CS 355
Operating Systems
Threads

Introduction to Threads
• Multitasking OS can do more than one thing concurrently by running more than a single process
• A process can do several things concurrently by running more than a single thread
• Each thread is a different stream of control that can execute its instructions independently.
• Ex: A program (e.g. Browser) may consist of the following threads:
  • GUI thread
  • I/O thread
  • Computation thread

Thread Concept
• Threads are “light-weight” processes, much faster to create and destroy
  – 10-100 times faster than fork()
• Threads share
  – process instructions
  – address space/most data: global variables, open file descriptors, signal handlers, current working directory, process/group/user ID
• Threads do not share
  – thread ID
  – registers: program counter and stack pointer
  – stack [local variables, return address]
  – errno

Implementation of a Process
• Process table – array or linked list
• PCB – define struct

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children's CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When are threads useful?

No memory protection btw threads!!
Challenges in single-threaded proc
- There are basically 4 activities to be scheduled
  - read(ri), read(li), write(lo), write(ro)
- read and write are blocking calls
- So before issuing any of these calls, the program needs to check readiness of devices, and interleave these four operations
  - System calls such as FD_SET() and select()
- Code can be quite complicated

Solution with Threads
```
in(int ri, int lo){
    int done=0;
    char b[MAX];
    int s;
    while (!done) {
        s=read(ri,b[MAX]);
        if (s<=0) done=1;
        if (write(lo,b,s)<=0) done=1;
    }
}
```
```
out(int li, int ro){
    int done=0;
    char b[MAX];
    int s;
    while (!done) {
        s=read(li,b[MAX]);
        if (s<=0) done=1;
        if (write(ro,b,s)<=0) done=1;
    }
}
```

Parallel Algorithms: Eg. mergesort
- Sort on 4 parallel threads
- Merge on 2 parallel threads
- Sort on 2 parallel threads
- Merge

Is there a speed-up?

Benefits of Threads
- Superior programming model of parallel sequential activities with a shared store
- Easier to create and destroy threads than processes.
- Better CPU utilization (e.g. dispatcher thread continues to process requests while worker threads wait for I/O to finish)
  - think keyboard input and disk I/O
- Threads really shine on multi-processor systems, where true parallelism is possible

Thread Usage: Web Server
- Web server process
  - Dispatcher thread
  - Worker thread
  - Network connection
  - User space
  - Kernel space

Pseudo Code
```
while(1){
    get_next_request(&buf);
    handoff(&buf);
}
```
```
while(1) {
    wait_for_work(&buf);
    look_in_cache(&page, &buf);
    if (!INCACHE)
        read_from_disk(&page, &buf);
    return_page(&page);
}
Threads Model
Three threads residing in the same user process

Each thread has its own stack, why?

Process and Thread Ownership

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>Stack</td>
</tr>
<tr>
<td>Pending alarms</td>
<td>State</td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

Thread Creation

- POSIX standard API for multi-threaded programming
- A thread can be created by `pthread_create` call
- `#include <pthread.h>`
- `int pthread_create (pthread_t *tid, 
   pthread_attr_t *attr, 
   void *(*func) (void*), 
   void *arg)`
  - `tid` – thread ID
  - `attr` – priority, initial stack, etc. `NULL` signifies default
  - `func` – function to call to start thread
  - `arg` – arguments to pass to `func`

Sample Code

```c
typedef struct {int i, int o} PAIR;

rlogind (int ri, int ro, int li, int lo) {
    pthread_t in_th, out_th;
    PAIR incoming={ri,lo}, outgoing={li,ro};
    pthread_create(&in_th,0,in,incoming);
    pthread_create(&out_th,0,out,outgoing);
}
```

If main thread terminates, memory for in and out structures may disappear, and spawned threads may access incorrect memory locations.

If the process containing the main thread terminates, then all threads are automatically terminated, leaving their jobs unfinished.

```
pthread_join
```

- Suspends calling thread to waits for a target thread to terminate
- `int pthread_join(pthread_t tid, 
   void **status)`
- `status`, if not `NULL`, returns a pointer to the `void *` returned by the thread when it terminated
Ensuring main thread waits...

typedef struct {int i, int o} PAIR;
rlqind ( int ri, int ro, int li, int lo) {
  pthread_t in_th, out_th;
  PAIR incoming={ri,lo}, outgoing={li,ro};
  pthread_create(&in_th,0, in, &incoming);
  pthread_create(&out_th,0, out, &outgoing);
  pthread_join(in_th,0);
  pthread_join(out_th,0);
}

Thread Termination

- A thread can terminate
  1. by calling void pthread_exit(void *status), or
  2. By returning from the initial routine (the one specified at the time of creation)
- Termination of a thread unblocks any other thread that’s waiting using pthread_join
- Termination of a process terminates all its threads

Implementing Threads in User Space

User-Level Threads

- The run-time support system for threads is entirely in user space.
- The threads run on top of a run-time system, which is a collection of procedures that manage threads.
- As far as the OS is concerned, it is a single (threaded) process.
- Threads can be implemented on an OS that does not support threads.
- Each process can have its own customized scheduling algorithm.

Implementing Threads in the Kernel

Kernel-supported Threads

- No run-time system is needed.
- For each process, the kernel has a table with one entry per thread, for thread's registers, state, priority, and other information.
- All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure.
- When a thread blocks, the kernel can run either another thread from the same process, or a thread from a different process.
User-level vs. Kernel-supported Threads
- If OS does not support threads, a library package in user space can do threads management
- What are the trade-offs for user-level vs kernel-level threads?
- Assume:
  - Process A has one thread and Process B has 100 threads.
  - Scheduler allocates the time slices equally

User-level vs. Kernel-supported Threads
- User-level Thread:
  - A thread in process A runs 100 times as fast as a thread in process B.
  - One blocking system call blocks all threads in process B.
- Kernel-supported Threads:
  - Process B receives 100 times the CPU time than process A.
  - Switching among the thread is more time-consuming because the kernel must do the switch.
  - Process B could have 100 system calls in operation concurrently.

Hybrid Implementations
Multiple user threads on a kernel thread
Multiplexing user-level threads onto kernel-level threads