System Programming
(ELEC462)

Threads

Dukyun Nam

HPC Lab@KNU

Contents

Introduction

- Ideas and skills
  - Threads of execution
  - Multithreaded programs
  - Creating and destroying threads
  - Passing multiple arguments to a thread
  - Sharing data between threads safely using “mutex locks”
  - Synchronizing data transfer using condition variables among threads
- System calls and functions
  - `pthread_create`, `pthread_join`
  - `pthread_mutex_lock`, `pthread_mutex_unlock`
  - `pthread_cond_wait`, `pthread_cond_signal`

Threads of Execution

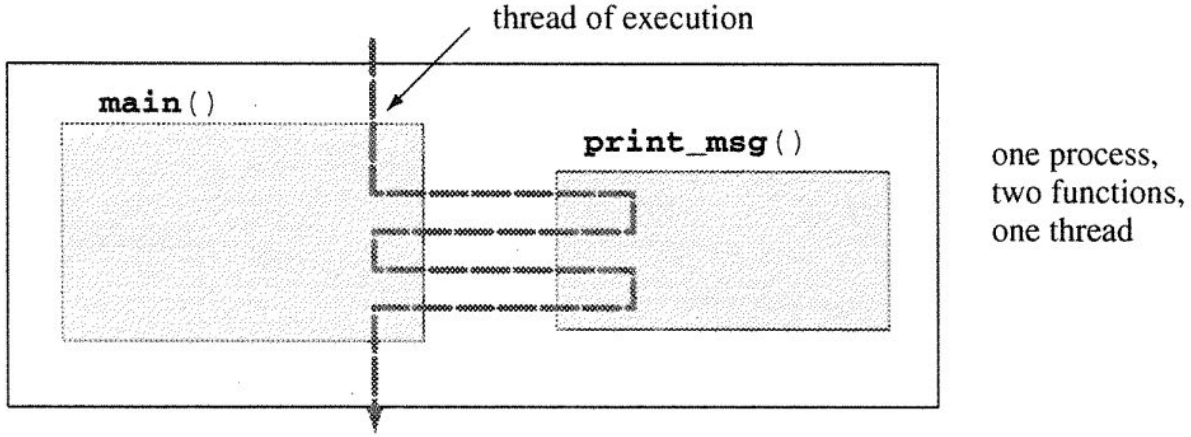
- A toy C program: `hello_single.c`
  - Write `print_msg()` to print a given string (e.g., `m`) and then sleep for 1 sec as many times as specified by `NUM`
  - This program prints "`Hello`" and then "`world\n`" and sleeps for 1 second, which repeats five times

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./hello_single
HelloHelloHelloHelloHelloworld
world
world
world
world
```

```c
#include  <stdio.h>
#include  <unistd.h>
#define NUM     5

int main()
{
        void    print_msg(char *);

        print_msg("Hello");
        print_msg("world\n");

        return 0;
}

void print_msg(char *m)
{
        int i;
        for(i=0 ; i<NUM ; i++){
                printf("%s", m);
                fflush(stdout);
                sleep(1);
        }
}
```

Threads of Execution (cont.)

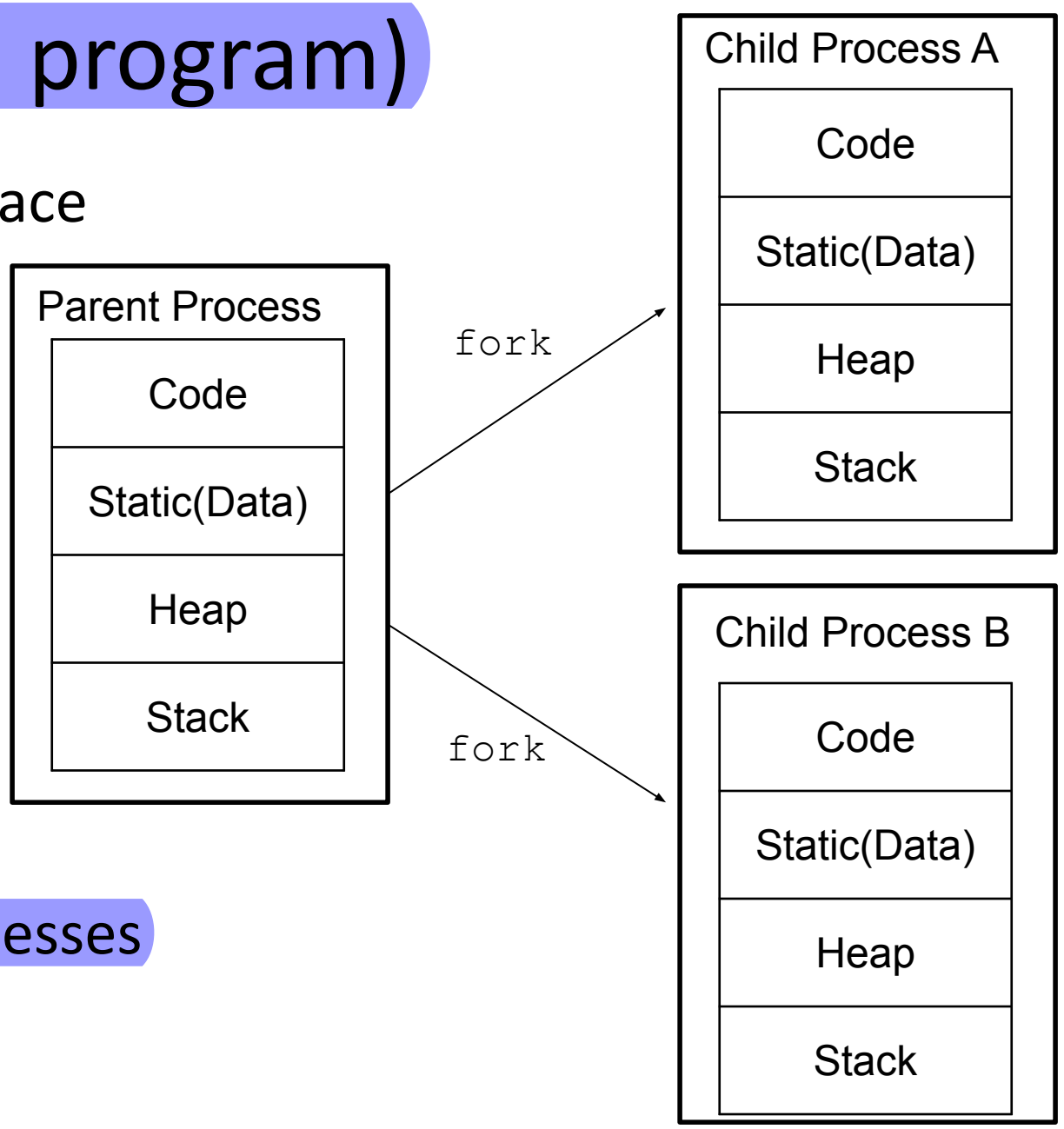
- An Execution of the Program: a Process
  - Thread of execution
    - "an unbroken path" tracing the order of execution of instructions
    - Control starts in main, then flows into repeat two times, then exits from the end of main



< Single thread of execution >

Process (a single-thread program)

- Gets a certain memory address space
  - Each (same) running process has its own memory space
- The parent and the two children have their own memory realms
- Environment variables can be inherited from the parent
- "`ps`" will tell you those three processes running independently

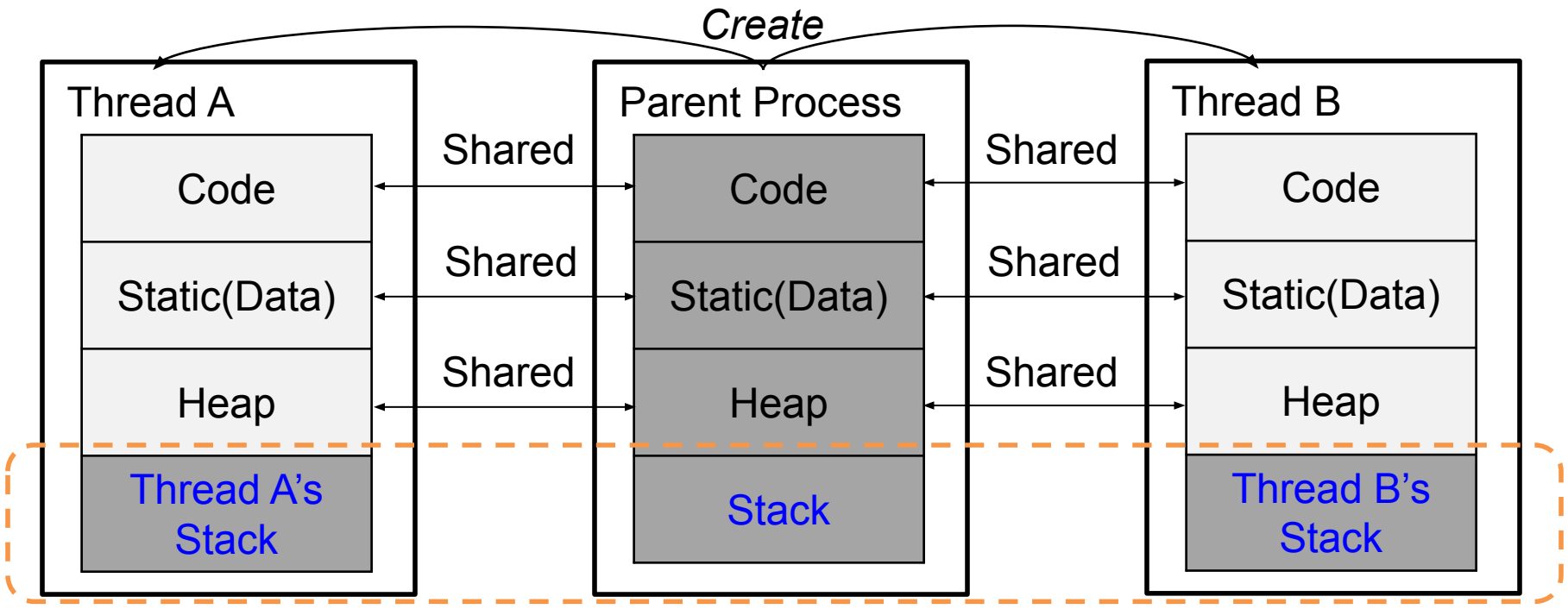


Thread(s)

- Definition: an (multiple) execution(s) within a *single* process
  - Functions what a process is to programs, an environment in which to run.
  - Sometimes referred as a *lightweight* process (LWP)
  - The “smallest unit” of processing that a kernel works on
- Shares the same memory address space as its parent (process)
  - Shares all global variables and file descriptors of the parent process
- Allows the programmer to separate multiple tasks easily in a single process

Thread(s) (cont.)

- Parent and two children

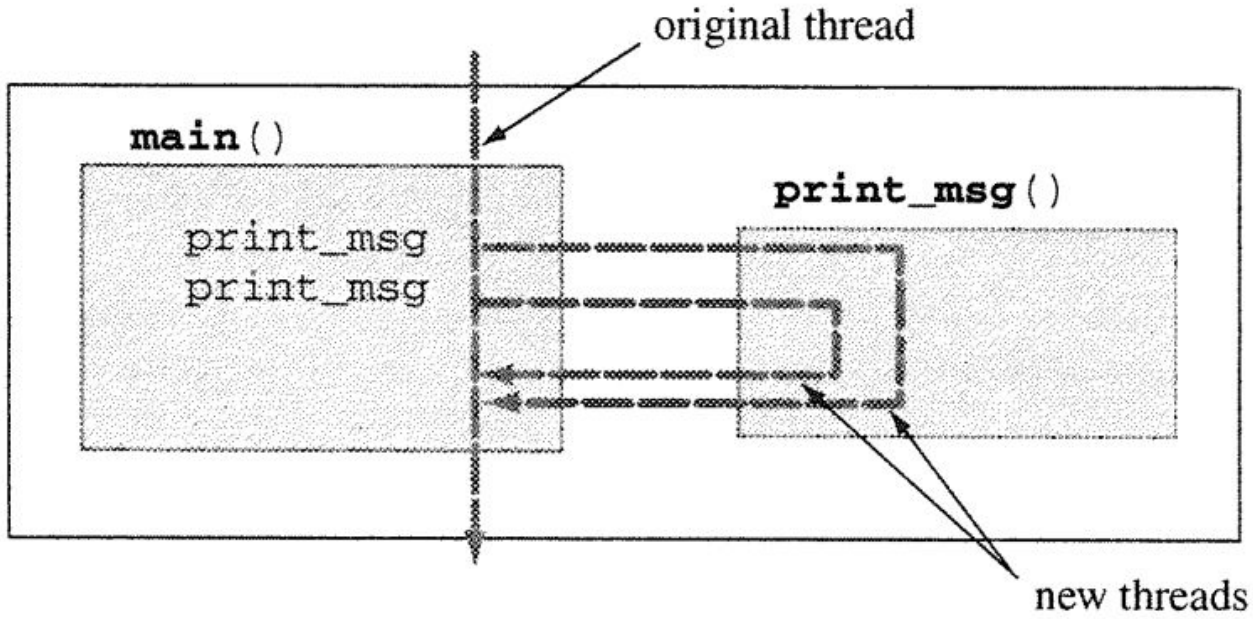


Why Thread(s)?

- Requires “fewer system resources” compared to forking
  - No extra memory space but stack
- Consumes less process time to create and to terminate than a process (compared to `fork`)
- Enables “resource” sharing
  - Some memory realms shared: code, data, heap, static
- Less communication overhead
  - Neither pipe nor socket needed!
- There are also some shortcomings …
  - We’ll get to discuss similarities and differences between processes and threads in later slides

A Multithreaded Program

- If we want to run `print_msg` concurrently, how?
  - A process may create a new thread by specifying a function to run and an argument



< Multiple threads of execution >

```c
int main()
{
        pthread_t t1, t2;                /* two threads */

        void    *print_msg(void *);

        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world\n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        return 0;
}
```

Relevant Function 1: `pthread_create`

pthread_create		
PURPOSE	Create a new thread	
INCLUDE	#include <pthread.h>	
USAGE	int pthread_create(pthread_t	*thread,
	pthread_attr_t	*attr,
	void	*(*func)(void *),
	void	*arg);
ARGS	thread	a pointer to a variable of type pthread_t
	attr	a pointer to a variable of type pthead_attr_t or NULL.
	func	the function this new thread will run
	arg	the argument to be passed to func
RETURNS	0	if successful
	errcode	if not successful

```c
int main()
{
        pthread_t t1, t2;                /* two threads */

        void    *print_msg(void *);

        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world\n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        return 0;
}
```

Relevant Function 2: `pthread_join`

pthread_join		
PURPOSE	Wait for termination of a thread	
INCLUDE	#include <pthread.h>	
USAGE	int pthread_join(pthread_t thread, void **retval)	
ARGS	thread	the thread to wait for
	retval	points to a variable to receive the return value from the thread
RETURNS	0	if thread terminates
	errcode	if an error

```c
int main()
{
        pthread_t t1, t2;                /* two threads */

        void    *print_msg(void *);

        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world\n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        return 0;
}
```

A Multithreaded Program: `hello_multi.c`

```c
/* hello_multi.c - a multi-threaded hello world program */

#include  <stdio.h>
#include  <unistd.h>
#include  <pthread.h>

#define NUM     5

int main()
{
        pthread_t t1, t2;               /* two threads */

        void    *print_msg(void *);

        pthread_create(&t1, NULL, print_msg, (void *)"hello");
        pthread_create(&t2, NULL, print_msg, (void *)"world\n");
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);

        return 0;
}
```

```c
void *print_msg(void *m)
{
        char *cp = (char *) m;
        int i;
        for(i=0 ; i<NUM ; i++){
                printf("%s", cp);
                fflush(stdout);
                sleep(1);
        }
        return NULL;
}
```

A Multithreaded Program: `hello_multi.c` (cont.)

- Execution
  - `-lpthread`: tells `gcc` to (dynamically) link the pthread library to its source code (`-l`*`libname`*: link a library named *libname*)

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ make
cc -o hello_multi hello_multi.c -lpthread
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./hello_multi
helloworld
helloworld
world
helloworld
helloworld
hellodynam@DESKTOP-Q4IJBP7:~/lab13$
```

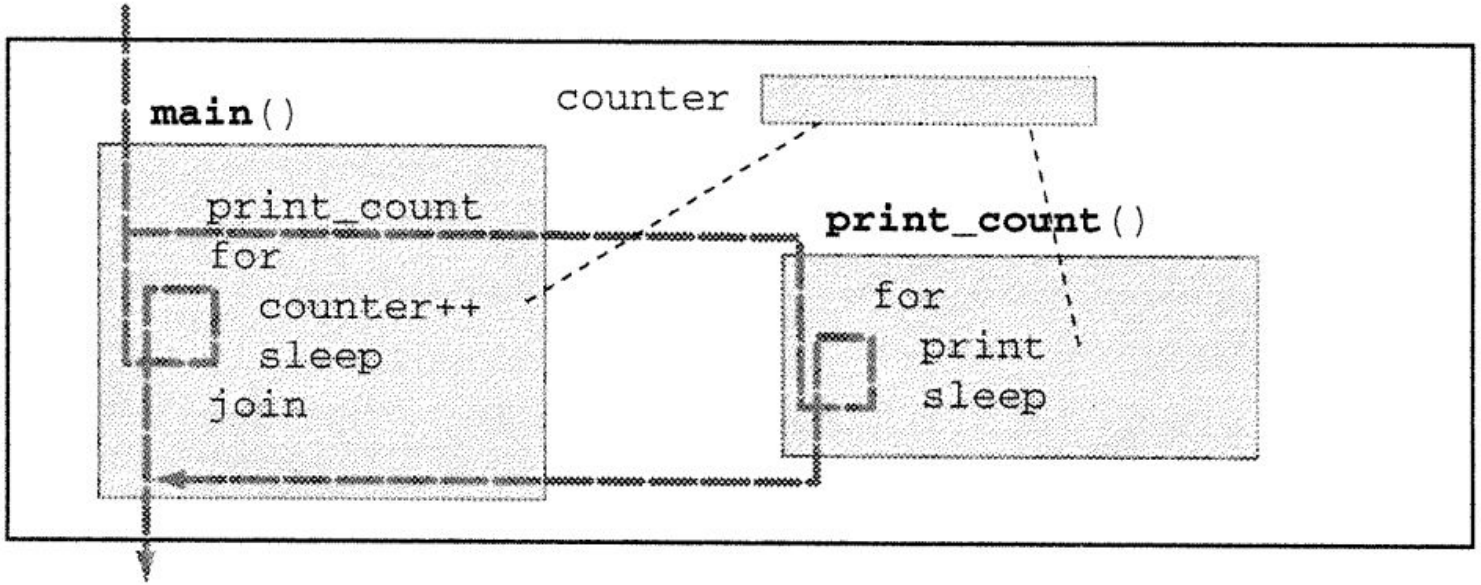
Interthread Communication

- Threads execute *functions* in a single process
- They share "**global variables**," or their communication channel
  - c.f., Processes communicate with each other using pipes, sockets, signals, `exit`/`wait`, and the environment
- Simultaneous access to memory: a powerful but risky feature

- Two big questions for threads and data
  - 1. How can threads share variables safely?
  - 2. How can threads transfer data effectively?

Example 1: `incrprint.c`

- All threads in a process share the same set of functions and global variables
  - Two “executions” (main process and one thread) share a global variable



< Two threads share a global variable >

Example 1: `incrprint.c` (cont.)

- In this program, one thread increments a variable, and the other thread prints the value

```c
/* incprint.c - one thread increments, the other prints */

#include  <stdio.h>
#include  <unistd.h>
#include  <pthread.h>

#define NUM     5

int     counter = 0;

int main()
{
        pthread_t t1;                   /* one thread */
        void      *print_count(void *); /* its function */
        int       i;

        pthread_create(&t1, NULL, print_count, NULL);
        for( i = 0 ; i<NUM ; i++ ){
                counter++;
                sleep(1);
        }

        pthread_join(t1, NULL);

        return 0;
}
```

```c
void *print_count(void *m)
{
        int i;
        for(i=0 ; i<NUM ; i++){
                printf("count = %d\n", counter);
                sleep(1);
        }
        return NULL;
}
```

Example 1: `incrprint.c` (cont.)

- When `main` changes the value of counter, `print_counter` sees that new value at once
  - There is no need to send the new value through a pipe or socket

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ make
cc -o incprint incprint.c -lpthread
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./incprint
count = 1
count = 2
count = 3
count = 4
count = 5
```

- A *single-threaded* program works! – Not practical though…
  - Many concurrent programs runs multiple threads in parallel!

A Target Program for Multi-Threading: `wc`

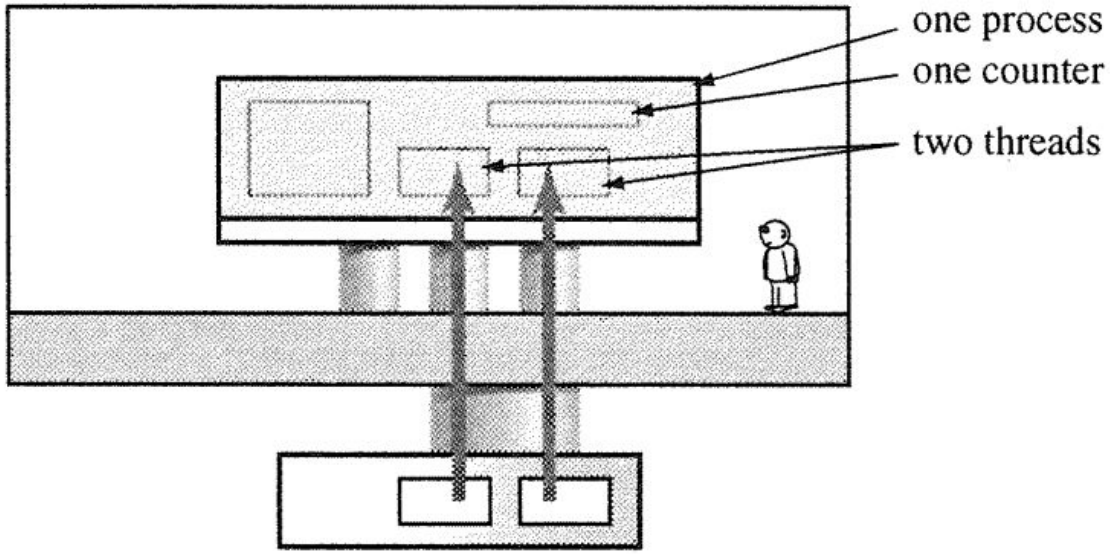
- Counts lines, words, and chars in one or more files in a *single-threaded*

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ wc incprint.c
 36  82 545 incprint.c
dynam@DESKTOP-Q4IJBP7:~/lab13$ wc -l incprint.c
36 incprint.c
dynam@DESKTOP-Q4IJBP7:~/lab13$ wc -w incprint.c
82 incprint.c
dynam@DESKTOP-Q4IJBP7:~/lab13$ wc -c incprint.c
545 incprint.c
```

- Now let's make the `wc` program be *threaded*

Example 2-1: `twordcount1.c`

- Version 1: "Two Threads with **One Counter**"
  - This real "**multi-threaded**" program counts the number of words in two files and prints the number of words
  - We create a separate thread to count each file
  - Both threads increment the counter while detecting words



< A common counter for two threads >

Example 2-1: `twordcount1.c` (cont.)

```c
/* twordcount1.c - threaded word counter for two files. Version 1 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <pthread.h>
#include  <ctype.h>

int       total_words ;

int main(int ac, char *av[])
{
        pthread_t t1, t2;               /* two threads */
        void      *count_words(void *);

        if ( ac != 3 ){
                printf("usage: %s file1 file2\n", av[0]);
                exit(1);
        }
        total_words = 0;
        pthread_create(&t1, NULL, count_words, (void *) av[1]);
        pthread_create(&t2, NULL, count_words, (void *) av[2]);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        printf("%5d: total words\n", total_words);

        return 0;
}
```

```c
void *count_words(void *f)
{
        char *filename = (char *) f;
        FILE *fp;
        int  c, prevc = '\0';

        if ( (fp = fopen(filename, "r")) != NULL ){
                while( ( c = getc(fp)) != EOF ){
                        if ( !isalnum(c) && isalnum(prevc) )
                                total_words++;
                        prevc = c;
                }
                fclose(fp);
        } else
                perror(filename);
        return NULL;
}
```

Example 2-1: `twordcount1.c` (cont.)

- Execution

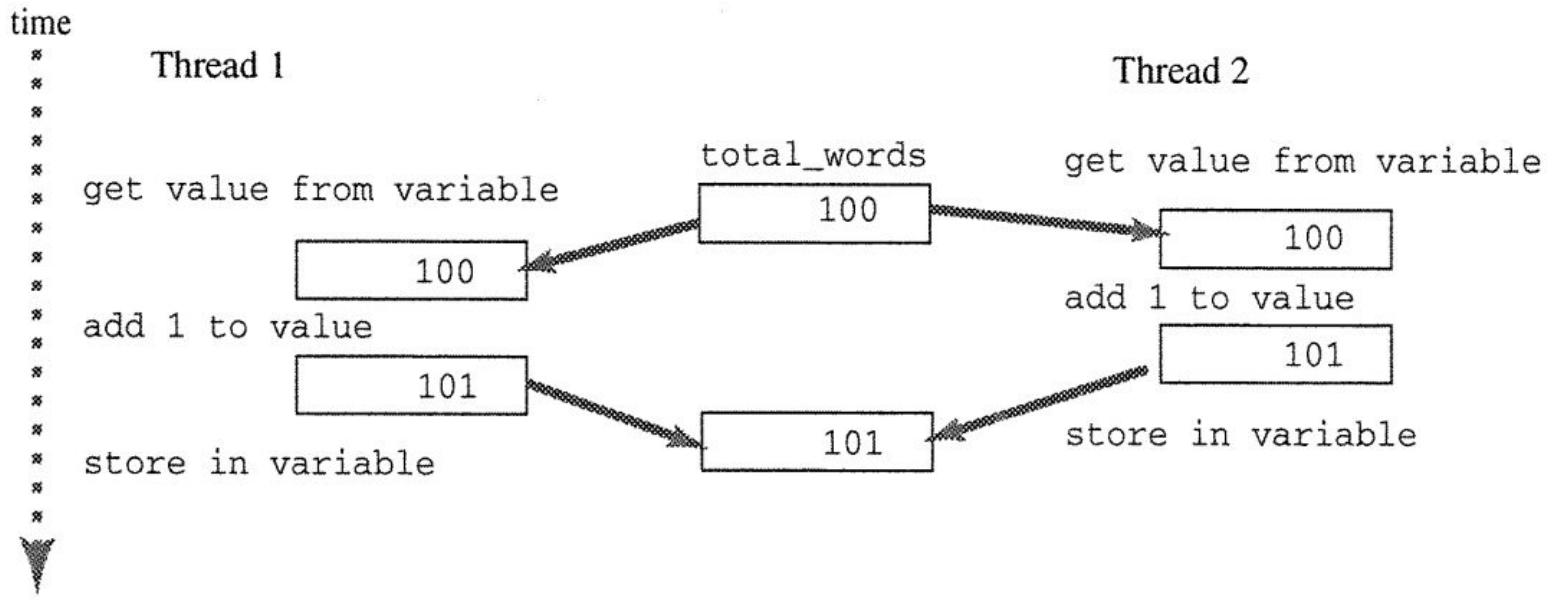
```
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
132005: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /dev/null
  249: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /usr/share/dict/words /dev/null
131756: total words
```

- What happens if both threads read then update a variable at the same time?
  - Answer) Unpredictable confusion

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131977: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131969: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131955: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131966: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
132005: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131966: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc1 /etc/group /usr/share/dict/words
131950: total words
```

Example 2-1: `twordcount1.c` (cont.)

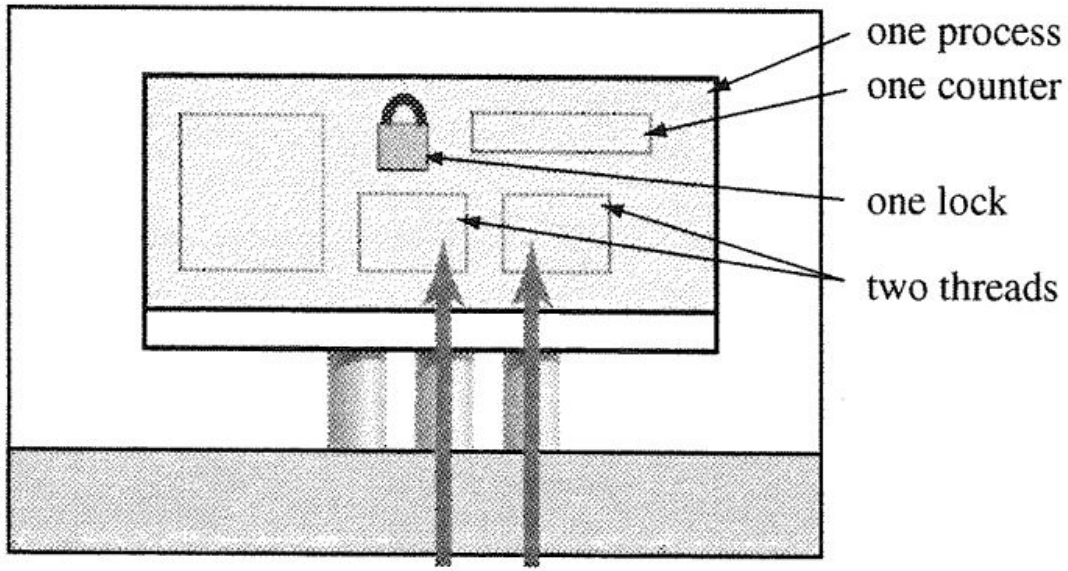
- ***Drawback*** of this program is …
  - Two increments take place although we have only "one counter" …
    - What happens if both threads increment the counter using **fetch-add-store** sequence at the **same** time?
    - In this case, both threads were not well synchronized on the same resource



< Two threads increment the same counter >

Example 2-2: `twordcount2.c`

- Version 2: "Two Threads, One Counter, **One Mutex**"
  - A mutex is a special kind of object that can be 'locked' and 'unlocked'
  - Only one thread may lock a mutex at a time
  - Attempts to lock a mutex block until the mutex is unlocked



< Two threads use a mutex to share a counter >

Example 2-2: `twordcount2.c` (cont.)

- The threads system includes variables, called *mutual exclusion locks* (abbr. ***mutex***)
  - Used by cooperating threads to prevent simultaneous access to variable, function, and resources
  - e.g., threads A and B
    - If thread A gets the lock first, thread B's call to `pthread_mutex_lock` waits until thread A calls `pthread_mutex_unlock`
    - This prevents simultaneous access to the shared counter

```
if ( !isalnum(c) && isalnum(prevc) ){
        pthread_mutex_lock(&counter_lock);
        total_words++;
        pthread_mutex_unlock(&counter_lock);
}
```

Example 2-2: `twordcount2.c` (cont.)

- Function Summaries: `pthread_mutex_lock`

pthread_mutex_lock		
PURPOSE	Wait for and lock a mutex	
INCLUDE	#include <pthread.h>	
USAGE	int pthread_mutex_lock(pthread_mutex_t *mutex)	
ARGS	mutex	a pointer to a mutual exclusion object
RETURNS	0	for success
	errcode	for errors

Example 2-2: `twordcount2.c` (cont.)

- Function Summaries: `pthread_mutex_unlock`

pthread_mutex_unlock		
PURPOSE	Unlock a mutex	
INCLUDE	#include <pthread.h>	
USAGE	int pthread_mutex_unlock(pthread_mutex_t *mutex)	
ARGS	mutex	a pointer to a mutual exclusion object
RETURNS	0	for success
	errcode	for errors

Example 2-2: `twordcount2.c` (cont.)

```c
/* twordcount2.c - threaded word counter for two files.    */
/*                 version 2: uses mutex to lock counter   */

#include  <stdio.h>
#include  <stdlib.h>
#include  <unistd.h>
#include  <pthread.h>
#include  <ctype.h>

int             total_words ;    /* the counter and its lock */
pthread_mutex_t counter_lock = PTHREAD_MUTEX_INITIALIZER;

int main(int ac, char *av[])
{
        pthread_t t1, t2;               /* two threads */
        void      *count_words(void *);

        if ( ac != 3 ){
                printf("usage: %s file1 file2\n", av[0]);
                exit(1);
        }
        total_words = 0;
        pthread_create(&t1, NULL, count_words, (void *) av[1]);
        pthread_create(&t2, NULL, count_words, (void *) av[2]);
        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        printf("%5d: total words\n", total_words);

        return 0;
}
```

```c
void *count_words(void *f)
{
        char *filename = (char *) f;
        FILE *fp;
        int  c, prevc = '\0';

        if ( (fp = fopen(filename, "r")) != NULL ){
                while( ( c = getc(fp)) != EOF ){
                        if ( !isalnum(c) && isalnum(prevc) ){
                                pthread_mutex_lock(&counter_lock);
                                total_words++;
                                pthread_mutex_unlock(&counter_lock);
                        }
                        prevc = c;
                }
                fclose(fp);
        } else
                perror(filename);
        return NULL;
}
```

Example 2-2: `twordcount2.c` (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc2 /etc/group /usr/share/dict/words
132005: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc2 /etc/group /usr/share/dict/words
132005: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc2 /etc/group /usr/share/dict/words
132005: total words
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc2 /etc/group /usr/share/dict/words
132005: total words
```

Is Mutex an *Ultimate* Solution?

- Mutex allows concurrent threads to be synchronized with a “lock”
  - Offers no inconsistent state of a variable
- But … using a mutex makes the program ***slower***
  - *A lot of operations* are added up due to:
    - Checking the lock, setting the lock, and releasing the lock for every word in both files
- What is a “more efficient” solution?

Example 2-3: `twordcount3.c`

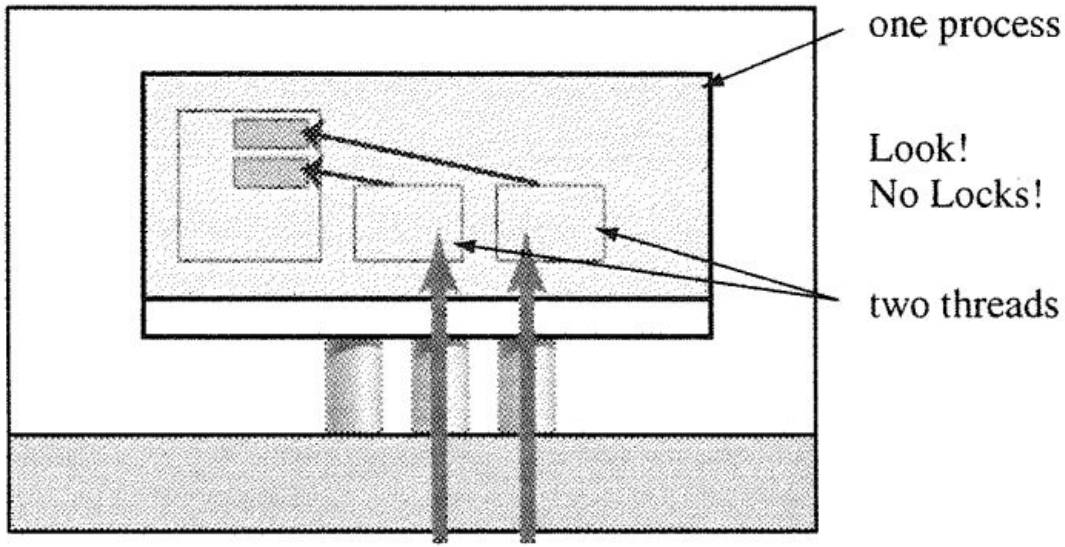
- Version 3: "Two Threads, **Two Counters, Multiple Arguments to Threads**"
  - We'll eliminate the need for a mutex
  - We'll instead give each thread its own counter, which sounds more efficient
  - Questions:
    - How do we get the threads to acquire those counters?
    - How do the threads pass their counts back?

Example 2-3: `twordcount3.c` (cont.)

- Our approach:
  - Makes the calling thread be able to pass to the function a ***pointer*** to a variable
    - Issue: `pthread_create` takes one argument, thereby no more room.
    - Solution: Let's use a (local) **struct** with two members of (i) *filename* and (ii) *number of words* and then have the struct handed over to each thread
  - Let the function ***increment*** that variable

Example 2-3: `twordcount3.c` (cont.)

- Two arguments (defined in a struct) are passed to the threads
  - The struct contains (i) file name and (ii) number of words.
  - Passing pointers to local structs
- This will remove the use of mutex and global variables
- Note
  - The argument to each thread must include a filename and a pointer to a counter



< Each thread has a pointer to its own struct >

Example 2-3: `twordcount3.c` (cont.)

```c
/* twordcount3.c - threaded word counter for two files.
 *               - Version 3: one counter per file
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <pthread.h>
#include  <ctype.h>

struct arg_set {                /* two values in one arg */
                char *fname;    /* file to examine       */
                int  count;     /* number of words       */
};
```

```c
int main(int ac, char *av[])
{
        pthread_t      t1, t2;          /* two threads */
        struct arg_set args1, args2;    /* two argsets */
        void           *count_words(void *);

        if ( ac != 3 ){
                printf("usage: %s file1 file2\n", av[0]);
                exit(1);
        }
        args1.fname = av[1];
        args1.count = 0;
        pthread_create(&t1, NULL, count_words, (void *) &args1);

        args2.fname = av[2];
        args2.count = 0;
        pthread_create(&t2, NULL, count_words, (void *) &args2);

        pthread_join(t1, NULL);
        pthread_join(t2, NULL);
        printf("%5d: %s\n", args1.count, av[1]);
        printf("%5d: %s\n", args2.count, av[2]);
        printf("%5d: total words\n", args1.count+args2.count);

        return 0;
}
```

Example 2-3: `twordcount3.c` (cont.)

```c
void *count_words(void *a)
{
        struct arg_set *args = a;       /* cast arg back to correct type */
        FILE *fp;
        int  c, prevc = '\0';

        if ( (fp = fopen(args->fname, "r")) != NULL ){
                while( ( c = getc(fp)) != EOF ){
                        if ( !isalnum(c) && isalnum(prevc) )
                                args->count++;
                        prevc = c;
                }
                fclose(fp);
        } else
                perror(args->fname);
        return NULL;
}
```

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ make
cc -o twc3 twordcount3.c -lpthread
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc3 /etc/group /usr/share/dict/words
  249: /etc/group
131756: /usr/share/dict/words
132005: total words
```

Interthread Cooperation: Summary

- Three approaches to sharing values between threads
  - 1. Allowing threads to modify the same variable with no cooperation
    - `twordcount1.c`: works most of the time, which is NOT correct.
  - 2. Using a mutual exclusion object, a *mutex*,
    - To allow "only one thread" to increment the shared counter at a time
    - `twordcount2.c`: works most of the time
      - But makes many calls to the functions to check, set, and release the lock
  - 3. Creating separate counters for each thread while eliminating a shared counter
    - `twordcount3.c`: uses `pthread_join` to block until all threads finish
    - The threads no longer share a variable; then no need of cooperation
    - But can still cooperate with the original thread

Interthread Notification

- How can one thread notify another?
  - Consider a simple experiment
    - `twc really-big-file tiny-file`
    - The `twc` program would use `pthread_wait` to wait for the first thread to finish and then for the second to finish. Is this OK?
  - If the second thread finishes its work (e.g., counting) early, how can it notify the first thread that it has the counting results?
  - In processes, the `wait`/`exit` system calls are used to know termination and its status of any children.
  - However, threads don't work that way; they don't have a parent to notify

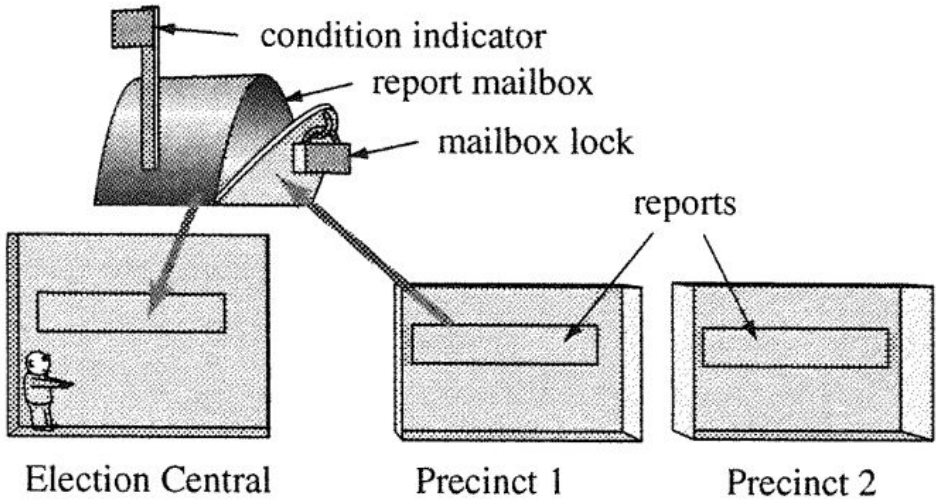
Notifying Election Central

- Analogy: Counting votes
  - Each Precinct delivers a report to the mailbox and notifies EC
  - EC then consumes the report
  - EC waits for the flag to be signaled

    while Precinct (선거구) can

    raise—sends a signal to—the flag
    - * precinct: an electoral district of

      a city or town



< Using a locked mailbox to transfer data >

  - This mailbox can only store one certified vote count at a time
    - The polling places produce vote counts and election central consumes vote counts
    - Only one vote count can be stored at a time

Notifying Election Central (cont.)

- (a) Election Central (EC) sets up a vote report mailbox
  - This mailbox has space for only one vote report at a time
  - This mailbox has a flag that can be raised and then snaps right back
  - This mailbox has a mutex that can be locked and unlocked
- (b) EC unlocks the box and waits until the flag is signaled
- (c) Voting place (VP) waits until it can lock the mailbox
  - If the mailbox is not empty, VP unlocks the mailbox and waits until the flag is signaled before it can lock the mailbox again
  - VP now puts the voting report in the mailbox

Notifying Election Central (cont.)

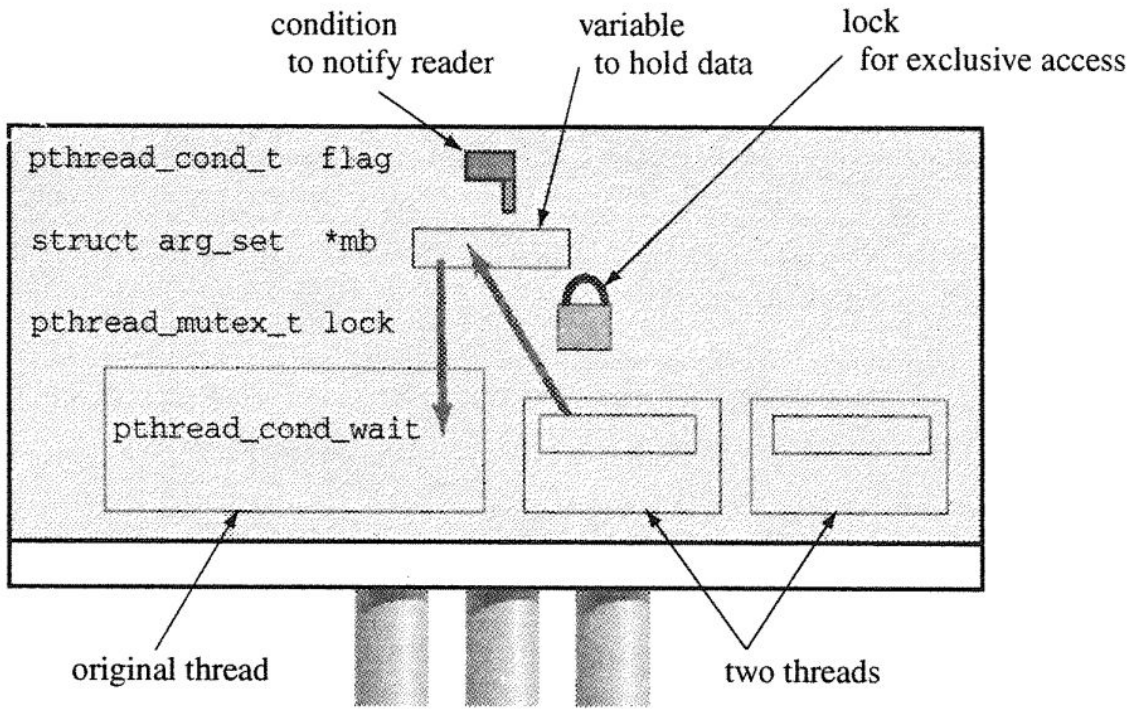
- (d) VP signals the flag on the mailbox
  - VP releases the lock on mailbox
- (e) EC stops waiting because the flag was signaled
  - EC locks the mailbox, takes the voting report from the mailbox, and processes the data on the voting report
  - EC then signals the flag in case a VP is waiting
  - EC returns to Step (b)

Condition Variables: wait and signal

- A condition variable is an object threads use to notify other threads
  - A thread ***waits for notification*** with `pthread_cond_wait()`
    - e.g., wait for signal on condition variable
  - A thread ***sends notification*** with `pthread_cond_signal()`
    - e.g., signal that you are done

Example 2-4: `twordcount4.c`

- Version 4: "Three Threads, Three Variables"
  - Programming with *conditional variables*
  - Let's use a "locked (conditional) variable" to transfer data



< Using a locked variable to transfer data >

Example 2-4: `twordcount4.c` (cont.)

- Logic of the program
  - **The main program** (original thread) launches the "two counting threads" and then waits for results to come in.
    - Calls `pthread_cond_wait` to wait for the flag to be signaled
      - This call blocks the original thread.
  - When a **counting thread** finishes counting, it is ready to deliver the result by storing a "pointer" in the mailbox.
    - 1) That **counting thread** has to acquire a lock for the mailbox.
    - 2) It checks the mailbox once the lock is obtained.
      - If not empty, then the thread unlocks the mailbox and waits for the flag to be signaled before locking the mailbox again.
    - 3) The thread puts the result into the mailbox.
    - 4) The thread signals the condition variable flag by calling `pthread_cond_signal`

Example 2-4: `twordcount4.c` (cont.)

- Signaling this flag wakes up the original thread (or, the main program)
  - Note that it was blocked on that flag by calling `pthread_cond_wait`
  - The (awakened) original thread rushes to open the mailbox and tries to obtain a lock for that mailbox, but the lock is still held by the counting thread.
- When the **counting thread** releases the lock with `pthread_mutex_unlock`, the original thread gets the lock.
  - After holding the lock, the original thread now:
    - Takes the report out of the mailbox
    - Reports the result to the screen and adds the number to its total
    - Signals the flag in case a counting thread is waiting, and
    - Loops back to call `pthread_cond_wait`, which atomically unlocks the mutex and blocks the thread until the flag is signaled (by the counting thread) again

Example 2-4: `twordcount4.c` (cont.)

```c
/* twordcount4.c - threaded word counter for two files.
 *               - Version 4: condition variable allows counter
 *                            functions to report results early
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <pthread.h>
#include  <ctype.h>

struct arg_set {                /* two values in one arg*/
               char *fname;     /* file to examine      */
               int  count;      /* number of words      */
};

struct arg_set  *mailbox = NULL;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  flag = PTHREAD_COND_INITIALIZER;
```

Example 2-4: `twordcount4.c` (cont.)

```c
int main(int ac, char *av[])
{
        pthread_t       t1, t2;          /* two threads */
        struct arg_set args1, args2;     /* two argsets */
        void            *count_words(void *);
        int             reports_in = 0;
        int             total_words = 0;

        if ( ac != 3 ){
                printf("usage: %s file1 file2\n", av[0]);
                exit(1);
        }

        args1.fname = av[1];
        args1.count = 0;
        pthread_create(&t1, NULL, count_words, (void *) &args1);

        args2.fname = av[2];
        args2.count = 0;
        pthread_create(&t2, NULL, count_words, (void *) &args2);
```

```c
        while( reports_in < 2 ){
                printf("MAIN: waiting for flag to go up\n");
                pthread_cond_wait(&flag, &lock);  /* wait for notify */
                printf("MAIN: Wow! flag was raised, I have the lock\n");
                printf("%7d: %s\n", mailbox->count, mailbox->fname);
                total_words += mailbox->count;
                if ( mailbox == &args1)
                        pthread_join(t1,NULL);
                if ( mailbox == &args2)
                        pthread_join(t2,NULL);
                mailbox = NULL;
                pthread_cond_signal(&flag);      /* announce state change */
                reports_in++;
        }
        printf("%7d: total words\n", total_words);

        return 0;
}
```

Example 2-4: `twordcount4.c` (cont.)

```c
void *count_words(void *a)
{
        struct arg_set *args = a;       /* cast arg back to correct type */
        FILE *fp;
        int  c, prevc = '\0';

        if ( (fp = fopen(args->fname, "r")) != NULL ){
                while( ( c = getc(fp)) != EOF ){
                        if ( !isalnum(c) && isalnum(prevc) )
                                args->count++;
                        prevc = c;
                }
                fclose(fp);
        } else
                perror(args->fname);
        printf("COUNT: waiting to get lock\n");
        pthread_mutex_lock(&lock);      /* get the mailbox */
        printf("COUNT: have lock, storing data\n");
        if ( mailbox != NULL ){
                printf("COUNT: oops..mailbox not empty. wait for signal\n");
                pthread_cond_wait(&flag,&lock);
        }
        mailbox = args;                 /* put ptr to our args there */
        printf("COUNT: raising flag\n");
        pthread_cond_signal(&flag);     /* raise the flag */
        printf("COUNT: unlocking box\n");
        pthread_mutex_unlock(&lock);    /* release the mailbox */
        return NULL;
}
```

Example 2-4: `twordcount4.c` (cont.)

- Execution

```
dynam@DESKTOP-Q4IJBP7:~/lab13$ make
cc -o twc4 twordcount4.c -lpthread
dynam@DESKTOP-Q4IJBP7:~/lab13$ ./twc4 /etc/group /usr/share/dict/words
MAIN: waiting for flag to go up
COUNT: waiting to get lock
COUNT: have lock, storing data
COUNT: raising flag
COUNT: unlocking box
MAIN: Wow! flag was raised, I have the lock
    249: /etc/group
MAIN: waiting for flag to go up
COUNT: waiting to get lock
COUNT: have lock, storing data
COUNT: raising flag
COUNT: unlocking box
MAIN: Wow! flag was raised, I have the lock
 131756: /usr/share/dict/words
 132005: total words
```

Example 2-4: Functions for Condition Variables - `pthread_cond_wait`

pthread_cond_wait		
PURPOSE	Blocks a thread on a condition variable	
INCLUDE	`#include <pthread.h>`	
USAGE	`int pthread_cond_wait(pthread_cond_t *cond,` `pthread_mutex_t *mutex);`	
ARGS	cond	pointer to a condition variable
	mutex	pointer to a mutex
RETURNS	0	if successful
	errcode	if not successful

Example 2-4: Functions for Condition Variables - `pthread_cond_signal`

pthread_cond_signal	
PURPOSE	Unblocks a thread waiting on a condition variable
INCLUDE	`#include <pthread.h>`
USAGE	`int pthread_cond_signal(pthread_cond_t *cond);`
ARGS	cond pointer to a condition variable
RETURNS	0 if successful errcode if not successful

Threads vs. Processes

Processes	Threads
Supports ***parallelism***; ***multitasking***	
• Part of the Unix since the beginning	• Added later
• Model of the process is “clear and uniform.”	• Evolved from variety of sources • Different type of threads with different attributes • *POSIX threads* used in our examples
• Have “own” data space, file descriptors, and process ID number	• “Share” one data space, set of file descriptors, and process ID number

Threads vs. Processes (cont.)

Processes	Threads
• Heavyweight • Low throughput • Long response time • High IPC overhead via pipes, sockets, signal, `exit/wait` • Little synchronization overhead by no shared variables • Easier to debug	• Lightweight • High throughput • Short response time • Low IPC overhead via global variables • Much synchronization overhead by mutex and conditional variables • Harder to debug
• `fork()` can create a new process • `wait()` can wait for a child to complete.	• `pthread_create()` can create a new thread • `pthread_join()` waits for a thread to be done.
• `ps` *can* list processes	• `ps` *cannot* list threads

More Stuff about Threads

- 1) Shared Data Space
  - Multiple threads can read a large, complex tree-structured database in memory easily
    - Multiple queries from clients can be served from one process
    - Variables do not change; so no problems!
  - But consider another example: a program using `malloc` and `free` to manage memory.
    - One thread: allocates a chunk of memory to store a string
    - Another (different) thread: calls free to deallocate that chunk of memory.
    - Then what happens to the original thread?

More Stuff about Threads (cont.)

- 2) Shared File Descriptors
  - `fork` automatically duplicate the file descriptors to the child processes
    - So if the child process closes a file descriptor inherited from the parent, that file descriptor still open for the parent
  - In a multi-threaded program, it's possible to pass the same file descriptor to *two different* threads
    - Both those values refer to the same file descriptor
    - If one thread *closes* the file descriptor, then that file descriptor is *closed* for all threads in the same process although other threads need that connection

More Stuff about Threads (cont.)

- 3) `fork`, `exec`, `exit`, and `signal`s:
  - All threads "share" the same process
    - If one thread calls `exec`, that thread will surprise the other threads by replacing the current program with a new one
    - If one thread calls `exit` or crashes, the "entire" process finishes or dies
    - If one thread calls `fork`, only that thread is running in the new process
      - No other threads get duplicated in the new process

Summary

- A thread of execution is the flow of control through a program
  - The pthreads library allows a program to run several functions at the same time
- Functions running at the same time have their own local variables
  - But they share all global variables and dynamically allocated data
- When threads share a variable, they need to make sure they don’t get in each other’s way
  - Threads may use a mutex lock to make sure only one thread is using a shared variable at a time

Summary (cont.)

- When threads need to coordinate or synchronize their action, they may use a condition variable.
  - One thread waits for the condition variable to change in a specific way
  - The other thread signals the variable to change.
- Threads need to use a mutex lock to prevent simultaneous access to functions that operate on shared resources.
  - Functions that are not reentrant must be protected this way

Appendix

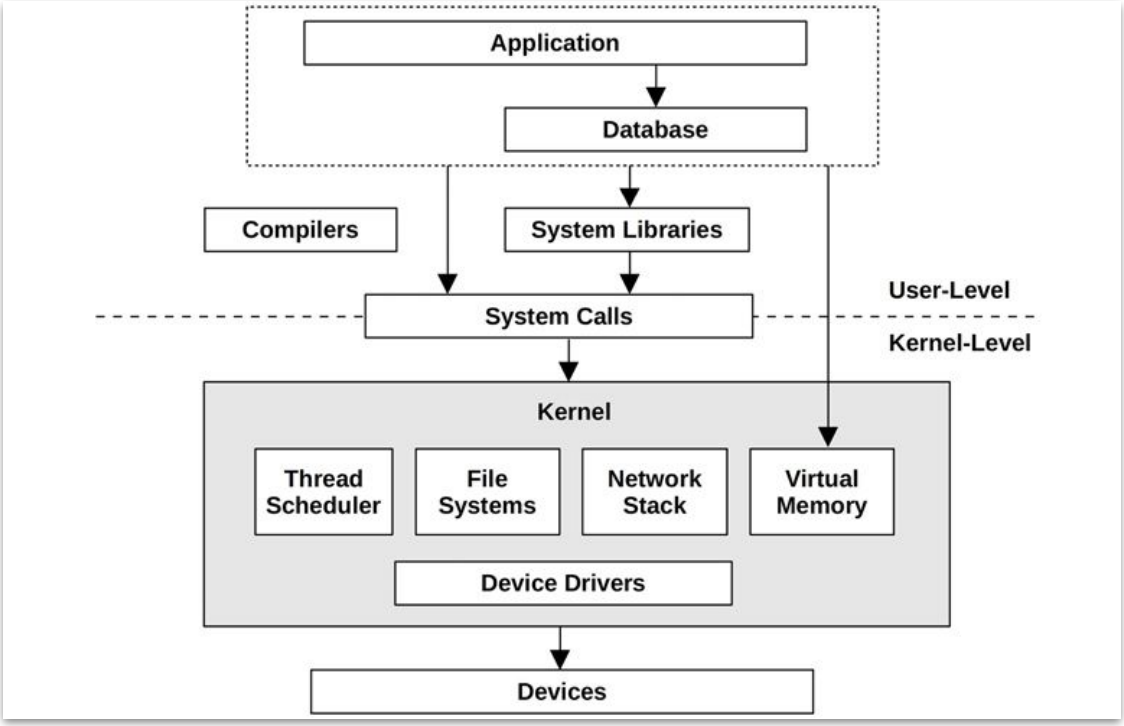
Systems Performance Intro

Introduction

- Systems performance
  - studies the performance of an entire computer system
    - including all major software and hardware components
- Data path from storage devices to application software
  - If you don’t have a diagram of your environment showing the data path, find one or draw it yourself
- Goal
  - to improve the end-user experience by **reducing latency**
  - to reduce computing cost
    - can be achieved by eliminating inefficiencies, improving system throughput, and general tuning

Introduction (cont.)

- Full stack (in terms of systems performance)
  - The entire software stack from the *application* down to the *hardware*
    - including system libraries, the kernel, and the hardware itself



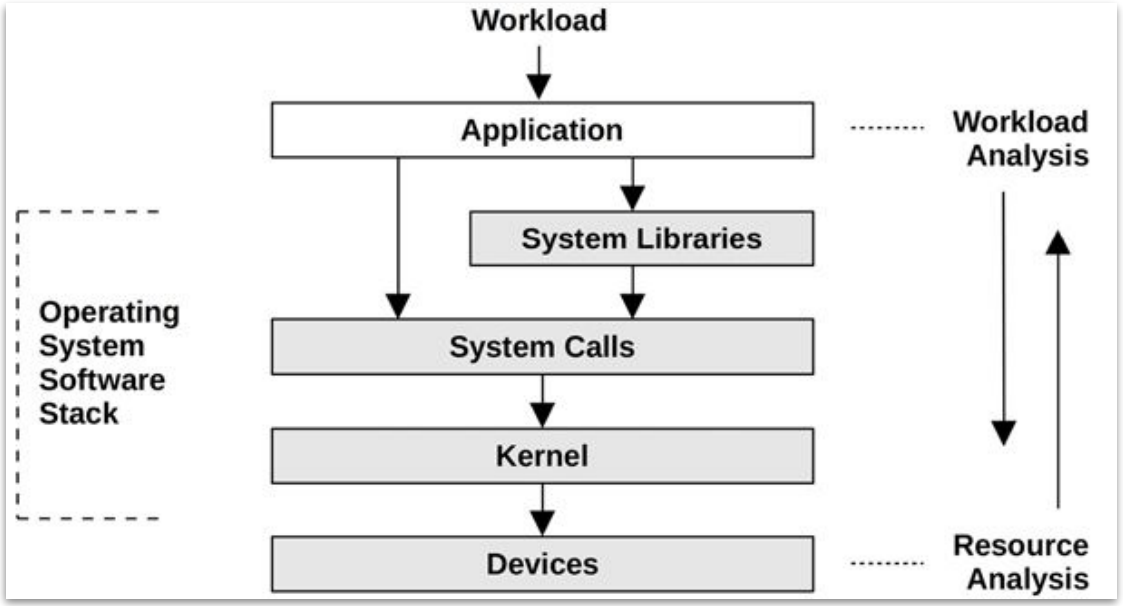
< Generic system software stack on a single server >

Introduction (cont.)

- Systems performance is done by a variety of job roles
  - including system administrators, site reliability engineers, application developers, network engineers, database administrators, web administrators, and other support staff
  - For some performance issues, finding the root cause or contributing factors requires a *cooperative effort* from more than one team

Two Perspectives

- Workload analysis
  - By application developers for the delivered performance of the workload
- Resource analysis
  - By system administrators, who are responsible for the system resources



< Analysis perspectives >

Performance is Challenging

- Subjectivity (주관성)
  - e.g., The average disk I/O response time is 1 ms. Is this “good” or “bad”?
- Complexity
  - Complexity of systems and the lack of an obvious starting point for analysis
- Multiple causes
- Multiple performance issues
  - The real task *isn’t finding* an issue; it’s **identifying** which issue or issues matter *the most*