SYSTEM PROGRAMMING

Module 5

EGCI 252 SYSTEM PROGRAMMING

Computer Engineering Department
Faculty of Engineering
Mahidol University

Inter-Process Communication

- Single Machine Inter-process Communication
  - Signals
  - Pipes (named and unnamed)
  - System V and POSIX IPC (message queues, semaphore sets and shared memory segments)
- Multiple Machine Inter-process Communication
  - Sockets
  - Remote Procedure Calls (Sun ONC, OSF DCE, Xerox Courier (4.3BSD))
  - Distributed Shared Memory

System V IPC

- System V IPC was first introduced in SVR2, but is available now in most versions of UNIX
- Message Queues represent linked lists of messages, which can be written to and read from
- Shared memory allows two or more processes to share a region of memory, so that they may each read from and write to that memory region
- Semaphores synchronize access to shared resources by providing synchronized access among multiple processes trying to access those critical resources.

What is a Signal?

- A signal is a software interrupt delivered to a process by the OS because:
  - it did something (mostly unexpected)
  - the user did something (pressed ^C)
  - another process wants to tell it something (SIGUSR?)
- A signal is asynchronous, it may be raised at any time (almost)
- Some signals are directly related to hardware (illegal instruction, arithmetic exception, such as attempt to divide by 0)
- Others are purely software signals (interrupt, bad system call, segmentation fault)

Common Signals

Defined in <signal.h> kill –l will list all signals

• SIGHUP (1):	Sent to a process when its controlling terminal has disconnected
• SIGINT (2):	Ctrl-C (or DELETE key)
• SIGQUIT (3):	Ctrl-\ (default produces core)
• SIGILL (4):	Illegal instruction (default core)
• SIGFPE (8):	Floating Point Exception (divide by 0; integer overflow; floating-point underflow)
• SIGSEGV (11):	Segmentation fault (invalid memory reference)
• SIGUSR [1,2]:	User-defined signals (10,12)

More Details in Signals

- SIGABRT: This signal is generated by calling the abort function. The process terminates abnormally.
- SIGALRM: This signal is generated when a timer set with the alarm function expires. This signal is also generated when an interval timer set by the setitimer(2) function expires.
- SIGINT: This signal is generated by the terminal driver when we type the interrupt key (often DELETE or Control-C). This signal is sent to all processes in the foreground process group. This signal is often used to terminate a runaway program, especially when it's generating a lot of unwanted output on the screen.
- SIGKILL: This signal is one of the two that can't be caught or ignored. It provides the system administrator with a sure way to kill any process.
- SIGSTOP: This job-control signal stops a process. It is like the interactive stop signal (SIGTSTP), but SIGSTOP cannot be caught or ignored.
- SIGTERM: This is the termination signal sent by the kill(1) command by default.
- SIGUSR1: This is a user-defined signal, for use in application programs.
- SIGUSR2: This is another user-defined signal, similar to SIGUSR1, for use in application programs.

More Details in Signals (Cont.)

- SIGCHLD: Whenever a process terminates or stops, the SIGCHLD signal is sent to the parent. By default, this signal is ignored, so the parent must catch this signal if it wants to be notified whenever a child's status changes. The normal action in the signal-catching function is to call one of the wait functions to fetch the child's process ID and termination status.

Earlier releases of System V had a similar signal named SIGCLD (without the H). The semantics of this signal were different from those of other signals, and as far back as SVR2, the manual page strongly discouraged its use in new programs. (Strangely enough, this warning disappeared in the SVR3 and SVR4 versions of the manual page.) Applications should use the standard SIGCHLD signal, but be aware that many systems define SIGCLD to be the same as SIGCHLD for backward compatibility. If you maintain software that uses SIGCLD, you need to check your system's manual page to see what semantics it follows.

Four things to do when a signal occurs

1. Ignore a signal *
2. Do whatever we want to handle the condition via a signal handler routine

   this is called the disposition of the signal, or the action associated with a signal :

   - ✓ Clean up and terminate
   - ✓ Handle Dynamic Configuration (SIGHUP)
   - ✓ Report status, dump internal tables
   - ✓ Toggle debugging on/off
   - ✓ Implement a timeout condition
3. Block the signal. In this case, the OS queues signals for possible later delivery
4. Let the default action apply (usually process termination)

* SIGKILL and SIGSTOP can't be ignored to provide the superuser with a way of either killing or stopping any process. Also, if we ignore some of the signals that are generated by a hardware exception (such as illegal memory reference or divide by 0), the behavior of the process is undefined.

Signal Handling

- Two includes: <sys/types.h> and <signal.h>
- void (*signal(int signo, void (*func) (int))) (int)
- Function handler can either be:
  - a function (that takes a single int which is the signal)
  - the constant SIG_IGN is telling the system to ignore the signal
  - the constant SIG_DFL is setting the action associated with the signal to its default value
- Signal will return SIG_ERR in case of error

Signal Handling (Cont.)

- Suspending the calling process until a signal is received:
  - int pause(void); // (#include <unistd.h>)
- Sending signals (2 forms)
  - int kill(pid_t pid, int sig); // send a signal to any process
  - int raise(int sig); // send a signal to itself
- Printing out signal information (#include <siginfo.h>)
  - void psignal(int sig, const char *s);

Alarming Signals

- SIGALRM can be used as a kind of “alarm clock” for a process
- By setting a disposition for SIGALRM, a process can set an alarm to go off in x seconds with the call:
  - #include <unistd.h>
  - unsigned int alarm(unsigned int seconds);

Signal Handling : Example

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

void my_alarm(int sig)
{ for (int i = 0; i < 5; i++) printf(“Alarm!!!  \n”); }

int main(void)
{
    int pid;
    printf(“Alarm clock is starting…\n”);

    if (pid = fork()) == 0)
   { sleep(3); kill(getppid(), SIGALRM); exit(0); }
    printf(“Waiting for alarm…\n”);
    (void) signal(SIGALRM, my_alarm);
    pause(); printf(“Done!\n”); exit(0);
}
```

Signal Handling: Example (More)

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

/* one handler for both signals */
static void sig_usr(int);

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();
}
```

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

Signal Handling: Example (More)

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

static void sig_cld(int);

int main()
{
    pid_t pid;
    if (signal(SIGCLD, sig_cld) == SIG_ERR)
         perror("signal error");
    if ((pid = fork()) < 0) perror("fork error");
    else if (pid == 0) { /* child */ sleep(2); _exit(0); }
    pause(); /* parent */
    exit(0);
}
```

```
static void sig_cld(int signo)
/* interrupts pause() */
{
    pid_t pid; int status;
    printf("SIGCLD received\n");
    /* reestablish handler */
    if (signal(SIGCLD, sig_cld) == SIG_ERR)
        perror("signal error");
     /* fetch child status */
     if ((pid = wait(&status)) < 0)
         perror("wait error");
    printf("pid = %d\n", pid);
    return;
}
```

Reliable and Unreliable Signal APIs

- Signal model provided by AT&T Version 7 was not reliable, meaning that signals could get “lost” on the one hand, and programs could not turn off selected signals during critical sections, on the other hand.
- BSD 4.3 and System V Release 3 delivered reliable signals, which solved many of the problems with signals present in Version 7.
- And if that weren’t enough, SVR4 introduced POSIX signals.

BSD and SysV Handle Unreliability Issue In Incompatible Ways

- Berkeley Unix 4.2BSD responded with inventing a new signal API, but it also rewrote the original signal() function to be reliable
- Thus, old code that used signal() could now work unchanged with reliable signals, optionally calling the new API (sigvec(), etc.)
- Luckily, few programmers used the new (incompatible) API, most stuck with signal() usage

BSD and SysV Handle Unreliability Issue In Incompatible Ways (Cont.)

- AT&T SVR3 provided reliable signals through a new API, and kept the older signal() code unreliable (for backward compatibility reasons)
- Introduced a new primary function:
  - void (*sigset(int sig, void (*handler)(int)))(int)
  - Since sigset accepted the same parameters as before:
    - #define signal sigset /* would port older or BSD4.2 code */
- Introduced a new default for disposition: SIG_HOLD
  (in addition to SIG_DFL, SIG_IGN)

BSD and SysV Handle Unreliability Issue In Incompatible Ways (Cont.)

- SVR3 added its own set of new functions for reliable signals:
  - int sighold(int sig);   /*adds sig to the signal mask disposition */
  - int sigrelse(int sig);   /* removes sig from the signal mask disposition, and waits for signal to arrive (suspends) */
  - int sigignore(int sig);   /* sets disposition of sig to SIG_IGN */
  - int sigpause(int sig);   /* combination of sigrelse and pause(), but safe */

POSIX Signals

- Uses the concept of signal sets from 4.2BSD
- A signal set is a bitmask of signals that you want to block, i.e., signals that you specifically don't want to handle
- Each bit in the bitmask (an array of 2 unsigned longs) corresponds to a given signal (i.e., bit 10 == SIGUSR1)
- All signals not masked (not blocked) will be delivered to your process
- In POSIX signals, a blocked signal is not thrown away, but buffered as pending, should it become unmasked by the process at some later time

Central POSIX Functions

- int sigemptyset(sigset_t * set);
  - Zeros out the bitmask
- int sigfillset(sigset_t * set);
  - Masks all signals
- int sigaddset(sigset_t * set, int signo);
  - Adds a particular signal to the set
- int sigdelset(sigset_t * set, int signo);
  - Unmasks signo from the set

POSIX sigaction

int sigaction (int sig, const struct sigaction *iact, struct sigaction *oact);

This function allows us to examine or modify the action associated with a particular signal.

```
struct sigaction {
        __sighandler_t sa_handler;
        void (*sa_sigaction)(int, siginfo_t *, void *);
        unsigned long sa_flags
        …
        sigset_t sa_mask;  //set of signals to be BLOCKED
};
```

- sa_flags
  - * SA_RESTART flag to automatically restart interrupted system calls
  - * SA_NOCLDSTOP flag to turn off SIGCHLD signaling when children die.
  - * SA_RESETHAND clears the handler (ie. resets the default) when the signal is delivered (recidivist).
  - * SA_NOCLDWAIT flag on SIGCHLD to inhibit zombies.
  - * SA_SIGINFO flag indicates use value in sa_sigaction over sa_handler

sigaction( ) : Example

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


void rx_int(int sig)
{ printf(“SIGINT - %d Received!!!\n”, sig); }


int main(void)
{
   struct sigaction act;


   act.sa_handler = rx_int;
   sigemptyset(&act.sa_mask);
   act.sa_flags = 0;


   sigaction(SIGINT, &act, 0);


   while(1) { printf(“Running…\n”); sleep(1); }
}
```

Reentrant Functions

- A computer program or routine is described as reentrant if it can be safely called recursively or from multiple processes. To be reentrant, a function must hold no static data, must not return a pointer to static data, must work only on the data provided to it by the caller, and must not call non-reentrant functions.

Reentrant Functions: Example

Both functions f and g are not reentrant

```
int g_var = 1;
int f() { g_var = g_var + 2; return g_var; }
int g() { return f() + 2; }
int main() { g(); return 0; }
```

In the example, f depends on a global variable g_var; thus, if two threads execute it and access g_var concurrently, then the result varies depending on the timing of the execution. Hence, f is not reentrant. Neither is g; it calls f, which is not reentrant.

Reentrant Functions: Example (Cont.)

Both functions f and g are reentrant

```
int f(int i)
{ int priv = i; priv = priv + 2; return priv; }

int g(int i)
{ int priv = i; return f(priv) + 2; }

int main() { g(1); return 0; }
```

Both of the new versions are designed to leave the global variable g_var alone -- to have a parameter passed to them, work on it privately, and return the result, not to go out and change some possibly-shared object.

Reentrant Functions (Cont.)

- Reentrant functions are those functions which are safe for reentrance:
  - Scenario: a signal SIGUSR1 is received in the middle of myfunc().
  - The handler for SIGUSR1 is called, which makes a call to myfunc()
  - myfunc() has just been “reentered”
- A function “safe” for reentrance is one that:
  - defines no static data
  - calls only reentrant functions or functions that do not raise signals

POSIX Reentrant-Safe Functions

- alarm, sleep, pause
- fork, execle, execve
- stat, fstat
- open, close, creat, lseek, read, write, fcntl, fstat
- sigaction, sigaddset, sigdelset, sig* etc.
- chdir, shmod, chown, umask, uname

End of Module