Began to explore the implementation of the UNIX process model

- The user API is very simple:
  - **fork()** creates a new process
  - **exit()** terminates a running process
  - **wait()**, **waitpid()** reap terminated (“zombie”) child processes
  - **execve()** loads and runs a program in a process
  - **kill()** sends a signal to another process

- The implementation is *much* more involved

Implementation of process model can vary widely

- We will cover the major themes
**LAST TIME: UNIX PROCESS STATES**

- Only one process can be Running per CPU (core)
  - Processes waiting for the CPU are Ready
  - Processes waiting for slow resources are Blocked
  - Can also stop (Suspend) and resume processes

![Diagram of process states]

- Running
  - Scheduled for execution
  - Request resource
  - Time-slice expired

- Ready
  - Resume
  - Stop

- Ready_Suspended
  - Resume
  - Stop

- Blocked
  - Resume
  - Stop

- Blocked_Suspended
  - Resume
  - Stop

- Terminated
  - Exit

- Resource available
  - Resource available

- Exit
**States and Processes**

- At application level, UNIX processes are either Running, Stopped, or Terminated...
- At implementation level, only one process may be in Running state for each CPU in the computer
  - e.g. 4 cores = 4 processes in Running state
- Many processes can be in the other states!
- Kernel needs to manage collections of processes in each state
  - Different strategies for managing these processes, so different kinds of collections are employed
Process Control Block

- Each process has a Process Control Block (PCB) associated with it
  - Contains all information necessary for managing the process, and for performing context-switches

- The PCB can contain a lot of information:
  - Process ID, parent and child process IDs
  - When not running, register and memory state of process
  - Information about resources the process is using
  - Pending resource-requests that need to be filled
  - Scheduling information
  - etc.

- All necessary to allow kernel to coordinate processes using different resources on a single physical system
Each process control block in the system contains information like this:

<table>
<thead>
<tr>
<th>ID</th>
<th>IDType</th>
<th>Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU State</td>
<td>StateType</td>
<td>State Vector</td>
</tr>
<tr>
<td>Processor</td>
<td>Int</td>
<td>Page Table Flags ...</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td>Unit Flags ...</td>
</tr>
<tr>
<td>Resources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td>StatusType</td>
<td>Status Info</td>
</tr>
<tr>
<td>Status Data</td>
<td></td>
<td>Running, Ready, Blocked To ready-list for process</td>
</tr>
<tr>
<td>Parent</td>
<td></td>
<td>Parent process</td>
</tr>
<tr>
<td>Children</td>
<td></td>
<td>List of children</td>
</tr>
<tr>
<td>Priority</td>
<td>Int</td>
<td>Other</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PROCESS IDENTIFICATION, HIERARCHY

- The kernel manages a mapping of Process IDs to Process Control Blocks
  - Identification information uniquely identifies the process

- Several options for mapping PIDs to PCBs
  - Linux uses a hashtable, with bins containing linked-lists of PCBs
  - Rationale:
    - More space-efficient than a table where PIDs are indexes
    - Expect that process-count will typically be much smaller than the system limits

- UNIX process model also includes parent and child processes
**Process State Vector**

- State vector specifies all process context information
  
<table>
<thead>
<tr>
<th>CPU State</th>
<th>StateType</th>
<th>Processor</th>
<th>Memory</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td>Int</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Int</td>
<td></td>
<td>Int</td>
<td></td>
<td>Int</td>
</tr>
<tr>
<td>Int</td>
<td></td>
<td>Int</td>
<td></td>
<td>Int</td>
</tr>
</tbody>
</table>

- **CPU state:**
  - Process capabilities and protection info
  - When suspended, also includes PC and register contents
  - Depends on processor architecture

- **Processor:**
  - Set to CPU number when running; otherwise undefined

- **Memory:**
  - Contents of process’ code, data, stack, etc.
  - (Heavily leverages virtual memory system)

- **Resources:**
  - All allocated resources (files, network sockets, etc.)
  - Resource class + unit descriptions
Process Status Information

- Process control block also includes current status

<table>
<thead>
<tr>
<th>Status</th>
<th>StatusType</th>
<th>Running, Ready, Blocked</th>
<th>Status Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Data</td>
<td></td>
<td></td>
<td>To ready-list for process</td>
</tr>
</tbody>
</table>

- **Running:**
  - Process is currently running on a CPU

- **Ready:**
  - Process is ready to run, but waiting for a CPU

- **Blocked:**
  - Process cannot proceed until it receives a resource or a message

- Also includes other states, e.g. Suspended, etc.

- Status data can be used for:
  - Specifying pending resource-requests for this process
  - Specifying other processes in the same state and priority
PROCESS MANAGEMENT

Must provide several operations to support this model
- **Create()**: create and initialize a new process
- **Destroy(p)**: remove a process from the system
- **Suspend(p)**: change process state to Suspended
- **Resume(p)**: change process state to Ready

Another important question:
- *If we destroy or suspend a process, which process in the Ready state should actually run next?*

A scheduler decides which process should run next
- Given information about process behaviors and priorities, scheduler picks a process to resume, and then resumes it
- When we need a new process to run, invoke the scheduler: it will choose a process, and then resume it
- *(Will talk about how the scheduler makes this decision momentarily...)*
**CREATE OPERATION**

- Steps for the `Create()` operation:
  - Allocate a new Process Control Block
  - Assign a Process ID to this PCB
  - Initialize `p->CPU_State`
    - Set initial register values
    - Set Program Counter to the starting address of the process
    - UNIX: `fork()` creates a process, so use Program Counter from invoker as initial Program Counter for new process
      - `fork()` is called once but returns twice
  - Initialize all resources
    - UNIX: child process inherits all resources of parent process
  - Initialize other process accounting
    - e.g. `p->Priority = Normal`, etc.
  - Set process status to Ready; add to Ready collection!
Overview of Suspend operation:
- If the process is currently running, need to stop it
  - Record the process’ context into its Process Control Block
- Change the process status to Suspended
  - Specifically, Ready_Suspended or Blocked_Suspended

Pseudocode for `Suspend(p)`:
```c
if (p->Status == Running) {
    /* Process was running on a CPU. */
    Stop(p); /* Record context of p into its PCB. */
    p->Status = Ready_Suspended;
    schedule(); /* Choose a Ready process to run. */
}
else if (p->Status == Ready) {
    /* Process wasn’t actually running, so no need to */
    p->Status = Ready_Suspended;
    /* stop it, or to invoke the */
} else if (p->Status == Blocked) {
    /* stop it, or to invoke the */
    p->Status = Blocked_Suspended; /* scheduler. */
}
```
STOPPING A PROCESS

- **Stop(p)** operation is generally simple
  - Records entire process context into p->CPU_State
- Make it a separate operation to handle multi-processor systems more easily
  - Also, other operations can use Stop()
- If computer has multiple processors:
  - Kernel process may be executing on one CPU, while process p is running on another CPU
  - Cause an interrupt on the CPU running p, so that the process can be interrupted and suspended
- If the computer has a single processor:
  - The kernel code is already running! ☺
  - Don’t need to interrupt p; simply record its context
RESUMING SUSPENDED PROCESSES

- Suspended processes are not scheduled for execution, until they are resumed
  - Only processes in the Ready state may be scheduled for execution

- Overview of the Resume operation:
  - Process is in either the Ready_Suspended state, or the Blocked_Suspended state
  - If in Ready_Suspended state, change to Ready, then invoke scheduler to start running the process
  - If in Blocked_Suspended state, change to Blocked
    - Can’t schedule this process, so don’t invoke the scheduler
**RESUME OPERATION**

- Pseudocode for **Resume(p)**:
  
  ```cpp
  if (p->Status == Ready_Suspended) {
    p->Status = Ready;
    schedule();
  }
  else {
    // Process is Blocked_Suspended */
    p->Status = Blocked;
  }
  ```
**Destroy Operation**

- Destroy may be called on a running process
  - e.g. process calls `exit()`, or receives a signal that terminates the process
  - In this case, a new process needs to be scheduled for the CPU
- If Destroy is given a suspended process, not necessary to schedule a new process
- Process also holds a number of resources when it is terminated
  - Destroy operation needs to release these resources
**Destroy Operation (2)**

- Pseudocode for `Destroy(p)`:
  ```c
  bool sched = false;
  if (p->Status == Running) {
    Stop(p);
    sched = true;
  }
  ...
  /* Properly handle child processes. */
  for each resource r the process holds:
    release_resource(r);
  free(p);
  if (sched) /* A CPU is available, so */
    schedule(); /* schedule another process. */
  ```
OTHER OPERATIONS

- This is a very high-level overview!
  - Many details left out of the description
  - e.g. moving a process from Blocked_Suspended to Ready_Suspended
    - Updating state is easy; managing hardware resources, requests, and interrupts is definitely not!

- The other important issue:
  - How to manage processes that are Ready to execute?
  - How to choose the process to execute next?

- The scheduler is responsible for this task
  - Given a set of Ready processes, choose a specific process to start running on the CPU
**Scheduling Example**

- You are running:
  - *emacs* to implement your *sthreads* package
  - *gcc* to compile your *sthreads* package
  - VLC to watch a video on your computer
  - A program searching for the next Mersenne Prime

- Process scheduling considerations?
  - When you type on *emacs*, it should respond quickly
  - *gcc* and Mersenne Prime program shouldn’t mess up your video player
  - Prime number program will run much longer than *gcc*, so it shouldn’t impede *gcc* progress
SCHEDULING CONSIDERATIONS

- Processes vary **widely** in their behavior! 😐
- Compute-intensive processes:
  - Execute for long periods of time
  - Use the CPU heavily; typically not blocked on IO
  - e.g. compilers, database servers
- Interactive processes:
  - Constantly waiting for user input (i.e. blocked on IO)
  - Usually not running on the CPU...
  - ...but when input comes, need to respond quickly!
  - e.g. text editors, web browsers
- Real-time processes:
  - Require relatively small, but **very** regular, time on the CPU
  - Typically deadline-driven scheduling requirements
  - e.g. audio/video players
Scheduling Considerations (2)

- The scheduler should also be **fair**
  - All processes should eventually receive time on CPU
  - (Actual time given to each process may vary…)

- The scheduler should also be **fast**
  - Time spent in scheduler takes away from running processes…
  - Need to come up with a good answer, and quickly
  - Context-switches take approx. 5-10µs
    (roughly time of several thousand instructions)
**What Do We Know?**

- How can scheduler know what a process needs?
  - How often will the process block on IO operations?
  - How quickly will IO requests be satisfied?
    - e.g. BitTorrent download vs. you writing your hum paper
  - Does a process have real-time requirements?
  - Does the user expect that some programs will run for a long time?

- We cannot know with certainty:
  - How long before a program blocks on IO
  - How long a program will take to complete
  - How long a human being will take to respond to the program!
What Can We Guess?

- The scheduler can observe some aspects of process behavior
  - Does a process block regularly?
  - Does a process get preempted regularly by the kernel, because it does a lot of computation but no IO?
- Assumption: the future will be like the past
  - A process that regularly blocks on input, will continue to do so
  - A process that blocks on IO but regularly receives data, will continue to do so
  - A process that regularly consumes large amounts of CPU cycles without blocking, will continue to do so
- Scheduler uses a simple model to track behavior
  - Must be inexpensive to update and to use
What Must We Be Told?

- If a process has real-time constraints, the only real solution is to inform the kernel
  - e.g. Linux `sched_setscheduler()` function allows a process to control how it is scheduled by the kernel

- Similarly, if a user doesn’t expect a program to finish quickly, they can alter its priority
  - Programs can use `getpriority()`, `setpriority()` to adjust the relative priority of a process
  - The `nice` utility uses these functions to run a process at a lower priority
  - e.g. don’t want Mersenne Prime program to slow down `gcc` compilation!
ROUND-ROBIN SCHEDULING

A simple scheduling approach:
- Choose a fixed length time-slice, e.g. 100ms
- Scheduler cycles through all processes, giving each one a turn to execute
- If a process blocks or terminates, scheduler immediately goes on to next process

Benefits:
- Very simple scheduling algorithm
- Completely fair – no process is starved
ROUND-ROBIN SCHEDULING (2)

- **Compute-intensive tasks:**
  - Benefit greatly from this approach! Large, regular time-slices with infrequent context-switches.

- **Interactive tasks:**
  - When a process blocks on IO, it is removed from the ready queue
    - When it becomes ready again, it is added to end of queue
  - If a process blocks on IO regularly, it will be forced to wait for long-running processes to use their full time-slice

- Can vary size of time-slice given to each process
  - Simple way to implement priorities
Interactive Tasks

- With round-robin, interactive tasks are delayed by long-running processes
  - Text editor or web browser becomes sluggish and unresponsive

- Another approach:
  - Use past behavior of each process to estimate the run-time until it blocks

- Scheduler runs shortest jobs first
  - Higher priority than long-running tasks
  - Interactive processes are no longer delayed by long-running tasks
With interactive processes, we care about response time
  • Time between data IO (e.g. key-press) and first response from the process

Example:
  • Four jobs, with estimated length 1, 2, 4, 8

Scheduling shortest jobs first:

- Response times are 0, 1, 3, 7. Avg = 2.75, max = 7.

Conversely, scheduling longest jobs first:

- Response times are 0, 8, 12, 14. Avg = 8.5, max = 14.
Shortest Jobs First Scheduling (2)

- What if new jobs show up while current jobs are executing?
- From previous example:
  - Four jobs, with estimated length 1, 2, 4, 8
  - At timestep 5, a new job of length 2 shows up...
- Can schedule the new shortest job next
  - Wait times = 0, 1, 3, 2, 9. Avg = 3, max = 9.
- Problems?
  - If new short jobs keep arriving, long jobs will be starved! This approach is not fair!
SHORTEST JOBS FIRST SCHEDULING (3)

- Can also defer the new shortest job until after previous jobs have all completed.
  - Wait times = 0, 1, 3, 7, 10. Avg = 4.2, max = 10.
- Average and maximum response times get worse, but at least it’s fair.
Round-robin scheduling is good for compute-intensive tasks, but bad for interactive tasks.

Shortest jobs first scheduling good for interactive tasks, but bad for compute-intensive tasks.

Next time:

- How do we schedule for real-time tasks?
- How do we build a generic scheduler that can properly handle various process behaviors?