

## Preview

- Inter-process Communication
- The sigsuspend() System Call
- The abort() System Call
- Concept of Thread.
- Other View of Thread
- Summary of Thread
- Benefits with Multiple Threads
- What are Pthreads?
- The Thread ID
- The Thread Creation and Termination

## 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 region (critical section)-Mutual Exclusion**
  3. Proper sequencing when dependencies are present (ex. Producer-Consumer: a producer produce outputs and save into a buffer, a consumer consume outputs from the buffer)

## 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 regions at the same time (mutual exclusion), 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;
/* exclude all signal in signal set*/
sigemptyset (&newmask);
/* set SIGINT */
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_SETMASK ERROR");
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;
/* exclude all signal set */
sigemptyset(&newmask);
/* include SIGINT signal */
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 handler*/
    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

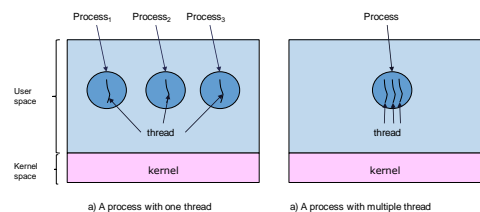
- The abort() system call cause abnormal program termination.
- The abort() system call send SIGABRT signal to caller process

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

## Concept of Threads

- In a traditional OS, each process has an address space and a single thread of control.
- But in modern software, there are multiple threads of control in the same addresses space.
- Each threads inside a process need its independent spaces for running **but sharing the same address space of program.**

## Concepts of Threads



## Concept of Threads

### Threads

- ❑ Threads are processes in a process!!! – **multiple execution in the same process environment.**
- ❑ It is made up with a **thread ID**, a **program counter**, a **register set**, and a **stack**.
- ❑ Different threads are not quite as independent as different processes since they share same address space
- ❑ It shares with other threads belonging to the same process its **code section**, **data section** and other operating system resources such as **files** and **signals**.

## Concept of Threads

- ❑ **Multiple process running on a computer** – Process are share physical memory, disks, printers and other resources
- ❑ **Multiple threads running on a process** – the threads are share an address space, open files, and other resources
- ❑ With thread – the ability for multiple threads of execution to share a set of resources so they can work together closely to perform some task.

## Concepts of Threads

- ❑ The CPU switches rapidly back and forth among the threads in the single CPU system (Same idea as multiprogramming)
- ❑ No protection between threads – using same address space (share the global variables)
- ❑ A thread can be in any one of several states: running, blocked, ready

## Concepts of Threads

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

## Concept of Threads

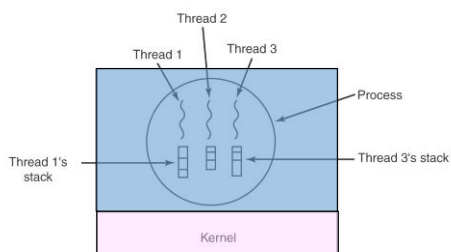
- ❑ Since each thread usually call different functions from the same address space (each thread executing different part of program), each thread need its own stack.
- ❑ A multithread software start with single thread.
- ❑ The thread has ability to create new threads by calling a library procedure (i.e. thread\_create)
- ❑ When a thread has finish its work, it can exit by calling a library procedure ( i.e. thread\_exit)

## Concept of Threads

- ❑ pthread functions for POSIX

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

## Concepts of Threads



COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

19

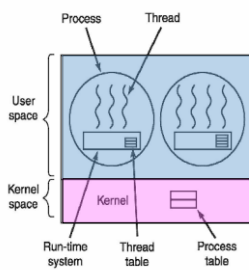
## Threads Implementation

- Implementing thread in user space
  - Threads are handled by the run-time system
  - OS does not know the existence of threads
- Implementing thread in the kernel
  - Thread are handled by OS

COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

20

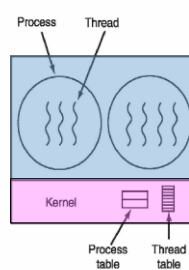
## Implementing Threads in User's space



COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

21

## Implementing Threads in Kernel's Space



COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

22

## Other View of Thread

1. A thread is defined as **an independent stream of instructions** that can be scheduled to run by the operating system.
2. **A function that runs independently from its main program:** can be scheduled by Operating System.

COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

23

## Motivation of Threads

Ex) A web browser

1. A thread for displaying images or text
2. A thread for retrieving data from network

Ex) Word processor

1. A thread for displaying graphics
2. A thread for reading input from keyboard
3. A thread for performing spelling and grammar checking in the background

COSC350 System Software, Fall 2024  
Dr. Sang-Eon Park

24

## Summary of Thread

- Exists within a process and uses the process resources
- Has its own independent flow of control as long as its parent process exists and the OS supports it
- Duplicates only the essential resources it needs to be independently schedulable
- Share the process resources with other threads that act equally independently
- Dies if the parent process dies
- Is "lightweight" because most of the overhead has already been accomplished through the creation of its process

## Benefits with Multiple Threads.

- With multiple thread, we can design a program to do more than one thing at a time within a single process, with each thread handling a separate task.
- We can simplify code that deal with asynchronous events by assigning a separate thread to handle each event type.
- In multi processor system, it is easy to apply parallel processing with multiple threads. Each thread can working with a processor.

## What are Pthreads?

- Hardware vendors had implemented their own proprietary versions of threads.
- For a requirement of standardized programming interface, this interface has been specified by the IEEE POSIX 1003.1c standard (1995) (POSIX thread).
- pthreads are defined as a set of C language programming types and procedure calls, implemented with a header file <pthread.h>

## The Thread ID

- A Process ID (**pid\_t**) for a process is unique in the system.
- But a thread ID (**pthread\_t**) has significance only within the context of the process where it belongs.
- Even though unsigned long (Linux), unsigned integer (Solaris 9) is used represent a **pthread\_t**, a function **pthread\_equal()** must be used to compare thread ID.

## The Thread ID

```
#include <pthread.h>
/*returns non-zero if equal, return 0 otherwise */
int pthread_equal(pthread_t t1, pthread_t t2);
```

```
#include <pthread.h>
/*returns thread ID of the calling thread */
pthread_t pthread_self(void);
```

## The thread Creation

- Initially, your main() program comprises a single, default thread.
- All other threads must be explicitly created by the programmer
- **pthread\_create()** function creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.
- Once created, threads are peers, and may create other threads. **There is no implied hierarchy or dependency between threads**

## The thread Creation

```
#include <pthread.h>
int pthread_create (pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine)(void*), void *arg);
```

- att point to structure of pthread attribute. If att is NULL, a default attribute will be used
- start\_routing point to address of a void function with no parameter or
- We can save parameters to typeless pointer arg and be able to pass to the function.
- If successful, return 0; otherwise return an error number
  - [EAGAIN]: The system lacked the necessary resources to create another thread,
  - [EPERM]: The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy,
  - [EINVAL]The attributes specified by attr are invalid

## The thread Creation

```
struct pthread_attr_t{
{
    int        flags
    int        stacksize
    int        contentionscope
    int        inheritsched
    int        detachstate
    int        sched
    struct sched_param    param
    struct timespec starttime deadline    period
};
```

## The thread Creation

- When multiple threads are created, there is **no guarantee which runs first**. It is depends on the thread scheduler.

```
/******
 * CreateTh.c: Demonstrate creation of threads
 * *****/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{
    int tid;
    tid = (int)threadid;
    printf("Hello World! It's me, thread %d!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, t;
    for(t=0;t<NUM_THREADS;t++)
    {
        printf("In main: creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        if (rc)
        {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
    exit (0);
}
```

## The thread Termination

- If any thread within a process call exit or \_exit system call, then the entire process terminate.
- A single thread inside a process can terminate three ways.
  - The thread can simply return from the start routine
  - The thread can be cancelled by another thread with **pthread\_cancel()** in the same process.
  - The thread can call pthread\_exit

## The thread Termination

```
#include <pthread>
void pthread_exit(void *rval_ptr);
```

- The rval\_ptr: typeless pointer. This pointer is available to other threads in the process by calling the pthread\_join() system call.