit
SIGPOLL	A pollable event occurred in I/O (poll())	22	Exit
SIGPROF	Profiling timer expired	29	Exit
SIGPWR	Power failure	19	Ignore
SIGQUIT	^Z, quit	3	Core-dump & exit

Signals (3)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGSEGV	illegal memory access (pointer, access kernel's or other process's area, write on read-only area)	11	Core-dump & exit
SIGSTOP	Stop (ex: debugger)	23	Stop
SIGSYS	Illegal system call	12	Core-dump & exit
SIGTERM	kill (1)	15	Exit
SIGTRAP	Trace/Breakpoint Trap	5	Core-dump & exit
SIGTSTP	^Z	24	Stop
SIGTTIN	A background attempts reading	26	Stop
SIGTTOU	A background attempts writing	27	Stop

Signals (4)

Signal name	*Reason of a signal*	*value*	*Default handler*
SIGURG	An urgent socket event	21	Ignore
SIGUSR1	User defined signal 1	16	Exit
SIGUSR2	User define signal 2사용자 정의 신호2	17	Exit
SIGVTALRM	Virtual timer alarm	28	Exit
SIGWINCH	Size change in a tty window	20	Ignore
SIGXCPU	CPU time-limit expire	30	Core-dump & exit
SIGXFSZ	File size-limit violation	31	Core-dump & exit

signal(2) function

```
// set a user defined signal handler
#include <signal.h>
typedef void (*sighandler_t)(int);  // pointer to a void function
sighandler_t signal (int signum, sighandler_t handler);

    inputs:
        - signum : signal number
        - sighandler : user defined signal handler function
    return:
        - normal : old signal handler의 address
        - error : SIG_ERR
```

After the first signal reception, in some Oses the user defined signal handler is reset to SIG-DFL (some OSes keep the user defined handler)

Using a Signal Handler(1)

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
static void sigcatcher(int);
void (*was)(int);

int main(void)
{
    if( was = signal( SIGINT, sigcatcher ) == SIG_ERR) {
        perror(“SIGINT”);
        exit(1);
    }
    while(1) pause();        // block until any signal happens
}
```

Using a Signal Handler(2)

```
static void sigcatcher( int signo )
{
     switch( signo) {
          case SIGINT :
                    printf("PID %d caught signal SIGINT.\n", getpid());
                    signal(SIGINT, was); // dependent on Linux, bsd versions
                    break;
          default :
                    fprintf(stderr, "something wrong\n");
                    exit(1);
     }
}


$ ./a.out
^CPID 22986 caught signal SIGINT.
^C$
```

kill( ): sending a signal to a process

```
#include <sys/types.h>
#include <signal.h>

int kill ( pid_t pid, int sig);
    input:
        - pid : process id
        - sig : signal number
    return:
        - normal : 0
        - error : -1
```

pid	*Receiver process*
>0	To the process with pid
0	To all processes in my group
-1	To every process except for the init(pid =1) process (broadcasting)

kill( ) usage

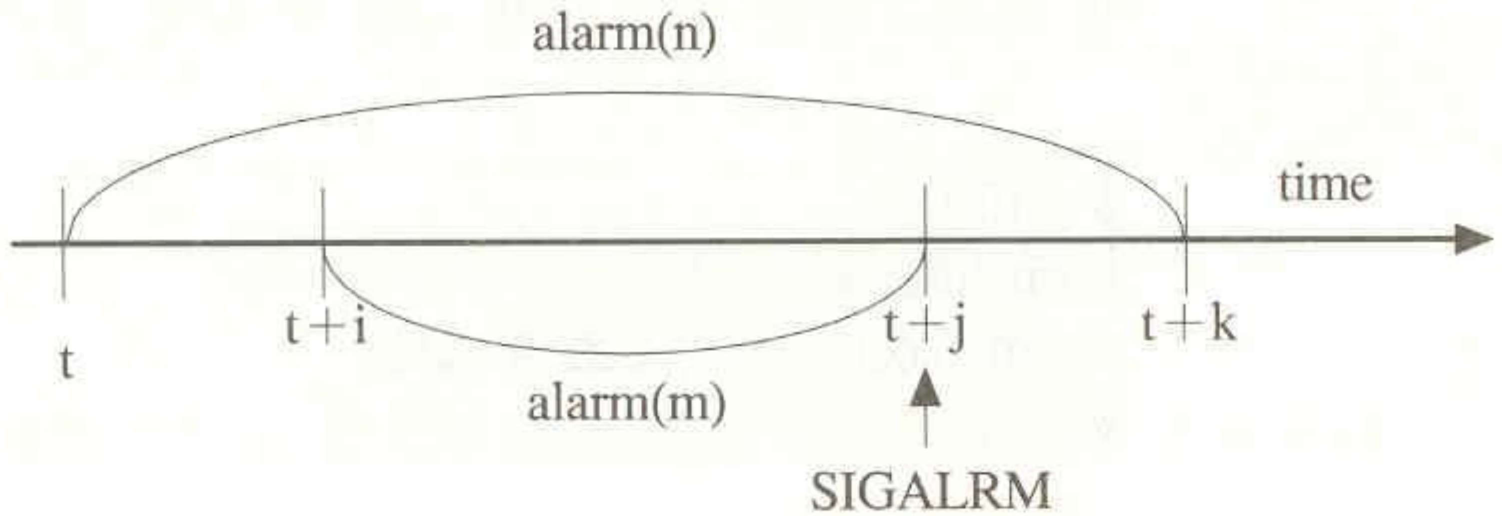
```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
     int pid;
     if ((pid = fork()) == 0) {  // child
               while(1);
     } else {             // parent
               kill (pid, SIGKILL);     // kill the child, signal number = 9
               printf("send a signal to the child\n");
               wait();
               printf("death of child\n");
     }
}
```

alarm( )

```
#include <unistd.h>

unsigned alarm(unsigned sec);
    input:
        - sec : after sec, send SIGALRM to me
    return:
        - there is a previous alarm( ) call : time left to the alarm time of the
          previous alarm() call
        - the first alarm( ) call : 0
```

Duplicated alarm() calls



alarm( ) usage

```
// File #1
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
static void sig_catcher(int);
volatile int alarmed = 0;
int main()
{
    int pid;
    signal(SIGALRM, sig_catcher);
    alarm(3);
    do something;
    while(alarmed == 0);
    printf("after alarm in main\n");
}
```

```
// File # 2
void sig_catcher(i)
{
    alarmed = 1;
    alarm(0);
}
```

// **volatile**: tell the optimizing complier that “do not optimize (var.-> constant”, “keep this as a variable” when it complies the file #1.

sleep( )

#include <unistd.h>

unsigned int sleep (unsigned int seconds);

input: seconds : waiting time in second,
the process will be blocked for the seconds

return: 0 or time left to the wakeup time

- When being unblocked, a SIGALRM happens (same as alarm())!
- So be careful in using the ***sleep()*** and ***alarm()*** together! (do not use them together)

- cf:
  - ***nanosleep (nano-sec);***
  - ***usleep (micro-sec);***

Signal Handling during a System call

- **Type 1: finish the system call -> signal handling**
  - After the waiting I/O has been completed, the signal is handled.
  - The case when I/O completion is guaranteed such as disk I/O (disk file read())
  - Linux's blocked state = "***TASK_UNINTERRUPTIBLE*** "
- **Type 2: stop the system call -> signal handling -> (restart or error return)**
  - The process is waken up (***be ready***) and the do the signal handling first, (this means the system call read() is interrupted.)
  - Linux's blocked state = "***TASK_INTERRUPTIBLE*** "
  - After the signal handling,
    - Linux default: recall the I/O system call (ex: read(), getchar(), etc.)
    - Other versions (UINX): ***error return*** from the I/O system call
- **For both cases,**
  - If ***siginterrupt (signal_no, TRUE/FALSE)*** is called by ***TRUE***, the I/O system call is interrupted (error return) after the signal handling. (***FALSE***: restart the system call after the signal handling)

Example Program (alarm() & getchar())

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

#define TIMEOUT         5       // login time limit = 5 sec.
#define MAXTRIES        5       // retry login five times when timeout
#define LINESIZE        100     // login name/passwd buffer size
#define CTRL_G          '\007‘  // bell
#define TRUE            1
#define FALSE           0

volatile int timed_out;   // set when an alarm occurs
char myline [LINESIZE];   // character buffer
void sig_catch(int);            // alarm signal handler
```

Example Program (alarm() & getchar())

```
char *quickreply(char *prompt) {

    void (*was)(int);
    int ntries, i;
    char *answer;

    was = signal (SIGALRM, sig_catch);
    siginterrupt (SIGALRM, 1);
    // set error return when a signal
    occurs

    for (ntries = 0; ntries <MAXTRIES;
    ntries++) {  // retry loop

        timed_out = FALSE;

        printf("\n%s > ",prompt);
        fflush(stdout);
        alarm(TIMEOUT);
```

```
    for (i = 0; i < LINESIZE; i++) {
            if ((myline[j] = getchar()) < 0)
                break;  // error return by alarm
            if (myline[j] == '\n') {
                    myline[j] = 0;
                break;  // end of line input
            }
    }
// normal case or alarm case here

    alarm(0);  // reset the alarm
    if (!timed_out)  // normal case
            break;
  }  // end of retry loop
// normal or fail 5 times
    answer = myline;

    signal(SIGALRM,was);
    return(ntries == MAXTRIES ? ((char *)
    0) : answer);
}
```

Example Program (alarm() & getchar())

```
void sig_catch (int sig_no)
{
    timed_out = TRUE;
    putchar (CTRL_G);   // ring a bell
    fflush (stdout);                        // insure that the bell-ring
    signal (SIGALRM, sig_catch);  // reinstall the signal handler
}

int main()
{
    quickreply(“login-name:");
}
```

Signals & Threads

- In a multithread environment, a created thread ***inherits the signal handler*** of the thread sibling group (eg. main).
- When a signal happens, the signal is delivered to any one thread (not a fixed one) of the thread group.
- If a thread blocked the signal by using the ***pthread_sigmask()***, the signal will be delivered to one of the other threads.
- If all threads blocked the signal, the signal will be queued.

Interval Timer
(itimer & POSIX timer)

Interval Timer (itimer)

- **An interval timer generates a SIGALRM signal periodically.**
- **Used to implement a periodic job.**

```
#include <sys/time.h>
int setitimer (int which,                              // timer type
               const struct itimerval *value,          // new interval
               struct itimerval *oval);                // old interval, maybe NULL
int getitimer (int which,                              // timer type
               struct itimerval *oval);                // return current setting
```

which	description
ITIMER_REAL	Time in real: at expiration, SIGALRM
ITIMER_VIRTUAL	Time in user mode: at expiration, SIGVALRM
ITIMER_PROF	Process running time (user mode + kernel mode), SIGPROF

Interval Timer

```
struct itimerval {
    struct timeval it_interval;    // periodic interval after the 1st alarm
    struct timeval it_value;   // first interval
}

struct timeval {
    long tv_sec;     // seconds
    long tv_usec;   // micro seconds
}

if  it_interval = 0;    // one time timer
if  it_value = 0;           // off the itimer
```

Interval Timer

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

void alarm_handler (int signo)
{
    printf ("Timer hit\n");
    do the periodic job;
}
```

```
int main()
{
    struct itimerval delay;
    int ret;
    signal (SIGALRM, alarm_handler);
    delay.it_value.tv_sec = 5;  // first alarm
    delay.it_value.tv_usec = 0;
    delay.it_interval.tv_sec = 1;  // periodic
    delay.it_interval.tv_usec = 0;
    ret = setitimer (ITIMER_REAL, &delay, NULL);
    if (ret) { perror ("setitimer"); return; }
    while (1) {
        pause();
    }
}
// 단점: alarm()이나 sleep()과같이 사용 하면 혼선
```

POSIX Timer (Advanced)

- **The *POSIX Timer* is a more advanced & controllable timer.**
- **Merits**
  - One process/thread-group can have multiple timers.
  - Instead of the ***SIGALRM***, another signal can be specified to be used. (***SIGRTMIN ~ SIGRTMAX***) : no conflict with ***sleep(), alarm().***
  - signal hander: ***function or thread***: selectable.
  - One can get ***overrun-count*** at every timer's tick.
- **Low portability**
- **See man. pages;**
  - ***timer_create(,,,);***
  - ***timer_settime(,,,);***
  - ***timer_gettime(,,,);***
  - ***timer_getoverrun (,);***
  - ***timer_delete();***
- **At compile time, *"-lrt"* must be added. (rt library)**

Example (POSIX Timer) (1/3)

```
#include <signal.h>
#include <time.h>
void kernel::StartClock ()
{  // use the POSIX real-time timer
     timer_t   timerid;                        // timer id 저장소
     struct  sigevent      sigev;          // timer의 발생 signal의 종류 지정,
                                        // signal handler는  function인지 thread인지? 지정
     struct  itimerspec   itval, oitval;      //  timer의 interval과 start time 지정 structure (new&old)
     struct  sigaction     newact;         // 임의 signal  발생 시의 signal handler function 정보 지정

// signal-handler set up  for SIGRTMIN  with  struct sigaction  newact
     sigemptyset (&newact.sa_mask);      // clear signal  event structure
     newact.sa_flags = SA_SIGINFO;        // signal handler will use 3 arguments-format
     newact.sa_sigaction = ClockHandler;        // signal handler name
     sigaction (SIGRTMIN, &newact, NULL);  // “SIGRTMIN” signal의 signal-handler 지정
```

Example (POSIX Timer) (2/3)

```
// timer set up with  struct  sigevent   sigev
     sigev.sigev_notify = SIGEV_SIGNAL;    // signal handler is a function, (not a thread)
     sigev.sigev_signo = SIGRTMIN;            // timer's signal is "SIGRTMIN"
     sigev.sigev_value.sival_ptr = &timerid;            // timer id 저장소의 주소
     timer_create (CLOCK_REALTIME, &sigev, &timerid);
                              // create a POSIX timer, signal = "SIGRTMIN",
                              // use a signal handler (not a thread)


// timer interval set up with struct  itimerspec  itval
     itval.it_value.tv_sec = 0;
     itval.it_value.tv_nsec = (long)10*(1000000L);     // the first tick
     itval.it_interval.tv_sec = 0;
     itval.it_interval.tv_nsec = (long)10*(1000000L); // 10 milli-sec interval (100Hz)
     timer_settime (timerid, 0, &itval, &oitval);      // initialize the timer with a new timer-spec
}
```

Example (POSIX Timer) (3/3)

```
void ClockHandler (int sig, siginfo_t *info, void * context)
{
    int n_overrun;
    n_overrun = timer_getoverrun (timerid);        // POSIX clock overrun count

    if (n_overrun >= 1) {
        printf("clock overrun = %d\n", n_overrun);
        fflush(stdout);
    }
    // do some periodic things or wakeup a periodic thread on time;
}

int main() {
    startclock();
    while(1) pause();
}
```

Program Assignment #1 (POSIX Timer)

- **Making a time-triggered thread management system**
  - *pthread_condition* ***cond_array[10];*** *// for 10 threads in max.*
  - *pthread_mutex* ***API_Mutex;*** *// for multi-thread safeness*
  - *struct TCB* ***TCB_array[10];*** *// thread 주기 정보*

  - Make an API: ***tt_thread_register (period, thread_id)***
    // ***period*** in milli-sec, ***thread_id*** = 0,1,2,… // creation order
    - *pthread_mutex_lock(&API_Mutex);*
    - *TCB[thread_id].period = period;*
    - *TCB[thred_id].thread_id = thread-id*
    - *TCB[thread_id].time_left_to_invoke = period;*
    - *pthread_mutex_unlock(&API_Mutex);*

  - Make an API: ***tt_thread_wait_invocation (thread_id)***
    - *pthread_mutex_lock(&API_Mutex);*
    - *pthread_cond_wait(cond_array[thread_id], &API_Mutex); // wait*
    - *pthread_mutex_unlock(&API_Mutex);*

  - Time-triggered threads
    - …. // enter as a thread
    - ***tt_thread-register (myperiod, thread-id);***
    - *while (**tt_thread_wait_invocation (thread_id)**) { // wait for the periodic invocation*
    - *do some periodic things; ex) print **thread_id & current time;***
    - *}*

Program Assignment #1 (POSIX Timer)

- Main thread
  - *Initialize **condition & mutex** variables;*
  - *Create several time-triggered threads; // n threads (period = 1,2,3, id = 0,1,2)*
    - » ***num_threads** = n;*
  - *Set & Create an POSIX timer; (tick = 10 msec)*
  - *while (1) { pause(); };*

- POSIX timer handler   // every 10 msec
  - ***pthread_mutex_lock(&API_Mutex);***
  - *for (i = 0; i < num_threads; i++) {*
  - *TCB[i].time_left_to_invoke -= 10;*
  - *if ((TCB[i].time_left_to_invoke -= 10) <= 0) {*
  - *TCB[i].time_left_to_invoke = TCB[i].period;*
  - ***pthread_cond_signal (&cond_array[thred_id]);***
  - *}*
  - *}*
  - ***pthread_mutex_unlock(&API_Mutex);***

Program Assignment #2

- **Pipeline processing with *Pthreads*.**

1. ***Data_acquisition_thread*** : the 1st thread
   - A blood-pressure sensor simulator
   - Run periodically (freq. = 100 Hz), ***turn*** = 0; // local var.
     – By waiting a ***condition-signal*** from the ***timer*** signal handler;
   - At each run,
     – if (***turn***%2 == 0) generate a random number as a ***bp***: [60, 90]; // it's **diastolic** blood pressure.
     – else generate a random number: [110, 150]; // **systolic** bp
   - Enqueue the ***bp*** into the ***bp-***queue; // in a critical section
   - At every 1/10 sec (10Hz, 10 ***bp's*** are queued),
     – wakeup the ***Bp_processing_thread*** **by a signal**;
     – ***turn++;*** // systolic -> diastolic or reverse.

Program Assignment #2

2. ***Bp_processing_thread*** : the 2nd thread

- ***turn*** = 0; // local var., initially, diastolic.
- loop: Wait ***condition-signal*** from the ***Data_acquisition_thread***
- At resume,
  - Dequeue all ***bp*** numbers; // in a critical section
  - ***avg_bp*** = average of them;
  - Save/append ***avg_bp*** to the ***record_file***.
  - Display "diastolic bp = ", ***avg_bp***; or "systolic bp = ", ***avg_bp***;
  - ***turn++***; // systolic -> diastolic or reverse
- repeat loop;

3. *POSIX timer signal handler*

- main() creates a ***POSIX timer*** (period = 100Hz)
- ***timer*** signal-handler wakeup the ***Data_acquisition_thread*** by a signal at every 1/100 sec;