

CSCE 515: Computer Network Programming

----- Processes vs. Threads

Wenyuan Xu

Department of Computer Science and Engineering
University of South Carolina

Unix processes and threads

■ Processes

- `fork()`
- `wait()` & `waitpid()`

■ Threads

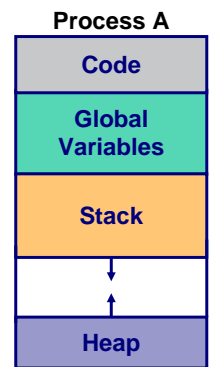
- threads vs. processes
- synchronization

CSCE515 – Computer Network Programming

Processes

What is a process?

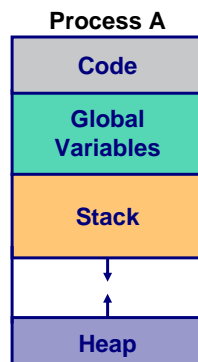
- A program *in execution*
- *context* (the information/data) maintained for an executing program.
- What makes up a process?
 - program code
 - machine registers
 - global data
 - stack
 - open files (file descriptors)
 - an environment (environment variables; credentials for security)



CSCE515 – Computer Network Programming

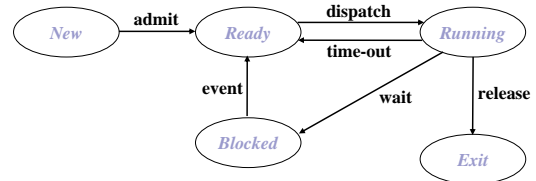
Some of the Context Information

- *Process ID* (`pid`) `getpid()`
 - unique integer
- *Parent process ID* (`ppid`) `getppid()`
- *Real User ID*
 - ID of user/process which started this process
- *Effective User ID*
 - ID of user who wrote the process' program
- *Current directory*
- *File descriptor table*
- *Environment*
 - VAR=VALUE pairs
- *Pointer to program code*
- *Pointer to data*
 - Memory for global vars
- *Pointer to stack*
 - Memory for local vars
- *Pointer to heap*
 - Dynamically allocated
- *Execution priority*
- *Signal information*



CSCE515 – Computer Network Programming

5 State Model - More realistic



- *New*: The process is being created.
- *Running*: Instructions being executed.
- *Blocked* (waiting): Must wait for some event to occur.
- *Ready*: Runnable but waiting to be assigned to a processor.
- *Exit* (terminate): The process has finished execution.

CSCE515 – Computer Network Programming

Unix processes

- The only way to create a new process in UNIX is to duplicate an existing process
- Parents create children; results in a tree of processes.
- Who is the ancestor of all processes?
 - 0: process scheduler ("swapper") system process
 - 1: init process, invoked after bootstrap, mother of all processes.

CSCE515 – Computer Network Programming

Create a new process - fork()

- Creates a child process by making a copy of the parent process --- an **exact** duplicate.
 - Implicitly specifies code, registers, stack, data, files
- `fork()` is called *once* but it returns *twice*
- Return value:
 - 0: return in the child
 - Non-0: the PID of the newly created process

```
listenfd=socket(...)
bind(listenfd...)
listen(listenfd,LISTENQ);
For( ; ; ) {
    connfd = accept(listenfd, ...);
    if ( (pid = fork())==0) {
        close(listenfd);
        doit(connfd);
        close(connfd);
        exit(0);
    }
}
close(connfd);
```

CSCE515 – Computer Network Programming

Terminate a process

- `exit()` is called
 - closes open files, sockets
 - releases other resources
 - saves resource usage statistics and exit status in proc structure
 - wakeup parent
 - calls switch
- Process is in **zombie** state

CSCE515 – Computer Network Programming

Special Exit Cases

- 1) A child exits when its parent is not currently executing `wait()`
 - the child becomes a *zombie*
 - status data about the child is stored until the parent does a `wait()`
- 2) A parent exits when 1 or more children are still running
 - children are adopted by the system's initialization process (`/etc/init`)
 - it can then monitor/kill them
- *Whenever we fork children, we must wait for them to prevent them from becoming zombies!*

CSCE515 – Computer Network Programming

wait Actions

```
#include <sys/wait.h>
pid_t wait(int *stat)
```

- Return
 - the PID of the terminated child
 - The termination status of the child
- A process that calls `wait()` can:
 - *suspend (block)* if all of its children are still running, or
 - *return* immediately with the *termination* status of a child, or
 - *return* immediately with an *error* if there are no child processes.

CSCE515 – Computer Network Programming

A server that collects child

```
listenfd=socket(...)
bind(listenfd...)
listen(listenfd,LISTENQ);
For( ; ; ) {
    connfd = accept(listenfd, ...);
    if ( (pid = fork())==0) { //child
        close(listenfd);
        doit(connfd);
        close(connfd);
        exit(0);
    } else { //parent
        wait( (int *)0 );
        printf( "child finished\n" );
    }
}
close(connfd);
```

Is the server still concurrent?

CSCE515 – Computer Network Programming

A better solution

- When a child terminate, there will be a SIGCHLD signal
- The parent process should catch SIGCHLD
- Within the signal handler, `wait` should be called
- To fetch the child exit status, use the macros:
 - `WIFEXITED`: the child exited normally
 - `WIFSIGNALED`: the child exited by a signal

CSCE515 – Computer Network Programming

Handling SIGCHLD signals

```
main(int argc, char **argv)
{ listenfd=Socket(...)
  Bind(listenfd...)
  Listen(listenfd,LISTENQ);
  Signal(SIGCHLD, sig_chld);
  For( ; ; ) {
    connfd = accept(listenfd,
...);
    if ( (pid = Fork())==0) {
      Close(listenfd);
      doit(connfd);
      Close(connfd);
      exit(0);
    }
  }
  Close(connfd);
}
```

Main function

```
void sig_chld(int signo)
{
  pid_t pid;
  int stat;

  pid = Wait(&stat);
  if (WIFEXITED(stat))
    printf("child %d
terminated normally\n", pid)
  return;
}
```

Signal handler

CSCE515 – Computer Network Programming

waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid( pid_t pid, int *status, int opts )
```

- `waitpid` - can wait for a particular child
- `pid == -1`
 - Wait for any child process.
 - Same behavior which `wait()` exhibits.
- `pid == 0`
 - Wait for any child process whose process group ID is equal to that of the calling process.
- `pid > 0`
 - Wait for the child whose process ID is equal to the value of `pid`.

CSCE515 – Computer Network Programming

waitpid()

■ options -

- `WNOHANG`
 - Return immediately if no child has exited.

■ Return value

- The process ID of the child which exited.
- -1 on error; 0 if `WNOHANG` was used and no child was available.

CSCE515 – Computer Network Programming

Handling SIGCHLD signals

```
main(int argc, char **argv)
{ listenfd=Socket(...)
  Bind(listenfd...)
  Listen(listenfd,LISTENQ);
  Signal(SIGCHLD, sig_chld);
  For( ; ; ) {
    connfd = Accept(listenfd,
...);
    if ( (pid = fork())==0) {
      Close(listenfd);
      doit(connfd);
      Close(connfd);
      exit(0);
    }
  }
  Close(connfd);
}
```

Main function

```
void sig_chld(int signo)
{
  pid_t pid;
  int stat;

  while( (pid = Waitpid(-1,
&stat, WNOHANG))> 0)
    printf("child %d
terminated normally\n", pid)
  return;
}
```

Signal handler

CSCE515 – Computer Network Programming

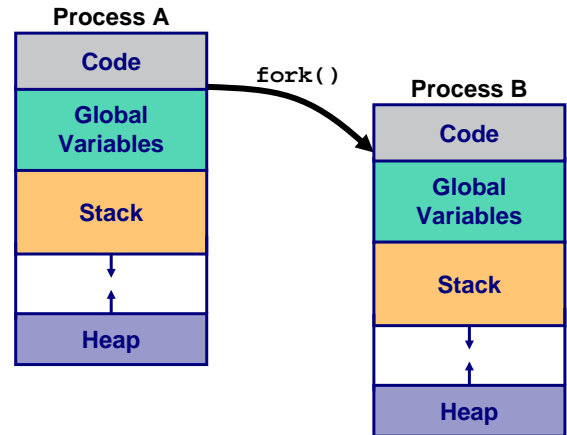
Threads

Threads vs. Processes

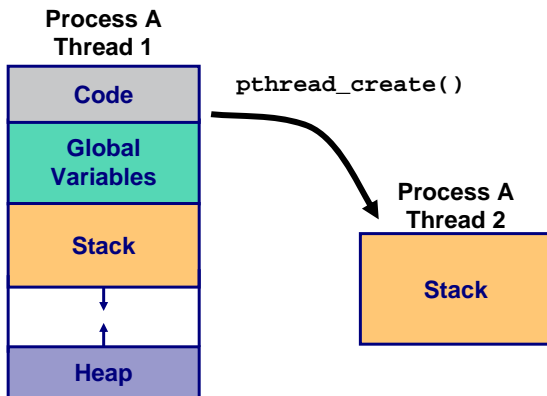
Creation of a new process using fork is *expensive* (time & memory).

A thread (sometimes called a *lightweight process*) does not require lots of memory or startup time.

fork()



pthread_create()



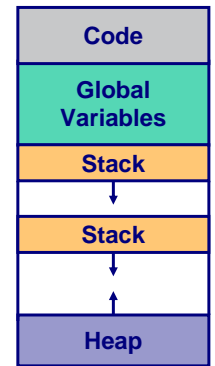
Multiple Threads

Each process can include many threads.

All threads of a process share:

- memory (program code, heap and global data)
- open file/socket descriptors
- signal handlers and signal dispositions
- working environment (current directory, user ID, etc.)

Multi Threaded Process



Thread-Specific Resources

Each thread has it's own:

- Thread ID (integer)
- Stack, Registers, Program Counter
- `errno` (if not - `errno` would be useless!)

Threads within the same process can communicate using shared memory.

Must be done carefully!

POSIX Threads

- Thread variants
 - POSIX (pthreads)
 - Sun threads (mostly obsolete)
 - Java threads
- We will focus on *POSIX Threads* - most widely supported threads programming API.
- Solaris - you need to link with `"-lpthread"`

Thread Creation

```
pthread_create(  
    pthread_t *tid,  
    const pthread_attr_t *attr,  
    void *(*func)(void *),  
    void *arg);
```

func is the function to be called.
When **func()** returns the thread is terminated.

CSCE515 – Computer Network Programming

pthread_create()

- The return value is 0 for OK.
 - *positive error number on error.*
 - EAGAIN, system has reach the limit on the number of threads
- Does *not* set **errno** !!!
- Thread ID is returned in **tid**

CSCE515 – Computer Network Programming

pthread_t *tid

The book says you can specify NULL for **tid** (thread ID), I've found this doesn't always work!

Thread attributes can be set using **attr**, including detached state and scheduling policy. You can specify NULL and get the system defaults.

CSCE515 – Computer Network Programming

Thread IDs

Each thread has a unique ID, a thread can find out it's ID by calling **pthread_self()**.

Thread IDs are of type **pthread_t** which is usually an unsigned int. When debugging, it's often useful to do something like this:

```
printf("Thread %u:\n",pthread_self());
```

CSCE515 – Computer Network Programming

Thread Arguments

- When **func()** is called the value **arg** specified in the call to **pthread_create()** is passed as a parameter.
- **func** can have only 1 parameter, and it can't be larger than the size of a **void ***.

CSCE515 – Computer Network Programming

Example

```
main(int argc, char **argv)  
{ listenfd=socket(...)  
  Bind(listenfd...)  
  Listen(listenfd,LISTENQ);  
  For( ; ; ) {  
    connfd = Accept(listenfd,  
...);  
    if ( (pid = Fork())==0 ) {  
      Close(listenfd);  
      doit(connfd);  
      Close(connfd);  
      exit(0);  
    }  
  }  
  Close(connfd);  
}
```

Process version

```
main(int argc, char **argv)  
{ pthread_t tid;  
  
  listenfd=socket(...)  
  Bind(listenfd...)  
  Listen(listenfd,LISTENQ);  
  For( ; ; ) {  
    connfd =  
    Accept(listenfd, ...);  
    Pthread_create(&tid,  
NULL, &doit, (void *)  
connfd);  
  }  
}  
  
static void doit (void *arg)  
{  
  ...  
  Close( (int) arg);  
  return NULL;  
}
```

Thread version

CSCE515 – Computer Network Programming

Will this concurrent server work?

```
main(int argc, char **argv)
{
    pthread_t tid;

    listenfd=socket(...)
    Bind(listenfd..)
    Listen(listenfd,LISTENQ);
    For( ; ; ) {
        connfd = Accept(listenfd,
        ...);
        Pthread_create(&tid, NULL,
        &doit, (void *) connfd);
    }
}
static void doit (void *arg)
{
    int connfd = (int) arg;
    ...
    Close(connfd);
    return NULL;
}
```

pass int

```
main(int argc, char **argv)
{
    pthread_t tid;

    listenfd=socket(...)
    Bind(listenfd..)
    Listen(listenfd,LISTENQ);
    For( ; ; ) {
        connfd = Accept(listenfd,
        ...);
        Pthread_create(&tid, NULL,
        &doit, (void *) &connfd);
    }
}
static void doit (void *arg)
{
    int connfd = *((int *) arg);
    ...
    Close(connfd);
    return NULL;
}
```

pass a pointer

CSCE515 – Computer Network Programming

CSCE515 – Computer Network Programming

Thread Arguments (cont.)

- What if you want to pass many parameters to the `func()`?
- Complex parameters can be passed by creating a structure and passing the address of the structure.
- The structure can't be a local variable (of the function calling `pthread_create`)!!
 - threads have different stacks!

CSCE515 – Computer Network Programming

Thread args example

```
struct { int x,y } 2ints;

void *blah( void *arg) {
    struct 2ints *foo = (struct 2ints *) arg;
    printf("%u sum of %d and %d is %d\n",
        pthread_self(), foo->x, foo->y,
        foo->x+foo->y);
    return(NULL);
}
```

CSCE515 – Computer Network Programming

Thread Lifespan

Once a thread is created, it starts executing the function `func()` specified in the call to `pthread_create()`.

If `func()` returns, the thread is terminated.

A thread can also be terminated by calling `pthread_exit()`.

If `main()` returns or any thread calls `exit()` all threads are terminated.

CSCE515 – Computer Network Programming

Joinable Thread

- Each thread can be either *joinable* or *detached*.
- **Joinable (the default)**: on thread termination the thread ID and exit status are saved by the OS.
- One thread can "join" another by calling `pthread_join` - which waits (blocks) until a specified thread exits.

```
int pthread_join( pthread_t tid,
                void **status);
```

CSCE515 – Computer Network Programming

Detached State

Detached: on termination all thread resources are released by the OS. A detached thread cannot be joined.

No way to get at the return value of the thread. (a pointer to something: `void *`).

```
int pthread_detach( pthread_t tid);
```

CSCE515 – Computer Network Programming

A Detached Thread

```
main(int argc, char **argv)
{
    pthread_t tid;

    listenfd=socket(...)
    Bind(listenfd...)
    Listen(listenfd,LISTENQ);
    For( ; ; ) {
        connfd = Accept(listenfd, ...);
        Pthread_create(&tid, NULL, &doit, (void *) connfd);
    }
}
static void doit (void *arg)
{
    Pthread_detach(pthread_self());
    ...
    Close( (int) arg);
    return NULL;
}
```

CSCE515 – Computer Network Programming

Shared Global Variables

```
int counter=0;
void *pancake(void *arg) {
    counter++;
    printf("Thread %u is number %d\n",
        pthread_self(),counter);
}
main() {
    int i; pthread_t tid;
    for (i=0;i<10;i++)
        pthread_create(&tid,NULL,pancake,NULL);
}
```

CSCE515 – Computer Network Programming

DANGER! DANGER! DANGER!

- Sharing global variables is dangerous - two threads may attempt to modify the same variable at the same time.
- Example
 - Suppose you want to deposit \$5 to my account and I want to withdraw \$10
 - What should the balance be after the two transactions have been completed?
 - What might happen instead if the two transactions were executed concurrently?

CSCE515 – Computer Network Programming

DANGER! DANGER! DANGER!

- The balance might be SB + 5
 - U reads SB
 - I read SB
 - I compute SB - 10 and save new balance
 - U compute SB + 5 and save new balance
- The balance might be SB -10
 - How?
- Ensure the orderly execution of cooperating threads/processes

Just because you don't see a problem when running your code doesn't mean it can't and won't happen!!!!

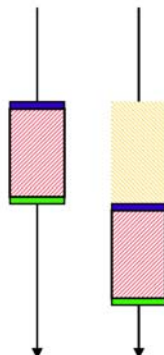
CSCE515 – Computer Network Programming

Avoiding Problems

threads includes support for *Mutual Exclusion* primitives that can be used to protect against this problem.

The general idea is to *lock* something before accessing global variables and to *unlock* as soon as you are done.

Shared socket descriptors should be treated as global variables!!!



CSCE515 – Computer Network Programming

pthread_mutex

A global variable of type `pthread_mutex_t` is required (*lock*):

```
pthread_mutex_t counter_mtx=
    PTHREAD_MUTEX_INITIALIZER;
```

Initialization to `PTHREAD_MUTEX_INITIALIZER` is required for a static variable!

CSCE515 – Computer Network Programming

Locking and Unlocking

- To lock use:

```
pthread_mutex_lock(pthread_mutex_t &);
```

- To unlock use:

```
pthread_mutex_unlock(pthread_mutex_t &);
```

- Both functions are blocking!

CSCE515 – Computer Network Programming

Example Problem (Quiz)

A server creates a thread for each client. No more than n threads (and therefore n clients) can be active at once.

How can we have the main thread know when a child thread has terminated and it can now service a new client?

CSCE515 – Computer Network Programming

pthread_join() doesn't help

`pthread_join` (which is sort of like `waitpid()`) requires that we specify a thread id.

We can wait for a specific thread, but we can't wait for "the next thread to exit".

CSCE515 – Computer Network Programming

Use a global variable?

When each thread starts up:

- acquires a lock on the variable (using a mutex)
- increments the variable
- releases the lock.

When each thread shuts down:

- acquires a lock on the variable (using a mutex)
- decrements the variable
- releases the lock.

CSCE515 – Computer Network Programming

Inside Threads

```
int active_threads = 0;
pthread_mutex_t at_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void * cld_func (void *vptr)
{
    pthread_mutex_lock(& at_mutex);
    active_threads ++;
    pthread_mutex_unlock(& at_mutex);
    ...
    pthread_mutex_lock(& at_mutex);
    active_threads--;
    pthread_mutex_unlock(& at_mutex);
    pthread_exit((void*) 0);
}
```

CSCE515 – Computer Network Programming

What about the main loop?

```
active_threads=0;
// start up n threads on first n clients
// make sure they are all running
while (1) {
    // have to lock/release active_threads
    pthread_mutex_lock(& at_mutex);
    if (active_threads < n)
        // start up thread for next client
        pthread_mutex_unlock(& at_mutex);
    busy_waiting(is_bad);
}
```

CSCE515 – Computer Network Programming

Condition Variables

threads support *condition variables*, which allow one thread to wait (sleep) for an event generated by any other thread.

This allows us to avoid the *busy waiting* problem.

```
pthread_cond_t foo =  
    PTHREAD_COND_INITIALIZER;
```

CSCE515 – Computer Network Programming

Condition Variables (cont.)

A condition variable is always used with mutex.

```
pthread_cond_wait(pthread_cond_t *cptr,  
                  pthread_mutex_t *mptr);
```

- puts the calling thread to sleep
- releases the mutex lock it holds
- when returns, the thread holds the mutex.

```
pthread_cond_signal(pthread_cond_t *cptr);
```

CSCE515 – Computer Network Programming

Condition Variables (cont.)

A condition variable is always used with mutex.

```
pthread_cond_wait(pthread_cond_t *cptr,  
                  pthread_mutex_t *mptr);
```

```
pthread_cond_signal(pthread_cond_t *cptr);
```

- Awakens one thread that is waiting on the condition variable

*don't let the word signal confuse you -
this has nothing to do with Unix signals*

CSCE515 – Computer Network Programming

Revised strategy

Each thread decrements `active_threads` when terminating and calls `pthread_cond_signal` to wake up the main loop.

The main thread increments `active_threads` when each thread is started and waits for changes by calling `pthread_cond_wait`.

CSCE515 – Computer Network Programming

Revised strategy

All changes to `active_threads` must be inside the lock and release of a mutex.

If two threads are ready to exit at (nearly) the same time – the second must wait until the main loop recognizes the first.

We don't lose any of the condition signals.

CSCE515 – Computer Network Programming

Global Variables

```
// global variable the number of active  
// threads (clients)  
int active_threads=0;
```

```
// mutex used to lock active_threads  
pthread_mutex_t at_mutex =  
    PTHREAD_MUTEX_INITIALIZER;
```

```
// condition var. used to signal changes  
pthread_cond_t at_cond =  
    PTHREAD_COND_INITIALIZER;
```

CSCE515 – Computer Network Programming

Child Thread Code

```
void *cld_func(void *arg) {
    . . .
    // handle the client
    . . .
    pthread_mutex_lock(&at_mutex);
    active_threads--;
    pthread_cond_signal(&at_cond);
    pthread_mutex_unlock(&at_mutex);
    return();
}
```

CSCE515 – Computer Network Programming

Main thread

```
// no need to lock yet
active_threads=0;
while (1) {
    pthread_mutex_lock(&at_mutex);
    while (active_threads < n ) {
        active_threads++;
        pthread_start(...)
    }
    pthread_cond_wait( &at_cond, &at_mutex);
    pthread_mutex_unlock(&at_mutex);
}
```

IMPORTANT!
Must happen while
the mutex lock is
held.

CSCE515 – Computer Network Programming

Term

- condition variable: pthread_cond_t
 - at_cond
- condition: some variable shared between threads
 - active_threads
- Why a mutex always associated with a condition variable?

CSCE515 – Computer Network Programming

Other pthread functions

Sometimes a function needs to have thread specific data (for example, a function that uses a static local).

Functions that support thread specific data:

```
pthread_key_create()
pthread_once()
pthread_getspecific()
pthread_setspecific()
```

The book has a nice
example creating a
safe and efficient
readline()

CSCE515 – Computer Network Programming

Thread Safe library functions

- You have to be careful with libraries.
- If a function uses any static variables (or global memory) it's not safe to use with threads!
- The book has a list of the Posix thread-safe functions...

CSCE515 – Computer Network Programming

Thread Summary

Threads are awesome, but dangerous. You have to pay attention to details or it's easy to end up with code that is incorrect (doesn't always work, or hangs in deadlock).

Posix threads provides support for mutual exclusion, condition variables and thread-specific data.

CSCE515 – Computer Network Programming



Assignment & Next time

- Reading:

- UNP 5.9, 5.10, 26.1-26.4, 26.7, 26.8 **

- Next Lecture:

- IP, Routing