Review

- The waitpid() System Call
- The system() System Call
- Concept of Signals
  - Linux Signals
  - Signal() System Call
  - The kill() and raise() System Call
  - The alarm() System Call
  - The pause() System Call

Preview

- The `pause()` System call
- The `signal()` system call
- Signal set
- The `sigprocmask()` system call
- The `sigaction()` system call
- Interprocess Communication
- The `sigsuspend()` system call
- The `abort()` system call

The pause() System Call

- The pause() system call suspends the calling process until a signal is caught

```
#include <unistd.h>
int pause (void);

Return -1 with error
```


The pause() System Call

```
/* alarm1.c: demonstrate a signal system call*/
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ding (int sig)
{
   printf("alarm fired signal number = %d\n", sig);
}

int main()
{
   printf("set alram 5 second for a process\n");
   alarm(5);
   signal (SIGALRM, ding);
   pause();
   exit(0);

}
```


The signal() System Call

```
#include <signal.h>
void (*signal(int signo, void (*func(int)))(int);
signo: name of signal SIG…
func: pointer to siglal handler function
```

- Function **signa**l accept two arguments and return a pointer to a function that returns nothing.
- Second argument is pointer to a function that take a single integer argument and return nothing.


```
/* sig_talk2.c: demonstrate a signal system call*/
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static int alarm_fired = 0;

void ding (int sig)
{
   alarm_fired = 1;
}
int main()
{
   pid_t pid;
   printf("alarm application start \n");
   if ((pid = fork() <0))
   {
      perror ("fork error");
      exit (1);
   }
   else if (pid == 0) /* child process */
   {
      sleep(5);
      kill(getppid(), SIGALRM);
      exit(0);
   }
   else /* parent process */
   {
      printf("waiting for alarm \n");
      signal (SIGALRM, ding);
      pause();
      if (alarm_fired)
         printf("Ding\n");
      printf("done\n");
      exit (0);
   }
}
```

```c
/* sig_talk1.c */
/* a process keep running waiting for a signal */

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

static void sig_usr(int); /* signal handler */

int main(void)
{
   if (signal(SIGUSR1, sig_usr) == SIG_ERR)
   {
      printf("can't catch SIGUSR1");
      exit(1);
   }
   if (signal(SIGUSR2, sig_usr) == SIG_ERR)
   {
      printf("can't catch SIGUSR2");
      exit(1);
   }
   for ( ; ; )
      pause(); /* can wait for signal */
}

/* sinal handler must have one single integer */
static void sig_usr(int signo)
{
   if (signo == SIGUSR1)
      printf("received SIGUSR1\n");
   else if (signo == SIGUSR2)
      printf("received SIGUSR2\n");
   else
      printf("received not SIGUSR1 or SIGUSR2\n");
}
```

The signal() System Call

```c
/*signal.c send signal to a process */
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    int pid =11294;
    int i;

    for (i =1; i<=20 i++)
    {
        if ( i % 2 == 1)
            kill (pid, SIGUSR1);
        else
            kill(pid, SIGUSR2);

        sleep(1);

    }
    return 0;
}
```


Signal Set

- POSIX define the data type **sigset_t** to contain a signal set.
- There are five functions to manipulate signal set.

```c
#include <signal.h>
int sigemptyset(sigset_t *set); /*set all signals are excluded */
int sigfillset(sigset_t *set); /*set all signals  are included */
int sigaddset (sigset_t *set, int signo) /* set a signal set */
int sigdelset (sigset_t *set, int signo) /* reset a signal */
int sigismember (const sigset_t *set, int signo) /*check membership */
```


Signal Set

- Either **sigemptyset**() or **sigfillset**() must be called for every object of type *sigset_t* before any other use of the object.
- The **sigemptyset**() function initializes a signal set to be empty.
- The **sigfillset**() function initializes a signal set to contain all sig-nals.
- The **sigaddset**() function adds the specified signal *signo* to the signal set.
- The **sigdelset**() function deletes the specified signal *signo* from the sig- nal set.
- The **sigismember**() function returns whether a specified signal *signo* is contained in the signal set.

The sigprocmask() System Call

- **Signal mask** of a process is the set of signals currently blocked from delivery to that process.
- The **sigprocmask()** system call examines, or changes, or both examines and changes the signal mask of the calling process.

```c
#include <signal.h>
 int sigprocmask( int how, const sigset_t *set, sigset_t *oset );

return 0 for ok, -1 for error
```


The sigprocmask() System Call

```c
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oset );

return 0 for ok, -1 for error
```

- ***how*** (Input) The way in which the signal set is changed.
- The possible value of how are
  - SIG_BLOCK:Indicates that the set of signals given by *set* should be blocked
  - SIG_UNBLOCK:Indicates that the set of signals given by *set* should not be blocked
  - SIG_SETMASK:the set of signals given by *set* should replace the old set of signals being blocked
- ****set*** (Input) A pointer to a set of signals to be used to change the currently blocked set. May be NULL.
- ****oset*** (Output) A pointer to the space where the previous signal mask is stored. May be NULL.

The sigaction() System Call

- We can modify or examine the action associated with a particular signal by using system call sigaction().

```
#include <signal.h>
int sigaction( int signo, const struct sigaction *act,
struct sigaction *oact);
```

```
struct  segaction{
    void (*sa_handler)(int)/*pointer to signal handler */
    sigset_t sa_mask; /*additional set of signals to block */
    int sa_flag; /*signal options */
    void (*sa_sigaction)(int siginfo_t *, void *);/* alternate handler */
}
```


The sigaction() System Call

```
#include <signal.h>
int sigaction( int signo, const struct sigaction *act,
struct sigaction *oact);
```

- ***signo*** (Input) A signal from the list
- ****act*** (Input) A pointer to the sigaction structure that describes the action to be taken for the signal. Can be NULL. If *act* is a NULL pointer, signal handling is unchanged. **sigaction()** can be used to inquire about the current handling of signal *sig*. If *act* is not NULL, the action specified in the sigaction structure becomes the new action associated with *sig*.
- ****oact*** (Output) A pointer to a storage location where **sigaction()** can store a sigaction structure. This structure contains the action currently associated with *signo*. Can be NULL. If *oact* is a NULL pointer, **sigaction()** does not store this information.


```
/** This program (signalset.c ) blocks SIGINT signal for 10 seconds using sigprocmask(2) After that the signal is
unblocked and  signal is handled. */
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void hdl (int sig)
{
        printf("Got signal Cnt-C\n");;
}

int main (int argc, char *argv[])
{
        sigset_t mask;
        sigset_t orig_mask;
        struct sigaction act;
        memset (&act, 0, sizeof(act)); //clear structure
        act.sa_handler = hdl; // set signal handler as hdl
        // set signal handler for SIGINT
        if (sigaction(SIGINT, &act, 0)) {
                perror ("sigaction");
                return 1;
        }
        sigemptyset (&mask); // clear signal set
        sigaddset (&mask, SIGINT); // add SIGINT
        // SIGINT is blocked
        if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
                perror ("sigprocmask");
                return 1;
        }
        sleep (10);
        // SIGINT is unblocked
        if (sigprocmask(SIG_SETMASK, &orig_mask, NULL) < 0) {
                perror ("sigprocmask");
                return 1;
        }
        sleep (10);

        return 0;
}
```

```
/** This program (signalset1.c) blocks SIGINT signal for 10 seconds using sigprocmask(2)
 * After that the signal is unblocked and  signal is handled.
 */
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
void hdl (int sig)
{
        printf("Got signal Cnt-C\n");;
}
int main (int argc, char *argv[])
{
        sigset_t mask;
        sigset_t orig_mask;
        struct sigaction act;
        memset (&act, 0, sizeof(act)); //clear structure
        act.sa_handler = hdl; // set signal handler as hdl
        // set signal handler for SIGINT
        if (sigaction(SIGINT, &act, 0)) {
                perror ("sigaction");
                return 1;
        }
        sigemptyset (&mask); // clear signal set
        sigaddset (&mask, SIGINT); // add SIGINT
        // SIGINT is blocked
        if (sigprocmask(SIG_BLOCK, &mask, &orig_mask) < 0) {
                perror ("sigprocmask");
                return 1;
        }
        int count =0;
        while (1)
        {
                if (count == 10)
                {
                        printf("now unblocked\n");
                        // SIGINT is unblocked
                        if (sigprocmask(SIG_UNBLOCK, &mask, NULL) < 0) {
                                perror ("sigprocmask");
                                return 1;
                        }
                }
                sleep(1);
                printf("running in cyle %d\n", ++count);
        }
        return 0;
}
```

```
/* sigset1.c: demonstrate block two signal with signal set */
/* regarding set of signal */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>


int main( int argc, char *argv[] )
{

    sigset_t new_set, old_set; // signal set
    time_t start, finish;

    sigemptyset( &new_set );//clear signal sets : block all signal
    sigaddset( &new_set, SIGTSTP );// add signal for Cnt-z
    sigaddset(&new_set, SIGINT); // add signal for Cnt-C

    sigprocmask( SIG_BLOCK, &new_set, &old_set); /*two signals are blocked */
    time( &start );
    printf( "SIGTSTP and SIGINT are blocked at %s\n", ctime(&start));

    sleep (10);
    time( &finish );
    printf( "SIGTSTP and SIGINT are unblocked at %s\n", ctime(&finish));
    sigprocmask( SIG_SETMASK, &old_set, NULL ); // unblock two signals

    while (1)
    {
        ;
     }
    return( 0 );
}
```

```
/* sigset.c: demonstrate several system calls */
/* regarding set of signal */
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

void catcher( int sig ) {
    printf( "The SIGALRM signal is handled well\n" );
}

int main( int argc, char *argv[] )
{
    time_t start, finish;
    struct sigaction aact; /*for sigaction function */
    sigset_t new_set, old_set; /* signal set */
    double diff;

    sigemptyset( &sact.sa_mask ); /* clear signaction structure mask */
    sact.sa_flags = 0; /* flag is set 0 to ignore option */
    sact.sa_handler = catcher; /* set signal action hander as function catcher */
    sigaction( SIGALRM, &sact, NULL ); /*change the action change for signal SIGALRM */
    sigemptyset( &new_set );/*clear signal sets : block all signal*/
    sigaddset( &new_set, SIGALRM );/*make unblock for a SIGALRM */
    sigprocmask( SIG_BLOCK, &new_set, &old_set); /*signal SIGALRM is blocked */
    time( &start );
    printf( "SIGALRM signals blocked at %s\n", ctime(&start) );

    alarm( 1 );     /* SIGALRM will be sent in 1 second */
    printf( "SIGALRM is fired but it is blocked.\n");
    do {
        time( &finish );
        printf( "Current time is %s\n", ctime(&finish) );
        diff = difftime( finish, start );
    } while (diff < 10);

    sigprocmask( SIG_SETMASK, &old_set, NULL ); /* signal set are reset with old set */
    printf( "SIGALRM signals unblocked at %s\n", ctime(&finish) );

    return( 0 );
}
```

Interprocess Communication

- Three issues in interprocess communication
  1. How one process can pass information to another
  2. How to make sure two or more processes do not get into the **critical section** (mutual exclusion)
  3. Proper sequencing when dependencies are present (ex. Producer-Consumer problem, Dinning Philosopher problem)

Interprocess Communication
(the Producer-Consumer Problem)

Description

- Two processes (or threads) share a common, fixed-sized buffer.
- Producer puts information into the buffer, and consumer takes it out.

Troubles arises

- When the producer wants to put a new item in the buffer, but it is already full.
- When the consumer tries to take a item from the buffer, but buffer is already empty.

Interprocess Communication
(the Producer-Consumer Problem)

- When the producer wants to put a new item in the buffer, but it is already full.
  - Solution – producer is go to sleep, awakened by consumer when consumer has removed on or more items.
- When the consumer tries to take a item from the buffer, but buffer is already empty.
  - Solution – consumer is go to sleep, awakened by the producer when producer puts one or more information into the buffer.

Interprocess Communication
(Dining Philosophers Problem)

Interprocess Communication
(Dining Philosophers Problem)

- Five silent philosophers sit at a round table with bowls of spaghetti. Chopsticks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right chopsticks.
- Each chopstick can be held by only one philosopher and so a philosopher can use the chopstick only if it is not being used by another philosopher.
- After an individual philosopher finishes eating, they need to put down both chopsticks so that the chopstick s become available to others. A philosopher can take the chopstick on their right or the one on their left as they become available, but cannot start eating before getting both chopsticks.

Interprocess Communication

- **Critical section** (critical region) – The part of program where the shared memory is accessed.
- If we could arrange matters such that no two processes were ever in their critical sections at the same time, we can avoid races condition.

Interprocess Communication

- We can change the signal mask for a process to block and unblock selected signal by sequence of system calls.
- It might be possible to use this technique to protect critical region (critical section).

Interprocess Communication

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
/*block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask)<0)
   error_sys(" SIG_BLOCK ERROR ");
/**** Critical Region of code ********/
/*reset signal mask, which unblocks SIGINT */
if (sigprocmask(SIG_SETMASK,  &oldmask, NULL)<0)
   error_sys(" SIG_BLOCK ERROR ");
/* hole */
pause();
```


Interprocess Communication

- If a signal is sent to the process while it is blocked, the signal delivery will be differed until the signal is unblocked.
- If a signal does occur between the unblocking and the pause, the signal can be lost.
- The result is pause forever!!!
- The sigsuspend() system guarantee both reset and put a process to sleep.

The sigsuspend() System Call

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

- The **sigsuspend()** function replaces the current signal mask of a process with the signal set given by **sigmask* and then suspends processing of the calling process.
- The process does not resume running until a signal is delivered

The sigsuspend() System Call

```
sigset_t newmask, oldmask;
sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);
/*block SIGINT and save current signal mask */
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask)<0)
   error_sys("SIG_BLOCK ERROR");

      /**** Critical Region of code ********/

/*reset signal mask, which unblocks SIGINT */
if (sigsuspend(&oldmask)<0)
   error_sys("SIG_SUSPEND ERROR");
```



```
/*sigsuspend.c */

#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
void catcher( int sig ) {
    printf( "inside catcher() function\n" );
}

void timestamp( char *str ) {

    time_t t;

    time( &t );
    printf( "%s the time is %s\n", str, ctime(&t) );
}

int main( int argc, char *argv[] )
{
    struct sigaction sigact; /* for sigaction call */
    sigset_t block_set;

    sigfillset( &block_set ); /*set all signals  are included */
    sigdelset( &block_set, SIGALRM ); /* exclude SIGALRM  from  signal set*/
    sigact.sa_handler = catcher;       /* set signal hadler*/
    sigaction( SIGALRM, &sigact, NULL ); /* set sigaction for SIGALRM signal */
    timestamp( "before sigsuspend()" );
    alarm( 5 );
    sigsuspend( &block_set ); /* replaces the current signal mask  with block_set */
    timestamp( "after sigsuspend()" );

    return( 0 );
}
```

The abort() System Call

- The abort() system call cause abnormal program termination.
- The abort() system call send SIGABRT signal to caller process

```
#include <stdlib.h>
void abort(void);
```