Processes & Threads
Managing Concurrency in Computer Systems

Process Management

- Process management deals with several issues:
  - what are the units of execution
  - how are those units of execution represented in the OS
  - how is work scheduled in the CPU
  - what are possible execution states, and how does the system move from one to another

The Process

- Basic idea is the process:
  - process is the unit of execution
  - it’s the unit of scheduling
  - it’s the dynamic (active) execution context (as opposed to a program, which is static)
- A process is sometimes called a job or a task or a sequential process.
- A sequential process is a program in execution; it defines the sequential, instruction-at-a-time execution of a program.

What’s in a Process?

- A process consists of at least:
  - the code for the running program
  - the data for the running program
  - an execution stack tracing the state of procedure calls made
  - the Program Counter, indicating the next instruction
  - a set of general-purpose registers with current values
  - a set of operating system resources (open files, connections to other programs, etc.)
- The process contains all the state for a program in execution.
Process State

• There may be several processes running the same program (e.g., an editor), but each is a distinct process with its own representation.
• Each process has an execution state that indicates what it is currently doing, e.g.,:
  – ready: waiting to be assigned to the CPU
  – running: executing instructions on the CPU
  – waiting: waiting for an event, e.g., I/O completion
• As a program executes, it moves from state to state

Process State Changing

Processes move from state to state as a result of actions they perform (e.g., system calls), OS actions (rescheduling), and external actions (interrupts)

Process Data Structures

• At any time, there are many processes in the system, each in its particular state.
• The OS must have data structures representing each process: this data structure is called the PCB:
  – Process Control Block
• The PCB contains all of the info about a process.
• The PCB is where the OS keeps all of a process' hardware execution state (PC, SP, registers) when the process is not running.

PCB

The PCB contains the entire state of the process

- process state
- process number
- program counter
- stack pointer
- general-purpose registers
- memory management info
- memory map of heap
- process pointers for state queues
- scheduling info (priority, etc.)
- accounting info
PCBs and Hardware State

- When a process is running its Program Counter, stack pointer, registers, etc., are loaded on the CPU (i.e., the processor hardware registers contain the current values).
- When the OS stops running a process, it saves the current values of those registers into the PCB for that process.
- When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process’ PCB.
- The process of switching the CPU from one process to another is called a context switch. Timesharing systems may do 100s or 1000s of context switches a second!

State Queues

- The OS maintains a collection of queues that represent the state of all processes in the system.
- There is typically one queue for each state, e.g., ready, waiting for I/O, etc.
- Each PCB is queued onto a state queue according to its current state.
  - As a process changes state, its PCB is unlinked from one queue and linked onto another.

PCBs and State Queues

- PCBs are data structures, dynamically allocated in OS memory.
- When a process is created, a PCB is allocated to it, initialized, and placed on the correct queue.
- As the process computes, its PCB moves from queue to queue.
- When the process is terminated, its PCB is deallocated.
Cooperating Processes

- Processes can be independent or they can be cooperating to accomplish a single job.
- Cooperating processes can be used:
  - to gain speedup by overlapping activities or performing work in parallel
  - to better structure an application as a small set of cooperating processes
  - to share information between jobs
- Sometimes processes are structured as a pipeline where each produces work for the next stage that consumes it, and so on.

Processes and Threads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a “thread of control”, which defines where the process is currently executing (basically, the PC and registers)
- Creating a new process is costly, because of all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly, because most communication goes through the OS

Parallel Programs

- Suppose I want to build a parallel program to execute on a multiprocessor, or a web server to handle multiple simultaneous web requests. I need to:
  - create several processes that can execute in parallel
  - cause each to map to the same address space (because they’re part of the same computation)
  - give each its starting address and initial parameters
  - the OS will then schedule these processes, in parallel, on the various processors
- Notice that there’s a lot of cost in creating these processes and possibly coordinating them. There’s also a lot of duplication, because they all share the same address space, protection, etc……

“Lightweight” Processes

- What’s similar in these processes?
  - They all share the same code and data (address space)
  - they all share the same privileges
  - they share almost everything in the process
- What don’t they share?
  - Each has its own PC, registers, and stack pointer
- Idea: why don’t we separate the idea of process (address space, accounting, etc.) from that of the minimal “thread of control” (PC, SP, registers)?
Threads and Processes

- Some newer operating systems (Mach, Chorus, NT) therefore support two entities:
  - the process, which defines the address space and general process attributes
  - the thread, which defines a sequential execution stream within a process
- A thread is bound to a single process. For each process, however, there may be many threads.
- Threads are the unit of scheduling; processes are containers in which threads execute.

How different OSs support threads

<table>
<thead>
<tr>
<th>address space</th>
<th>thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>example: MS/DOS</td>
<td><img src="example.png" alt="Thread symbol" /></td>
</tr>
<tr>
<td>example: Unix</td>
<td><img src="example.png" alt="Thread symbol" /></td>
</tr>
<tr>
<td>example: Xerox PDP</td>
<td><img src="example.png" alt="Thread symbol" /></td>
</tr>
<tr>
<td>example: Mach, OSF, Chorus, NT</td>
<td><img src="example.png" alt="Thread symbol" /></td>
</tr>
</tbody>
</table>

Separation of Threads and Processes

- Separating threads and processes makes it easier to support multi-threaded applications
- Concurrency (multithreading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
- So, multithreading is useful even on a uniprocessor.
- But, threads, even when supported separately from processes in the kernel, are too slow.

Kernel Threads

- Kernel threads still suffer from performance problems.
- Operations on kernel threads are slow because:
  - a thread operation still requires a kernel call
  - kernel threads may be overly general, in order to support needs of different users, languages, etc.
  - the kernel doesn’t trust the user, so there must be lots of checking on kernel calls
User-Level Threads

- To make threads really fast, they should be implemented at the user level.
- A user-level thread is managed entirely by the run-time system (user-level code that is linked with your program).
- Each thread is represented simply by a PC, registers, stack and a little control block, managed in the user’s address space.
- Creating a new thread, switching between threads, and synchronizing between threads can all be done without kernel involvement.

(Old) Example of thread performance

<table>
<thead>
<tr>
<th></th>
<th>Ultrix</th>
<th>Topaz</th>
<th>FastThreads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fork</td>
<td>1320</td>
<td>1208</td>
<td>39</td>
</tr>
<tr>
<td>Signal/Wait</td>
<td>1846</td>
<td>229</td>
<td>52</td>
</tr>
</tbody>
</table>

(performance on a 3 MIPS processor, in microseconds)

- Ultrix: 1 thread per address space
- Topaz: multiple threads per address space
- FastThreads: multiple user threads per address space

Example U-L Thread Interface

```c
int t = thread_fork(initial context)
int thread_stop()
int thread_start(t)
int thread_yield()
int thread_exit()
```

- `thread_fork`: create a new thread of control
- `thread_stop`: stop the calling thread, sometimes called thread_block
- `thread_start(t)`: start the named thread
- `thread_yield()`: voluntarily give up the processor
- `thread_exit()`: terminate the calling thread, sometimes called thread_destroy.