Processes and threads

Conceptual model

- Sequential process model
- Process → executing program
- Better think about “things” being executed in parallel rather than sequentially (too complicated)
- Switching back and forth of processes is called **multiprogramming**

Model

Processes

- Should not be designed with timing issues in mind since:
  - We don’t know when a context switch occurs
- Special actions need to be taken when timing is important

Process vs. Program

- Program: the instructions to be executed
  - The program is “unique”
- Process: the actual execution
  - There might be multiple instances (processes) of the same program

Process creation

- System initialization (boot time)
- Creation (by sys call) by a running process
- A user request (shell)
- Initiation of a batch job (or scheduled job)
Interactive vs. background

- Background processes
  - TSR (old DOS terminology)
  - Daemons (UNIX)
  - Services (Windows)
- Batch systems
  - When the system decides that there are enough resources it might start a new job. Users submit (possibly remotely) jobs to the system

Process creation/termination

- Creation
  - Unix: *fork()* → exact copy of the caller
  - Win32: CreateProcess() → a brand new one
- Termination
  - Voluntary: normal vs. error exit *exit()*
  - Involuntary: fatal error vs. killed *TerminateProcess()*

Process hierarchy

- Root → init
- a → login process
- c, d → shells
- b → background process

Wait for process termination

- Synchronous: *waitpid()*
- Asynchronous: SIGCHLD

Process states

- Running (using the CPU)
- Ready (runnable)
- Blocked (temporarily stopped, waiting)

Scheduler

```
0 1 ... N-2 N-1
```

```
scheduler
```
Processes

- Associated with each process:
  - Address space (program + data + stack)
  - Entry into the process table (a list of processes)
  - Set of registers (e.g., PC, PSW, etc.)
  - MMU status, registers
- Processes can be created, terminated, signaled (SW interrupt)
- They form a tree (a hierarchy) on some systems
- Process cooperation is obtained by means of IPC (inter-process communication) mechanisms
- Processes start with the privileges of the user who starts them

Implementation

- Process table
- Scheduler is called when particular events occur (I/O interrupts, blocking calls, timers, etc.)

Threads

- Two concepts:
  - Shared resources: signal handlers, open files, memory, etc.
  - Thread of execution: PC, stack, etc.
- Decoupling the two concept:
  - Process: the container of the shared resources
  - Thread: the execution

Multiple threads

- Lightweight processes
- Multithreading

Threads (cntd.)

- Threads share the same address space
- No protection between thread
- A thread has a state (running, blocked, ready)
- A thread of execution is scheduled by the scheduler (depending on the implementation)

Needless to say...

<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></td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
Exemplar thread calls

- `thread_create()`
- `thread_exit()`
- `thread_wait()`
  - Similar to `waitpid()`
- `thread_yield()`
  - Important, since there’s no clock interrupt

Why?

- Simpler programming model:
  - If we need multiple quasi-parallel activities then it’s better to provide a mechanism to support them
  - Background activity within an application
- Efficiency:
  - Keep the CPU busy
  - Multi-processor architectures

The web server

Many possibilities

- **Threads**: parallelism, blocking sys calls
- **Single-threaded process**: No parallelism, blocking sys calls
- **Finite state machine**: Parallelism, non-blocking sys calls (interrupt handling!)

Thread implementation

- User space
- Kernel space
- Hybrid

User space

- Each process maintain a thread table
- Threads are implemented by implementing library calls (user code, not kernel code)
- Efficient since there’s no kernel trap to call the thread code
- Switching can be easy (thread switching)
- The kernel knows nothing of threads
Issues

- How do we implement blocking sys calls?
  - Change libraries: messy
  - Use select() to see if a prospective call would block, requires a “wrapper” to the library
- Page fault:
  - What should a thread do while waiting for a chunk of memory from disk?
- How do we switch from thread to thread?
  - User space threads do not have a timer clock

Kernel space threads

- Since the kernel knows everything about the system it can easily take care of managing threads
- Creating/destroying threads has a cost: a system call
  - Thread recycling in the kernel
- The kernel scheduler, schedules threads instead of whole processes

Making code multithreaded

- Access to global variables:
  - Thread local storage (TLS), library calls
  - Example: the errno variable
- Reentrant library calls:
  - The possibility of having a second call made while a previous call has not yet finished
  - E.g. malloc (maintains lists of memory chunks)
- Who should catch unspecific interrupts?
- Stack growth: how do we handle it?