 Processes

- Process is a program in execution
- A process has a single “thread” of activity
  - Single program counter, stack
  - Code, data, heap, files, signals
- What if we want to do multiple related activities at the same time?
  - Multiple instances of same task
    - Web server: Serve multiple user requests at the same time
  - Multiple components of a task
    - Web browser: Read data from network while displaying graphics

 How to do Multiple Related Activities?

- One approach: Use multiple processes
  - Give one task to each process
- Problem 1: How to share data and communicate?
  - IPC: Pipes, files, shared memory, sockets, signals
  - Requires kernel support
  - Inefficient due to user/kernel crossings
- Problem 2: Overhead
  - Every process has its own memory map and resources
  - Paging and context-switch cost is typically high

Outline

- Threads
  - Thread Definition
  - Thread Usage
  - Kernel vs. User Threads
  - Thread Operations
What is a Thread?

- A thread is
  - An abstraction of a "process activity"
  - An active sequence of instructions in a process

- A process activity is defined by its context:
  - Program counter: Tells which instruction is being executed
  - Stack: Determines where in the program we are (e.g.: which function, what parameters)

 Threads

- Threads exist within a process
  - "Lightweight process"
  - Multiple threads run concurrently within the same process

- Threads share
  - Process code, data, heap
  - Files, signals

- Each thread has its own
  - Program counter, stack, registers, signal mask

- How do threads communicate?

Process vs. Threads

<table>
<thead>
<tr>
<th>Process</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Data</td>
</tr>
<tr>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>Heap</td>
<td>Heap</td>
</tr>
<tr>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>

Multiprogramming vs. Multithreading

- Multiprogramming means
  - OS has several processes in memory at a time and can execute any of them
  - Processes are address-space disjoint

- Multithreading means
  - Process can have multiple threads
  - Threads share address-space
Multithreading Example

```c
foo () {
    ...
}
bar () {
    ...
} void main () {
    ...
    create_thread (foo);
    create_thread (bar);
    ...
}
```

Thread Benefits

- **Concurrency**
  - When one thread is blocked, another can run
  - Great for multi-tasking applications (Web servers, file servers)

- **Resource Sharing**
  - Threads share resources of the process (e.g., code, data, files)
  - Less OS resources used up (e.g., memory, buffers, kernel data structures)

Thread Benefits (contd.)

- **Efficiency**
  - Thread operations cheaper than processes
  - Creating/destroying, context switches, scheduling
  - Communication (Common address-space)

- **Parallelism**
  - Multithreading gives real parallelism on multiprocessor machines
  - Can run multiple threads on multiple processors

Thread Problems

- **Programming Complexity**
  - Non-deterministic behavior
  - Need to be synchronized
  - Difficult to debug

- **Scalability**
  - Stacks could still use up lot of memory
  - Context switch has overhead

- **Portability problems due to different implementations**
Thread Implementation

- Can be implemented in user or kernel space
- User threads are implemented by a user-level runtime system
  - Language support or thread-package library
  - E.g.: Java, Pthreads
- Kernel threads are implemented directly inside the kernel
  - Like processes with shared address-space
  - E.g.: Linux kernel threads

User Threads

- User threads are more light-weight and efficient
  - No kernel scheduling, context-switching
- User threads are more flexible
  - Application-specific scheduling policy
- Blocking I/O Problem
  - If a single user thread blocks, the whole process and hence, all threads block
- User threads cannot exploit parallelism of multiprocessors

Kernel Threads

User vs. Kernel Threads
Hybrid Multithreading Models

- Many systems provide support for both user and kernel threads
- Use a combination of user and kernel threads
- Map user threads to kernel threads
- Dependent on
  - OS thread support
  - User thread library implementation

Pthreads

- POSIX Threads Package
- Provides library calls for creating and managing threads
- Implementation is dependent on system support
  - Could be a combination of user/kernel threads

Thread Operations

- Create a thread
  - Pass it a function and arguments, attributes
- Threads run concurrently
- Join a thread
  - Makes the calling thread wait for a child
- Detach a thread
  - Lets the thread release its resources when done
- Terminate a thread
  - Finish a thread without exiting the process

Thread Creation: pthread_create

```c
pthread_t tid;
pthread_attr_t attr;
int i;

void *foo(void *arg){...}

int main(){
  ...
pthread_create(&tid, &attr, foo, i);
  ...
}
```
Joining Threads: pthread_join

```c
int pthread_join(pthread_t thread, void **valp);
```

- Makes the calling thread wait on another thread
- Similar to waitpid
- Calling thread suspended until target thread finishes
- When target thread terminates:
  - Its return status is passed
  - Its resources are released

Detaching Threads: pthread_detach

```c
int pthread_detach(pthread_t thread);
```

- Makes a thread “independent”
- The thread’s resources are reclaimed when it exits
- Cannot be joined (waited on)
- A thread’s resources not released until it is either detached or joined

Thread Termination

- A thread can exit by
  - Returning from its top-level function
  - Calling pthread_exit
  - Calling exit: Terminates the whole process
- A thread can terminate another thread
  - pthread_cancel
  - The result of this call depends on the target thread’s type and cancellability

Pthread Example

```c
pthread_t tid;
int i=1;
void *foo(void *arg){
    int myval= (int) arg;
    printf("myval=%d\n", myval);
}
int main(){
    ...
    pthread_create(&tid, NULL, foo, i);
    ...
    pthread_join(tid, NULL);
}
```