Threads and Concurrency

B. Ramamurthy
Thread

- Unit of work

- A process has address space, registers, PC and stack (See Section 3, page 9.. in A Roadmap through Nachos for the detailed list)

- A thread has registers, program counter and stack, but the address space is shared with process that started it.
  - This means that a user level thread could be invoked without assistance from the OS. This low overhead is one of the main advantages of threads.
  - If a thread of a process is blocked, the process could go on.
  - Concurrency: Many threads could be operating concurrently, on a multi threaded kernel.
  - User level scheduling is simplified and realistic (bound, unbound, set concurrency, priorities etc.)
  - Communication among the threads is easy and can be carried out without OS intervention.
Thread requirements

- An execution state
- Independent PC working within the same process.
- An execution stack.
- Per-thread static storage for local variables.
- Access to memory and resources of the creator-process shared with all other threads in the task.
- Key benefits: less time to create than creating a new process, less time to switch, less time to terminate, more intuitive for implementing concurrency if the application is a collection of execution units.
Examples of thread usage

- Foreground and background work: Consider spreadsheet program: one thread could display menu and get response while another could be processing the request. Increases the perceived speed of the application.

- Asynchronous processing: Periodic backup (auto-saving) of RAM into disk. A thread could schedule itself to come-alive every 1 minute or so to do this saving concurrently with main processing.

- Speed execution: In hard-core data-processing simple concurrency can speed up process.

- Transaction processing: Many independent transactions can be modeled very nicely using threads. Such applications as neural networks, searches, graphics, agent/actor model suit well for thread-implementation.
Threads and Processes

- A thread is a unit of work to a CPU. It is strand of control flow.
- A traditional UNIX process has a single thread that has sole possession of the process’s memory and resources.
- Threads within a process are scheduled and execute independently.
- Many threads may share the same address space.
- Each thread has its own private attributes: stack, program counter and register context.
Threads and Processes

- One process
  - One thread

- One process
  - Multiple threads

- Multiple processes
  - One thread per process

- Multiple processes
  - Multiple threads per process
Thread Operations

- **Basic Operations associated with a thread are:**
  - **Spawn**: newly created into the ready state
  - **Block**: waiting for an event
  - **Unblock**: moved into ready from blocked
  - **Finish**: exit after normal or abnormal termination.
Nachos Threads (Section 3, RoadMap)

- Nachos process has an address space, a single thread of control, and other objects such as open file descriptors.
- Global variables are shared among threads. Ex: buffer of a mailbox you use in Lab1.
- Nachos “scheduler” maintains a data structure called ready list which keeps track of threads that are ready to execute.
- Each thread has an associated state: READY, RUNNING, BLOCKED, JUST_CREATED
- Global variable currentThread always points to the currently running thread.
Nachos Thread Description and Control

- **Thread specific data:** local data, stack and registers such as PC, SP.
- **Control:**
  - Thread creation (Ex: fork)
  - Thread schedule (Ex: yield)
  - Thread synchronization (Ex: using barrier)
  - Code for execution (Ex: fork’s function parameter)
Nachos Thread Class

| Thread
| //operations
| Thread *Thread(char *name);
| Thread *Thread(char *name, int priority);
| Fork (VoidFunctionPtr func, int arg);
| void Yield(); // Scheduler::FindNextToRun()
| void Sleep(); // change state to BLOCKED
| void Finish(); // cleanup
| void CheckOverflow(); // stack overflow
| void setStatus(ThreadStatus st); // ready, running.. // blocked
| char* getName();
| void Print(); // print name
| //data
| int* stackTop;
| int machineState[MachineStateSize]; // registers
| int* stack;
| ThreadStatus status;
| char* name;
Thread Control and Scheduling

- `Switch (oldThread, newThread);`  
  // assembly language routine
- Threads that are ready kept in a ready list.
- The scheduler decides which thread to run next.
- Nachos Scheduling policy is: FIFO.
Nachos Scheduler Class

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler()</td>
</tr>
<tr>
<td>~Scheduler()</td>
</tr>
<tr>
<td>void ReadyTnRun(Thread* thread);</td>
</tr>
<tr>
<td>Thread* FindNextToRun();</td>
</tr>
<tr>
<td>void Run(Thread* nextThread);</td>
</tr>
<tr>
<td>void Print();</td>
</tr>
</tbody>
</table>

List* readyList;