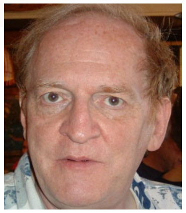
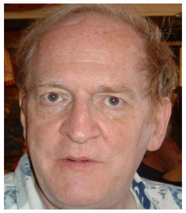
COSC 3360/6310
Wednesday, February 24

Announcements



- First assignment is now due on March 3 anywhere on earth
- Second quiz will be on Monday, March 1
    - Will cover chapter 2 of class notes
        - Three video lectures
            - F. Wednesday February 10
            - G. Monday February 22
            - H. Wednesday February 24

More announcements

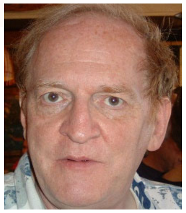


- Some people have not yet accepted the invitations I sent to their UH Mail accounts in January
  - □ Please check your UH mail account (`jsbach@uh.edu`)
- Videos of last two lectures are now online
  - □ Had problems with MS Teams default permissions

Chapter II
Processes

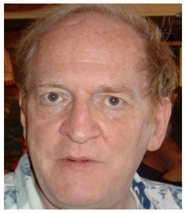
Jehan-François Pâris
jfparis@uh.edu

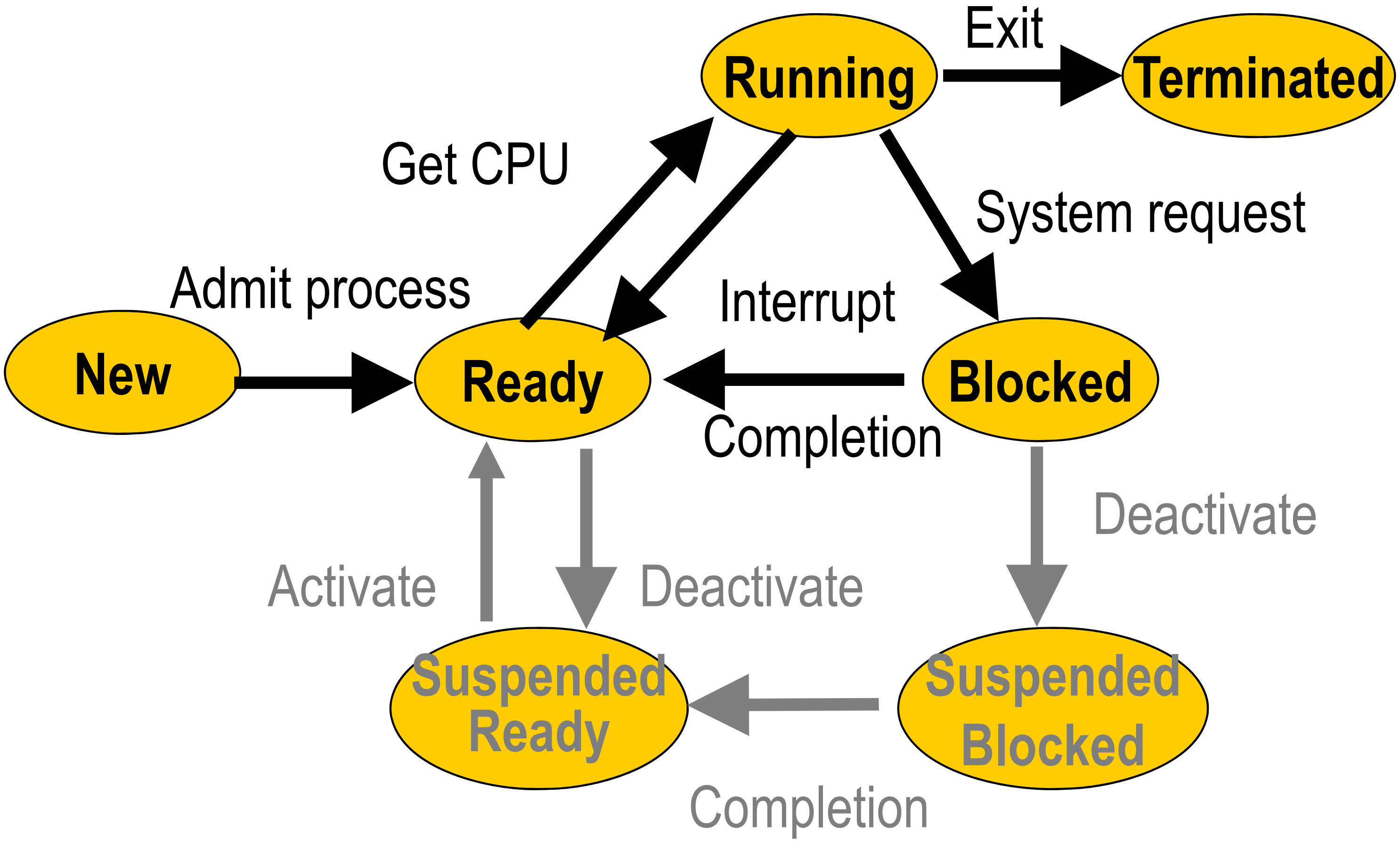
Chapter Overview

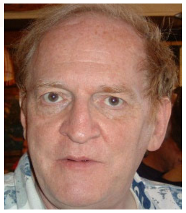


- Processes
- States of a process
- Operations on processes
  - □ `fork()`, `exec()`, `kill()`, `signal()`
- Threads and lightweight processes

How it works



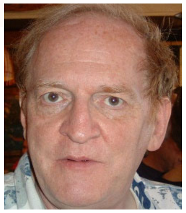

Running
Exit
Terminated
Get CPU
System request
Admit process
Interrupt
New
Ready
Blocked
Completion
Deactivate
Activate
Deactivate
Suspended Ready
Suspended Blocked
Completion



Operations on processes

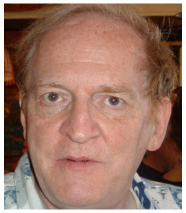
Process creation, deletion, …

Operations on processes

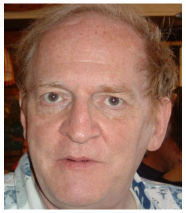


- Process creation
  - `fork()`
  - `exec()`
  - The argument vector
- Process deletion
  - `kill()`
  - `signal()`

What will this program print out?



```
  #include <iostream>
  using namespace std;
  main() {
      if ((pid = fork()) == 0) {
         //child
         cout << "Hello" << endl;
       }
       // parent
       cout << "Goodbye" << endl;
} // main
```

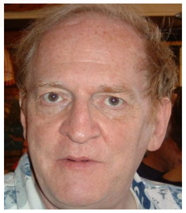


Answer

-
  ```
  Hello
  Goodbye
  Hello
  ```

or

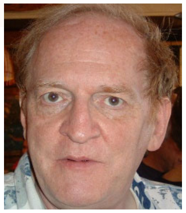
-
  ```
  Goodbye
  Hello
  Goodbye
  ```



Lightweight processes/threads

Kernel supported threads, user-level threads, POSIX threads (pthreads)

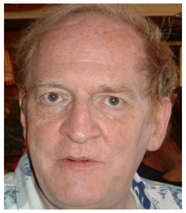
Limitations of processes



- Single threaded server:
  - □ Processes one request at a time

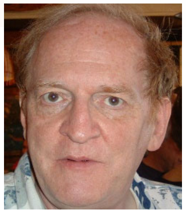
```
for (;;) {
   receive(&client, request);
   process_request(...);
   send(client, reply);
} // for
```

A basic question



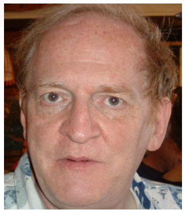
- ***What does a server do when it does not process client requests?***

Three good answers



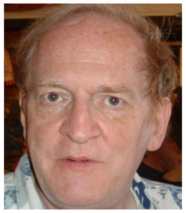
- Nothing
- It waits for client requests
- It "sleeps"
  - ***Blocked state*** *is sometimes called the* ***sleep state***

The problem

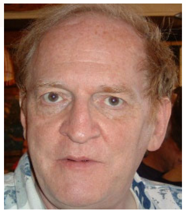


- Most client requests involve disk accesses
  - □ File servers
  - □ Authentications servers
- When this happens, the server remains in the BLOCKED state
  - □ Cannot handle other customers' requests
- Could end doing nothing most of the time
- ***Poor throughput***

An analogy

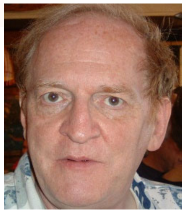


- *In most fast-food restaurants, counter employees process customer orders one order at a time.*
- *Not be possible in a traditional restaurant*
  - □ *A server that would only be able to wait on one table at a time would be idle most of the time.*



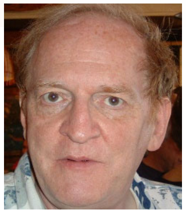
A first solution

```
int pid;
for (;;) {
   receive(&client, request);
   if ((pid = fork()) == 0) {
       process_request(...);
       send(client, reply);
       _exit(0); // done
   } // if
} // for
```



The good and the bad news

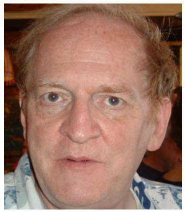
- ***The good news:***
  - Server can now handle several user requests in parallel
- ***The bad news:***
  - **`fork()`** is a ***very expensive*** system call
    - Has to create a new address space



A better solution

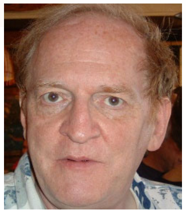
- Provide a faster mechanism for creating cheaper processes:
  - ***Lightweight processes***
  - ***Threads***

How?

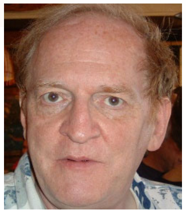


- Lightweight processes and threads ***share the address space of their parent***
  - ***No need to create a new address space***
    - Most expensive step of **`fork()`** system call

Is it not dangerous?



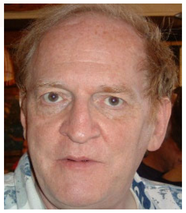
- ***To some extent because***
  - □ No memory protection inside an address space
  - □ Lightweight processes can now interfere with each other
- ***But***
  - □ All lightweight process code is written by the same team

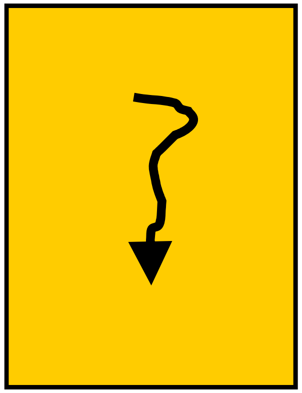


General Concept (I)

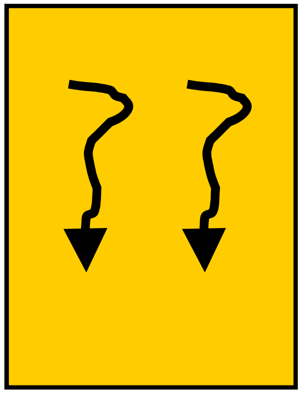
- A ***thread*** or ***lightweight process***
  - Does ***not*** have its ***own address space***
  - Shares it with its parent and other peer threads in the same address space (***task***)
- Each thread has a ***program counter***, a ***set of registers*** and its ***own stack***.
  - *Everything else is shared*

General Concept (II)



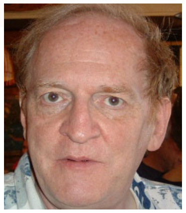


- A regular process (single-threaded)

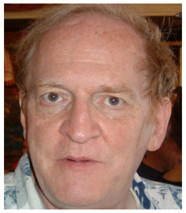


- A process containing several threads

Implementation



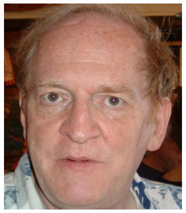
- Threads and LWPs can either be
  - ***Kernel supported:***
    - Mach, Linux, Windows NT and after
  - ***User-level:***
    - Original pthread library, …



Kernel-Supported Threads (I)

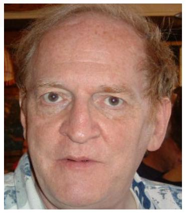
- Managed by the kernel through system calls
- One process table entry per thread
- This is the best solution for *multiprocessor architectures*
  - □ Kernel can allocate ***several processors*** to a ***single multithreaded task***

Kernel-Supported Threads (II)



- Supported by Mach, Linux, Windows NT and more recent systems
- ***Performance Issue:***
  - Switching between two threads in the same task involves a system call
  - Results in ***two context switches***

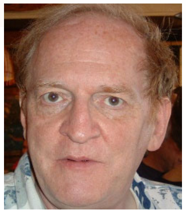
Linux Threads



- **`clone(fn, stack, flags)`**

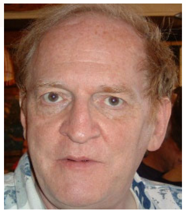
  where
  - □ **`fn`** specifies function to be executed by new thread or process
  - □ **`stack`** points to the stack it will use
  - □ **`flags`** is a set of flags specifying various options
    - **`CLONE_VM`** for threads
    - Regular process if **`CLONE_VM`** is missing

User-Level Threads (I)



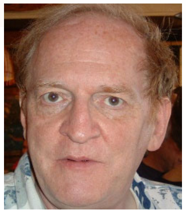
- User-level threads are managed by procedures **within** the task address space
  - □ The ***thread library***
- One process table entry per task/address space
  - □ Kernel is not even aware that process is multithreaded

User-Level Threads (II)



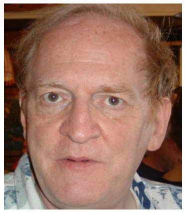
- Can be retrofitted into an OS lacking thread support
  - Portable thread libraries
- ***No performance penalty:***
  - Switching between two threads of the same task is done cheaply within the task
  - Same cost as a procedure call

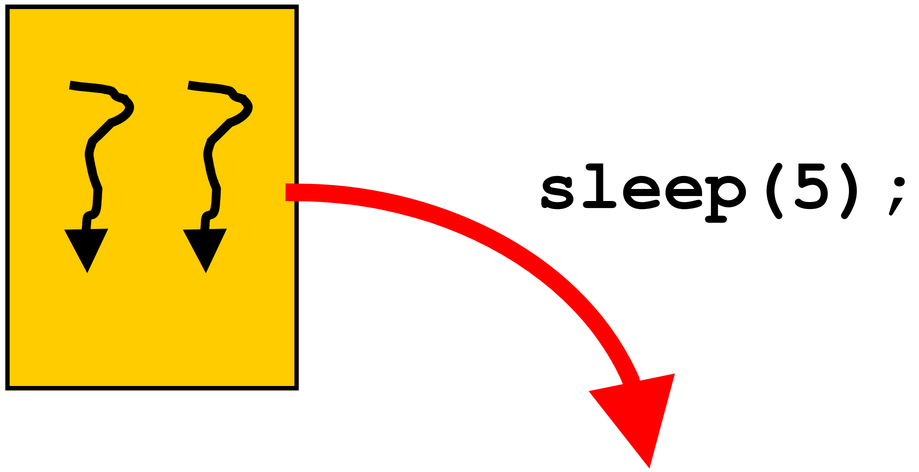
User-Level Threads (III)



- ***Programming issue:***
  - Each time a thread does a ***blocking system call***, kernel will move the ***whole process*** to the ***blocked state***
    - It does not know better
  - Must then use ***non-blocking*** system calls
    - *Complicates programmer's task*

User-Level Threads (IV)





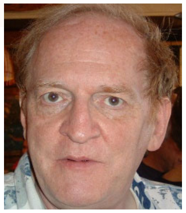
```
sleep(5);
```

<u>Kernel</u>

Process wants to sleep for 5 seconds:

Should be moved it to the blocked state

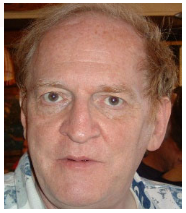
POSIX Threads



- POSIX threads, or ***pthreads***, started as pure user-level threads managed by the POSIX thread library
  - Gained later ***some kernel support***
- Ported to various Unix and Windows systems (***Pthreads-win32***).
- Function names start with **`pthread_`**
- Calls tend to have a complex syntax

An Example (I)

FYI



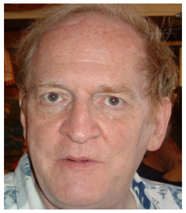
```
#include <pthread.h>
static int count[2];
```

Static variables are shared by all threads

Other variables are stored on the private stack of each thread.

An Example (II)

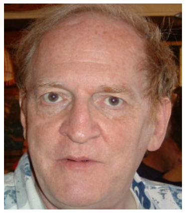
FYI



```
void *child(void *arg) {
    int index;
    index = (int) arg;  // required
    for(;;) {
        printf("Child count: %d\n",
             ++count[index]);
      sleep(1); // one second delay
     } // for loop
} // child
```

An Example (III)

FYI

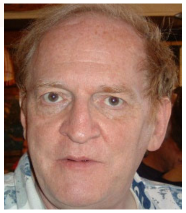


```
int main() {
    thread_t tid; // thread id
    int i = 0;
    pthread_create(&tid, NULL,
                   child, (void *) i);
    // pthread will execute
    // "child" function
```

NULL stack address specifies a new stack "anywhere"

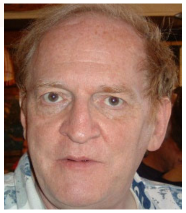
An Example (IV)

FYI



```
    i++; // now i == 1
    while (count[i] < 12) {
        printf("Parent count: %d\n", ++count[i]);
        sleep(1); // one second delay
    } // while loop
     return 0;
} // main
```

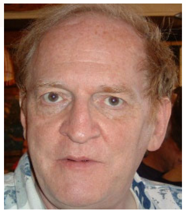
Understanding pthread_create()



FYI

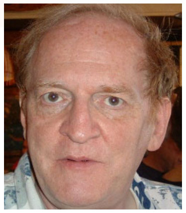
- **`pthread_create()`** has four arguments
  - **`&tid`**
    - Placeholder for **`thread_id`**
  - **`NULL`**
    - Stack address of new stack
    - **`NULL`** means “anywhere”
  - **`start_function`**
    - Void pointer to a function
  - **`(void *) arg`**
    - Sole argument passed to **`start_function`**

Comparing the approaches

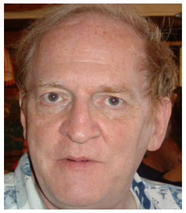


Feature	***Kernel threads***	***User-level threads***
Portability		
Multiprocessing		
Performance		
Ease of use		

Comparing the approaches

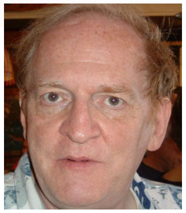


Feature	***Kernel threads***	***User-level threads***
Portability		☑
Multiprocessing		
Overhead		
Ease of use		



Comparing the approaches

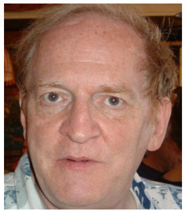
Feature	***Kernel threads***	***User-level threads***
Portability		☑
Multiprocessing	☑	
Overhead		
Ease of use		



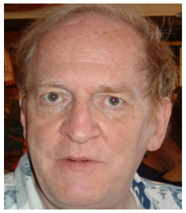
Comparing the approaches

Feature	***Kernel threads***	***User-level threads***
Portability		☑
Multiprocessing	☑	
Overhead		☑
Ease of use		

Comparing the approaches



Feature	***Kernel threads***	***User-level threads***
Portability		☑
Multiprocessing	☑	
Overhead		☑
Ease of use	☑	



Conclusion

- No clear winner between kernel-supported and user-level threads
- Solaris (from Sun, now taken over by Oracle)
  - □ Supports both ***user-level threads*** and ***kernel threads***
  - □ Lets programmers combine them as they need