Processes vs. Threads

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

- Threads
  - threads vs. processes
  - synchronization

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)

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
- Pointer to stack
  - Memory for local vars
- Pointer to heap
  - Dynamically allocated
- Execution priority
- Signal information

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.
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.

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);
  } else { //parent
    wait( (int *)0 );
    printf("child finished\n");
  }
}
close(connfd);
```

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

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!

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.

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?
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:
  - `WEXITED`: the child exited normally
  - `WIFSIGNALED`: the child exited by a signal

Handling SIGCHLD signals

```c
#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.
  - `0` Same behavior which `wait()` exhibits.
  - `> 0` Wait for the child whose process ID is equal to the value of `pid`

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listendf);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

Signal handler

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listendf);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listenfd);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listenfd);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listenfd);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listenfd);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```

Main function

```c
main(int argc, char **argv)
{
    listendf = socket(…);
    bind(listenfd);
    listen(listenfd, LISTENQ);
    Signal(SIGCHLD, sig_chld);
    For( ; ; ) {
        connfd = accept(listenfd, …);
        if ( (pid = fork())==0) {
            close(listendf);
            doit(connfd);
            close(connfd);
            exit(0);
        }
    }
    close(connfd);
}
```

```c
void sig_chld(int signo)
{
    pid_t pid;
    int stat;
    pid = wait(&stat);
    if (WEXITED(stat))
        printf("child %d terminated normally\n", pid);
    return;
}
```
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()

Process A

Code
Global Variables
Stack
Heap

Process B

Code
Global Variables
Stack
Heap

pthread_create()

Process A
Thread 1
Code
Global Variables
Stack
Heap

Thread 2
Stack

Process A

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.)

Thread-Specific Resources

Each thread has its 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

```c
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.

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*

**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.

**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:

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

**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*.

**Example**

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

```c
main(int argc, char **argv)
{
    listenfd=socket(…)
    Bind(listenfd, LISTENQ);
    For( ; ; ) {
        connfd = Accept(listenfd, …);
        if( (pid = Fork())==0 ) {
            doit(connfd);
            Close(connfd);
            doit((int)arg);
            return NULL;
        }
    }
    Thread version
}
```
Will this concurrent server work?

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

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!

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

Thread args example

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

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

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.

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.

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

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 *`).

```c
int pthread_detach( pthread_t tid);
```
A Detached Thread

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

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);
```

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?

Avoiding Problems

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

```
pthread_mutex_t counter_mtx=
    PTHREAD_MUTEX_INITIALIZER;
```

```
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!

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?

Example Problem (Quiz)

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.

Inside Threads

```c
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);
}
```

What about the main loop?

```c
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);
    pthread_mutex_lock(& at_mutex);
    active_threads ++;
    pthread_mutex_unlock(& at_mutex);
    while (active_threads < n) {
        // 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);
        pthread_mutex_lock(& at_mutex);
        active_threads ++;
        pthread_mutex_unlock(& at_mutex);
    }
    pthread_mutex_lock(& at_mutex);
    active_threads--;
    pthread_mutex_unlock(& at_mutex);
}
```
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.

\[
\text{pthread_cond_t foo} = \text{PTHREAD_COND_INITIALIZER};
\]

Condition Variables (cont.)

A condition variable is always used with mutex.

\[
\text{pthread_cond_wait(pthread_cond_t *cptr, pthread_mutex_t *mptr);} \\
\text{\hspace{1cm}} \begin{cases} \text{puts the calling thread to sleep} \cr \text{releases the mutex lock it holds} \cr \text{when returns, the thread holds the mutex.} \end{cases}
\]

\[
\text{pthread_cond_signal(pthread_cond_t *cptr);} \\
\text{\hspace{1cm}} \begin{cases} \text{Awakens one thread that is waiting on the condition variable} \end{cases}
\]

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

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.

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 = 
\text{PTHREAD_MUTEX_INITIALIZER};
```

```
// condition var. used to signal changes 
pthread_cond_t at_cond = 
\text{PTHREAD_COND_INITIALIZER};
```
Child Thread Code

```c
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();
}
```

Main thread

```c
// 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.

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?

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()

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...

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.
Assignment & Next time

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

- **Next Lecture:**
  - IP, Routing