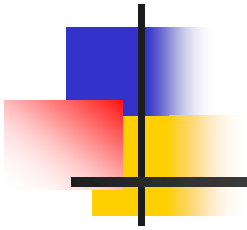
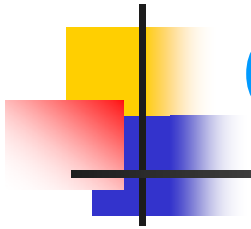
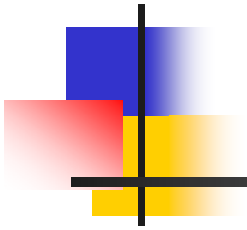


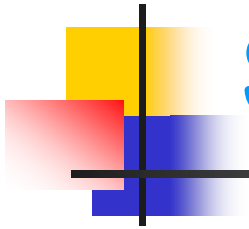
제10장 신호(Signal)



Contents

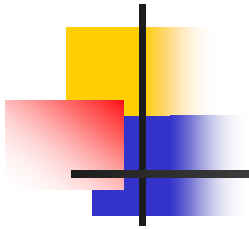


10.1 Signal Concepts



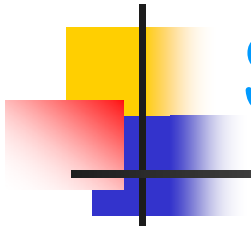
Signals

- Signals are software interrupts from unexpected events
  - a floating-point error
  - a power failure
  - an alarm clock
  - the death of a child process
  - a termination request from a user (Ctrl-C)
  - a suspend request from a user (Ctrl-Z)



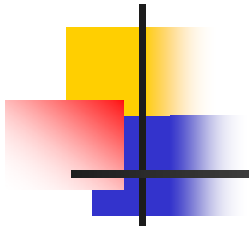
Predefined signals

- 31 signals
  - /usr/include/signal.h
- Every signal has a name
  - begin with 'SIG'
  - SIGABRT: abort signal from abort()
  - SIGALRM: alarm signal from alarm()
- Actions of the default signal handler
  - terminate the process and generate a core(dump)
  - ignores and discards the signal(ignore)
  - suspends the process (suspend)
  - resume the process



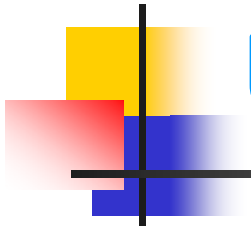
Signal Generation

- Terminal-generated signals
  - CTRL-C → SIGINT
  - CTRL-Z → SIGSTP signal
- Hardware exceptions generate signals
  - divide by 0 → SIGFPE
  - invalid memory reference → SIGSEGV
- `kill()`
  - sends any signal to a process or process group
  - need to be owner or superuser
- Software conditions
  - SIGALRM: alarm clock expires
  - SIGPIPE: broken pipe
  - SIGURG: out-of-band network data



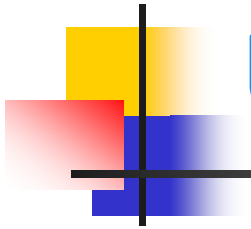
Unix Signals

- SIGART
  - generated by calling the abort function.
- SIGALRM
  - generated when a timer set with the alarm expires.
- SIGCHLD
  - Whenever a process terminates or stops, the signal is sent to the parent.
- SIGCONT
  - This signal(job-control) sent to a stopped process when it is continued.
- SIGFPE
  - signals an arithmetic exception, such as divide-by-0, floating point overflow, and so on
- SIGHUP
  - disconnect detected, session leader terminates



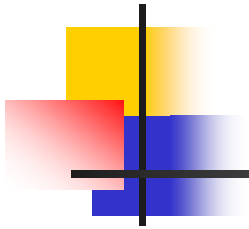
Unix Signals

- SIGILL
  - when the process has executed an illegal hardware instruction
- SIGINT
  - generated by the terminal driver when we type the interrupt key and sent to all processes in the foreground process group
- SIGIO
  - indicates an asynchronous I/O event
- SIGKILL
  - can't be caught or ignored. a sure way to kill any process.
- SIGPIPE
  - If we write to a pipeline but the reader has terminated, SIGPIPE is generated



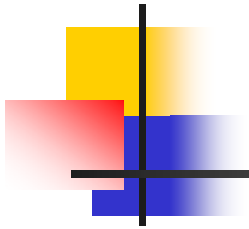
Unix Signals

- SIGSEGV
  - indicates that the process has made an invalid memory reference
- SIGSTOP
  - This signal(job-control) stops a process and can't be caught or ignored
- SIGSYS
  - signals an invalid system call
- SIGTERM
  - the termination signal sent by the kill(1) command by default.
- SIGTSTP
  - Cntl-Z from the terminal driver which is sent to all processes in the foreground process group.



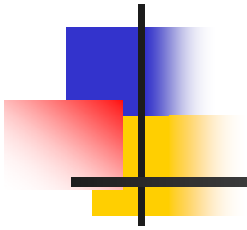
Unix Signals

- SIGTTIN
  - generated by the terminal driver when a background process tries to read from its controlling terminal
- SIGTTOU
  - generated by the terminal driver when a background process tries to write to its controlling terminal
- SIGPWR
  - related to power failure. (read the book for the detail)
- SIGQUIT
  - generated by the terminal driver when we type quit key and sent to all foreground processes
- SIGUSR1, SIGUSR2
  - user-defined signals for use in application programs

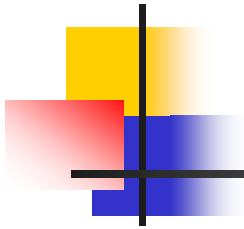


Handling of signals

- Action
  - Process has to tell the kernel
    "if and when this signal occurs, do the following."
- 3 kinds of actions
  - Ignore the signal
    - all signals can be ignored, except SIGKILL and SIGSTOP
  - Catch the signal
    - Call a function of ours when a signal occurs.
  - The default action
    - most are to terminate process



10.2 System call signal()



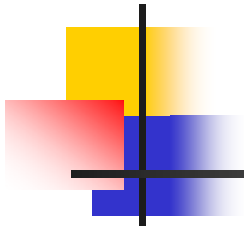
signal( )

- Signal handler 등록
  - 하나의 signal에 대한 hanlder 함수를 등록한다.

- signal(int *signo*, void (**func*)( )))
  - specify the action for a signal
    - *signo* → *func*
  - *func*
    - SIG_IGN
    - SIG_DFL
    - user-defined function
  - return
    - the previous signal handler function

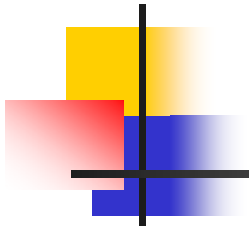
Example: sigusr.c

```
#include  <signal.h>  /* sigusr.c */

static void sig_usr(int signo)    /* argument is signal number */
{
  if (signo == SIGUSR1)
    printf("received SIGUSR1\n");
  else if (signo == SIGUSR2)
    printf("received SIGUSR2\n");
  else
    perror("received signal %d\n", signo);
  return;
}

int main(void) {
  if (signal(SIGUSR1, sig_usr) == SIG_ERR)
    perror("can't catch SIGUSR1");
  if (signal(SIGUSR2, sig_usr) == SIG_ERR)
    perror("can't catch SIGUSR2");

  for ( ; ; )
    pause( );
}
```



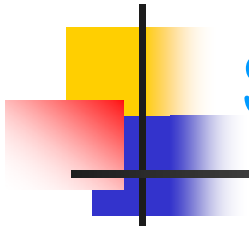
sigusr.c: Results

```
$ a.out &                          start process in background
[1] 4720                           job number & process ID
$ kill -USR1 4720                  send SIGUSR1
received SIGUSR1
$ kill -USR2 4720                  send SIGUSR2
received SIGUSR2
$ kill 4720                        send SIGTERM
[1] + Terminated a.out &
```



Program Startup

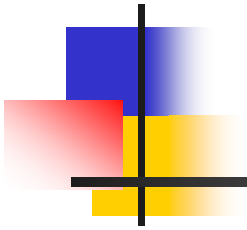
- When a process is forked,
  - the child inherits the parent's signal actions.
- When a program is execed
  - the action of all signals
    → default action or ignore



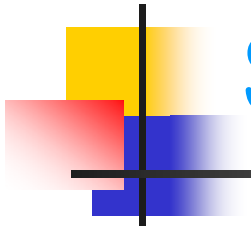
Signal handling: Background process

- Background process
  - Shell sets the action of interrupt and quit signals in background to be ignored
- Catches the signals in foreground process

```
int sig_int(), sig_quit();

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, sig_int);
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
        signal(SIGQUIT, sig_quit);
```

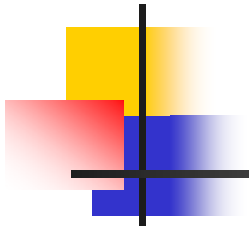


10.3 SIGCHLD



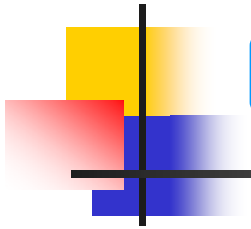
SIGCHLD

- SIGCHLD
  - Whenever a process terminates or stops, the signal is sent to the parent.



Example: Time Limit

```
#include <stdio.h>
#include <signal.h>
int delay;
childHandler( );

int main(int argc; char *argv[ ];)
{
    int pid;
    signal(SIGCHLD,childHandler);
    pid = fork();
    if (pid == 0) {
        execvp(argv[2], &argv[2]);
        perror("Limit");
    } else {
        sscanf(argv[1], "%d", &delay);
        sleep(delay);
        printf("Child %d exceeded limit and is being killed\n, pid);
        kill(pid, SIGINT); }
}
```



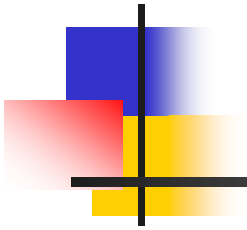
Example: Time Limit

```
childHandler( ) /* Executed if the child dies before the parent */
{
    int childPid, childStatus;
    childPid = wait(&childStatus);
    printf("Child %d terminated within %d seconds\n", childPid,
    delay);
    exit(0);
}
```

- **Usage**

%limit 5 ls

%limit 4 sleep 100

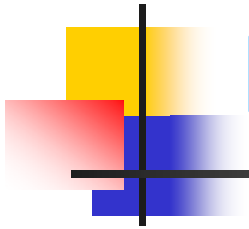


10.4 kill()/raise()

kill() / raise()

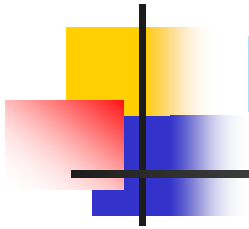
```
#include <sys/types.h>
#include <singnal.h>

int kill(pid_t pid, int signo);
int raise(int signo);
                    Both return: 0 if OK, 1 on error
```

- kill
  - sends a signal to a process or a group of process
- raise
  - allows a process to send a signal to itself



kill()

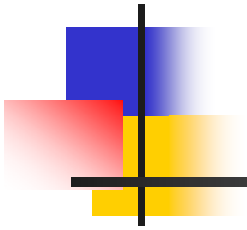
- *pid* > 0 :
  - signal to the process whose process ID is pid
- *pid* == 0 :
  - signal to the processes whose process group ID equals that of sender
- *pid* < 0 :
  - signal to the processes whose process group ID equals abs. of pid
- *pid* == -1 :
  - POSIX.1 leaves this condition unspecified (used as a broadcast signal in SVR4, 4.3+BSD)



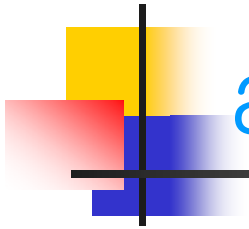
kill()

- Permission to send signals
  - The superuser can send a signal to any process
  - The user ID of the sender == the user ID of the receiver
  - SIGCONT can be sent to any member process of the same session

- NULL signal
  - signo == 0
  - Sent with kill for error checking
    - e.g. to check if a process exists
  - If the process doesn't exist,
    - kill returns -1



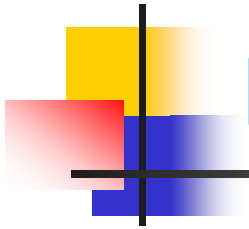
10.5 alarm()



alarm()

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
Returns: 0 or number of seconds until previously set alarm
```

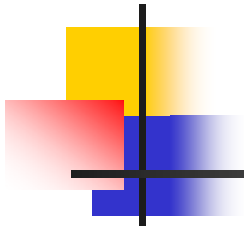
- alarm() sets a timer to expire at a specified time
  - when timer expires, SIGALRM signal is generated,
  - default action of the signal is to terminate the process.
- only one alarm clock per process
  - previously registered alarm clock is replaced by the new value
  - return # of sec. left for the previous alarm
- alarm(0)
  - a previous unexpired alarm is cancelled



pause()

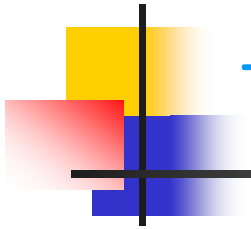
```
#include <unistd.h>

int pause (void);
                                Returns: -1 with errno set to EINTR
```

- suspends the calling process until a signal is caught.
- returns only if a signal handler is executed and that handler returns.



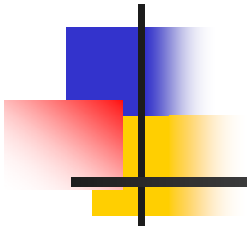
alarm: timeout on blocking operation

- Timeout
  - A read operation on a "slow" device can block for a long time
  - An upper time limit can be imposed using the alarm function and SIGALRM

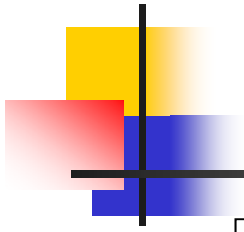


Timeout

```
#include  <signal.h>  /* read1.c */

static void
sig_alrm(int signo) {
  return;  /* nothing to do, just return to interrupt the read */
}

int main(void) {
  int    n;
  char  line[MAXLINE];

  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
          perror("signal(SIGALRM) error");
  alarm(10);
  if ( (n = read(STDIN_FILENO, line, MAXLINE)) < 0)
          perror("read error");
  alarm(0);

  write(STDOUT_FILENO, line, n);
  exit(0);
}
```



10.6 abort()/sleep()

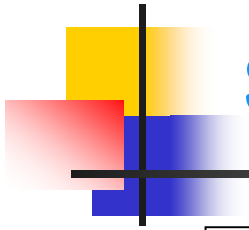


abort()

```
#include <stdlib.h>

void abort(void);
```

This function never returns

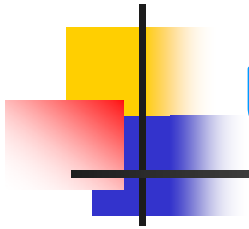
- abnormal program termination
  - This function sends the SIGABRT signal to the process
  - SIGABRT signal handler to perform any cleanup that it wants to do, before the process terminated



sleep()

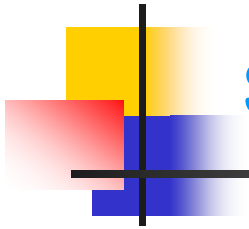
```
#include <signal.h>

unsigned int sleep(unsigned int seconds) ;
Returns: 0 or number of unslept seconds
```

- causes the calling process to be suspended until either
  - The amount of wall clock time has elapsed (returns 0)
  - A signal is caught by the process and the signal handler returns
    (returns the number of unslept seconds)



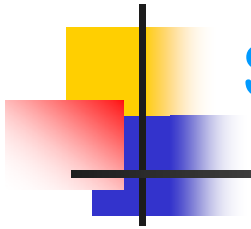
Using alarm/pause to implement sleep

```
#include  <signal.h> /* sleep1.c */
#include  <unistd.h>

static void sig_alrm(int signo) {
  return;  /* nothing to do, just return to wake up the pause */
}

unsigned int sleep1(unsigned int nsecs) {
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
    return nsecs;
  alarm(nsecs);    /* start the timer */
  pause();         /* next caught signal wakes us up */
  return alarm(0);          /* turn off timer, return unslept time */
}
```



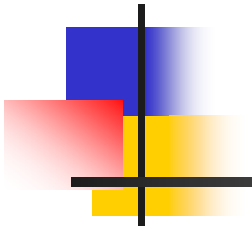
sleep1.c: Implementation Problems

- If the caller of sleep1() already has an alarm set, the alarm is erased by the first call to alarm.
- Modify the action for SIGALRM
  - Save the action and restore it when we're done
- Race condition: between alarm & pause
  - the alarm goes off before the pause()
  - the signal handler may be called before pause
  - the called is suspended forever at pause()

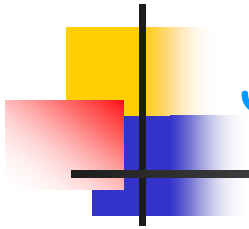


sleep2.c

```
#include  <setjmp.h>  /* sleep2.c */
#include  <signal.h>
#include  <unistd.h>
static jmp_buf  env_alrm;

static void sig_alrm(int signo) {
  longjmp(env_alrm, 1);
}

unsigned int sleep2(unsigned int nsecs) {
  if (signal(SIGALRM, sig_alrm) == SIG_ERR)
     return nsecs;
  if (setjmp(env_alrm) == 0) {
     alarm(nsecs);      /* start the timer */
     pause();           /* next caught signal wakes us up */
  }
  return alarm(0);     /* turn off timer, return unslept time */
}
```



10.7 Job Control Signals



Job-Control Signals

SIGCHLD :	Child process has stopped or terminated
SIGCONT :	Continue process, if stopped
SIGSTOP :	Stop signal (can't be caught or ignored)
SIGTSTP :	Interactive stop signal
SIGTTIN :	Read from control terminal by background processes
SIGTTOUT:	Write to control terminal by background processes

- A program that manages the terminal needs to handle job-control signals
- When type Control-Z (suspend Character), SIGTSTP sent to all foreground processes.

Example: Suspending and Resume

```
#include <signal.h>
#include <stdio.h>
main( ) {
    int pid1, pid2;
    pid1 = fork( );
    if (pid1 == 0) {
        while(1) {
        printf("pid1 isalive\n");
        sleep(1);
        }
    }
    pid2 = fork( );
    if (pid2 == 0) {
        while (1) {
        printf("pid2 is alive\n");
        sleep(1);  }
    }
    sleep(3);                    kill(pid1, SIGSTOP);
    sleep(3);                    kill(pid1, SIGCONT);
    sleep(3);                    kill(pid1, SIGINT);
    kill(pid2, SIGINT);
}
```