Administrivia

• PR0 due tomorrow
• PR1 out tomorrow
• Form project group by the end of today
Review

• OS is a resource manager
• OS presents an extended machine

• System calls

• Interrupt

• (Skipping “How to Design an OS” part)
[lec2] Typical Unix OS Structure

Application

Libraries

User level
(run in user mode)

Portable OS Layer

Kernel level
(run in kernel mode)

Machine-dependent layer
## [lec3] System Call

<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create entry for process list</td>
<td></td>
<td>Run main()</td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Load program into memory</td>
<td></td>
<td>Call system call</td>
</tr>
<tr>
<td>Setup user stack with argy</td>
<td></td>
<td>trap into OS</td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td></td>
<td>save regs</td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td>(to kernel stack)</td>
</tr>
<tr>
<td></td>
<td>move to kernel mode</td>
<td>move to user mode</td>
</tr>
<tr>
<td>Handle trap</td>
<td>jump to trap handler</td>
<td>jump to PC after trap</td>
</tr>
<tr>
<td>Do work of syscall</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>return-from-trap</td>
<td></td>
<td>return from main</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trap (via exit())</td>
</tr>
</tbody>
</table>

Free memory of process
Remove from process list

Figure 6.2: Limited Direct Execution Protocol
I/O Completion

• Interrupts are the basis for asynchronous I/O
  ♦ OS initiates I/O
  ♦ Device operates independently of rest of machine
  ♦ Device sends an interrupt signal to CPU when done
  ♦ OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  ♦ CPU looks up kernel address indexed by interrupt number, context switches to routine
Interrupt time line for a single process doing I/O
I/O Example

1. NIC receives packet, writes packet into memory
2. NIC signals a hardware interrupt
3. CPU stops current operation, switches to the kernel mode, saves machine state on the kernel stack
4. CPU reads address from interrupt table indexed by interrupt number, jumps to the address of the interrupt handle (in the NIC driver)
5. NIC device driver processes the packet
6. Upon completion, CPU restores saved state from stack and returns to user mode

Are there any other ways to perform I/O?
Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    » Setting timer is a privileged instruction
  - When timer expires, generates a hardware interrupt
  - Handled by kernel, which controls what runs next
    » Basis for OS scheduler (more later…)
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., sleep)
<table>
<thead>
<tr>
<th>OS @ boot (kernel mode)</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialize trap table</td>
<td>remember addresses of...</td>
</tr>
<tr>
<td></td>
<td>syscall handler</td>
</tr>
<tr>
<td></td>
<td>timer handler</td>
</tr>
<tr>
<td>start interrupt timer</td>
<td>start timer</td>
</tr>
<tr>
<td></td>
<td>interrupt CPU in X ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS @ run (kernel mode)</th>
<th>Hardware</th>
<th>Program (user mode)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Process A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**timer interrupt**
- save regs(A) → k-stack(A)
- move to kernel mode
- jump to trap handler

Handle the trap
- Call `switch()` routine
  - save regs(A) → proc.t(A)
  - restore regs(B) ← proc.t(B)
  - switch to k-stack(B)

**return-from-trap (into B)**
- restore regs(B) ← k-stack(B)
- move to user mode
- jump to B’s PC

Process B
- ...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)
[lec1] What is an OS?

- Resource manager
- Extended (abstract) machine

(Will have a 3rd def based on pragmatics next time)
Modern OSes are interrupt driven

• “An OS is a giant interrupt handler!” (Def 3)
• Once the system is booted, all entry to the kernel occurs as the result of an interrupt
  ♦ Timer interrupt → Context switches in multiprogramming
  ♦ (unexpected) I/O interrupts
  ♦ System calls to switch from user to kernel mode

• At the lowest level an OS is just a bunch of interrupt service routines
  ♦ Each routine simply returns to whatever was executing before it was interrupted
    » A user process, an OS process, another interrupt routine
  ♦ Else infinite wait loop
  ♦ There are, however, some exception: OS background threads
True or False

The transition from user space to kernel space can happen without any hardware assistance/involvement

Is malloc a system call?

Does every keyboard stroke cause an interrupt?
Interrupt Questions

- Interrupts halt the execution of a process and transfer control (execution) to the operating system
  - Can the interrupt handler itself be interrupted? (Consider why there might be different IRQ levels)
  - Can we and shall we disable interrupts?
- Interrupts are used by devices to have the OS do stuff
  - What is an alternative approach to using interrupts?
  - What are the drawbacks of that approach?
Process Management

• This lecture starts a class segment that covers process management (processes, threads, synchronization, and scheduling)
  ♦ These topics are very important but somewhat difficult
  ♦ (and will be covered in the exams)
• Today’s topics are processes and process management
  ♦ What are the units of execution?
  ♦ How are those units of execution represented in the OS?
  ♦ How are those units created?
  ♦ How is work scheduled in the CPU?
  ♦ What are the possible execution states of a process?
  ♦ How does a process move from one state to another?
Users, Programs, Processes

- Users have accounts on the system
- Users launch programs
  - Can many users launch the same program?
  - Can one user launch many instances of the same program?

→ A process is an “instance” of a program
So What Is A Process? (1)

- It’s one instance of a “program”

- Any relationship between two instances?
What Does This Program Do?

```c
int myval;

int main(int argc, char *argv[]) {
    myval = atoi(argv[1]);
    printf("myval is %d, loc 0x%lx\n", myval, (long) &myval);
}
```
Instances of Programs

• The address of the static variable is always the same!
• The values are different!
• Implications:
  ♦ Do instances think they’re using the same address?
  ♦ Are they seeing each other?

• Conclusion: addresses are not absolute!
  ♦ Each process has its own memory address space

• What are the benefits?
  ♦ Compiler/linker/loader do not have to be concerned
  ♦ Allows address space to be bigger than memory
  ♦ More in the second part of the course
So What Is A Process? (2)

• It is one instance of a “program”
• It is separate from other instances
So What Is A Process? (3)

- Process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program
Process Components

- A process contains all state for a program in execution
  - A memory address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    » Open files, network connections, etc.
Program vs. Process

```c
main()
{
    ... 
    foo()
    ... 
}

foo()
{
    ... 
    foo()
    ... 
}
```

Program

```c
main()
{
    ... 
    foo()
    ... 
}

foo()
{
    ... 
    foo()
    ... 
}
```

Process

- Code
- Data
- heap
- stack
- main
- foo
- registers
- PC
Basic Process Address Space

Address Space

0x00000000

0xFFFFFFFF

Stack

Heap (Dynamic Memory Alloc)

Static Data (Data Segment)

Code (Text Segment)

SP

PC
Process ID

- A process is named using its (unique) process ID (PID)
- Does a program know its process ID?
- When a program is running, how does the process know its ID?
Kernel Data Structure for Process

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the Process Control Block (PCB)
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of its hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware
Kernel data structure: Process Control Block

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)

- Memory management info
  - Root of page tables, stats, etc

- I/O and file management
  - Communication ports, directories, file descriptors, etc.

- It is a **heavyweight** abstraction
unsigned in_user_fault;  

unsigned brk_randomized;  

/* disallow userland-initiated cgroup migration */  
unsigned no_cgroup_migration;  
/* task is frozen/stopped (used by the cgroup freeze) */  
unsigned frozen;  

/* to be used once the psi infrastructure lands upstream. */  
unsigned use_memdelay;  

unsigned long atomic_flags; /* Flags requiring atomic access. */  

struct restart_block restart_block;  

pid_t pid;  

pid_t tgid;  

/* Canary value for the -fstack-protector GCC feature: */  
unsigned long stack_canary;  

/* Pointers to the (original) parent process, youngest child, younger sibling,  
* older sibling, respectively. (p->father can be replaced with  
* p->real_parent->pid)  
*/  

/* Real parent process: */  

struct task_struct __rcu *real_parent;  

/* Recipient of SIGCHLD, wait4() reports: */  

struct task_struct __rcu *parent;  

/* Children/sibling form the list of natural children:  
*/  

struct list_head children;  

struct list_head sibling;  

struct task_struct __rcu *group_leader;  

/* 'ptraced' is the list of tasks this task is using ptrace() on.  
* This includes both natural children and PTRACE_ATTACH targets.  
* 'ptrace_entry' is this task's link on the p->parent->ptraced list.  
*/  

struct list_head ptraced;  

struct task_struct __rcu *ptrace_entry;  

/* PID/PID hash table linkage. */  

struct struct_pid_hash [PIDTYPE_MAX];  

struct list_head thread_node;  

struct completion *vfork_done;  

/* CLONE_CHILD_SETTID: */  

int __user *set_child_tid;  

/* CLONE_CHILD_CLEARTID: */  

int __user *clear_child_tid;  

u64 utime;  

u64 stime;  

u64 utimescaled;  

u64 stimescaled;  

/* Context switch counts: */  

unsigned long nvcsw;  

unsigned long nivcsw;  

/* Monotonic time in nsecs: */  

u64 start_time;  

/* Boot based time in nsecs: */  

u64 start_boottime;  

/* MM fault and swap info: this can arguably be seen as either mm-spool  
unsigned long minflt;  

unsigned long majflt;  

/* Empty if CONFIG_POSIX_CPUTIMERS=0 */  

struct posix_cputimers *posix_cputimers;  

/* Process credentials: */  

/* Tracer's credentials at attach: */  

const struct cred __rcu *ptracer_cred;  

/* Objective and real subjective task credentials (CON): */  

const struct cred __rcu *real_cred;  

/* Effective (overrideable) subjective task credentials (CON): */  

const struct cred __rcu *cred;  

/* Cached requested key. */  

struct key *cached_requested_key;  

/* executable name, excluding path.  
* - normally initialized setup_new_exec()  
* - access it with (get_task_comm())  
* - lock it with task_lock()  
*/  

char comm[TASK_COMM_LEN];  

/* nameidata */  

struct sysvsem sysvsem;  

struct sysv_shm sysvshm;  

/* Filesystem information: */  

struct fs_struct *fs;  

/* Open file information: */  

struct files_struct *files;  

/* Namespaces: */  

struct namespace */
struct mutex futex_exit_mutex;  
unsigned int futex_state;  

#endif CONFIG_PERF_EVENTS

struct perf_event_context *perf_event_ctxp[ perf_nr_task_contexts];  
struct mutex perf_event_mutex;  
struct list_head perf_event_list;  

#endif CONFIG_DEBUG_PREEMPT

unsigned long preempt_disable_ip;

#endif CONFIG_NUMA

/* Protected by alloc_lock */

struct mempolicy *mempolicy;
short il1_prev;
short pref_node_fork;

#endif CONFIG_NUMA_BALANCING

int numa_scan_seq;
unsigned int numa_scan_period;
unsigned int numa_scan_period_max;
int numaPreferred_nid;
unsigned long numa_migrate_retry;

/* Migration stamp: */
node_id_t node_id;
int last_task_nid;
unsigned long last_nid_to_migrate;
unsigned long last_task_nid_to_migrate;
unsigned long last_time_nid_migrated;

struct callback_head numa_work;

/*
 * This pointer is only modified for current in syscall and
 * on fault context (and for tasks being destroyed), so it can be read
 * from any of the following contexts:
 * - RCU read-side critical section
 * - current->numa_group from everywhere
 * - task’s runqueue locked, task not running
 */

struct numa_group __rcu *numa_group;

* numa_faults is an array split into four regions:
* faults_memory, faults_cpu, faults_memory_buffer, faults_cpu_buffer
* in this precise order.
* faults_memory: Exponential decaying average of faults on a per-page
* basis. Scheduling placement decisions are made based on these
* counters. The values remain static for the duration of a PTE scan.
* faults_cpu: Track the nodes the process was running on when a NUMA
* hint fault was incurred.
* faults_memory_buffer and faults_cpu_buffer: Record faults per node
* during the current scan window. When the scan completes, the counts
* in faults_memory and faults_cpu decay and these values are copied.
*/

unsigned long *numa_faults;
unsigned long total_numa_faults;

/*
 * numa_faults locality tracks if faults recorded during the last
 * scan window were remote/local or failed to migrate. The task scan
 * period is adapted based on the locality of the faults with different
 * weights depending on whether they were shared or private faults
 */

unsigned long numa_faults_locality[3];
unsigned long numa_pages_migrated;

#endif /* CONFIG_NUMA_BALANCING */

#ifdef CONFIG_RSEQ

struct rseq __user *rseq;
unsigned long rseq_event_mask;

#ifdef CONFIG_KASAN

unsigned int kasan_depth;

#endif CONFIG_FUNCTION_GRAPH_TRACER

/* Index of current stored address in ret_stack */
int curr_ret_stack;
int curr_ret_depth;

/* Stack of return addresses for return function tracing */

#endif /* CONFIG_RSEQ */
670 LOC in total for task_struct in Linux 5.5.10!
Process State

- A process has an *execution state* that indicates what it is currently doing
  - **Running**: Executing instructions on the CPU
    » It is the process that has control of the CPU
    » How many processes can be in the running state simultaneously?
  - **Ready**: Waiting to be assigned to the CPU
    » Ready to execute, but another process is executing on the CPU
  - **Waiting (blocked)**: Waiting for an event, e.g., I/O completion
    » It cannot make progress until event is signaled (disk completes)
Life cycle of a process: Process State Transition

- Create a process
- Ready
- Resource becomes available
- Blocking
- Wait for resource
- Scheduler dispatch
- Running
- Terminate
**Context Switch**

- **Definition:**
  switching the CPU to another process, which involves saving the state of the old process and loading the state of the new process

- What state?
- Where to store them?
Context Switch

Process $P_0$ | Operating System | Process $P_1$

- **Executing**
  - Interrupt or system call
  - Save state into PCB$_0$
  - ... (repeated)
  - Reload state from PCB$_1$

- **Idle**

- **Executing**
  - Interrupt or system call
  - Save state into PCB$_1$
  - ... (repeated)
  - Reload state from PCB$_0$

**Context Switch overhead**
Process State Transition

- As a process executes, it moves from state to state
  - Unix “ps”: STAT/S column indicates execution state
  - What state do you think a process is in most of the time?
### Unix Process States

```
top - 20:48:08 up 275 days, 6:26, 3 users, load average: 0.06, 0.07, 0.05
Tasks: 171 total, 1 running, 170 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.1%us, 0.1%sy, 0.0%ni, 99.8%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 16467276k total, 14159652k used, 2307624k free, 171168k buffers
Swap: 0k total, 0k used, 0k free, 884340k cached

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>14677</td>
<td>voelker</td>
<td>20</td>
<td>0</td>
<td>55548</td>
<td>3232</td>
<td>2364</td>
<td>R</td>
<td>0.0</td>
<td>0.0</td>
<td>0:00:07</td>
<td>top</td>
</tr>
<tr>
<td>24637</td>
<td>voelker</td>
<td>20</td>
<td>0</td>
<td>86300</td>
<td>6364</td>
<td>1024</td>
<td>S</td>
<td>32:06:70</td>
<td>mosh-server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>57812</td>
<td>1636</td>
<td>584</td>
<td>S</td>
<td>1:26:73</td>
<td>init</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:03:13</td>
<td>kthreadd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:04:38</td>
<td>migration/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>9:54:94</td>
<td>ksoftirqd/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:00:01</td>
<td>watchdog/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:04:39</td>
<td>migration/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>11:22:89</td>
<td>ksoftirqd/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:00:01</td>
<td>watchdog/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:18:05</td>
<td>migration/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>9:44:37</td>
<td>ksoftirqd/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:00:01</td>
<td>watchdog/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>18:06</td>
<td>migration/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>9:01:67</td>
<td>ksoftirqd/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0:00:01</td>
<td>watchdog/3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>root</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>2:30:99</td>
<td>events/0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
PCBs and Hardware State

• When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  ♦ The hardware registers contain the current values
• When the OS stops running a process, it saves the current values of the registers into the process’ PCB
• When the OS is ready to start executing a new process, it loads the hardware registers from the values stored in that process’ PCB
  ♦ What happens to the code that is executing?
• The process of changing the CPU hardware state from one process to another is called a context switch
  ♦ This can happen 100 or 1000 times a second!
State Queues

How does the OS keep track of processes?

• The OS maintains a collection of queues that represent the state of all processes in the system.
• Typically, the OS has one queue for each state:
  ♦ Ready, waiting for network, waiting for disk, etc.
• Each PCB is queued on a state queue according to its current state.
• As a process changes state, its PCB is unlinked from one queue and linked into another.
State Queues

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
PCBs and State Queues

- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initializes it, and places it on the ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated
Next time...

- Read Chapters 26, 27
Context Switch in Nachos

Here's an example of how an operating system implements context switching:

- *Threads* instead of *processes*, but ignore that for now.

There are three classes involved:
- Thread: thread abstraction
- Scheduler: ready queue management
- Interrupt: interrupt simulation

... plus an interrupt handler for the interval timer and a machine specific function `SWITCH` for saving and loading a thread's state.
TimerInterruptHandler: {
    interrupt->YieldOnReturn(); }

calls currentThread->Yield() when handler returns

void Thread::Yield () {
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    nextThread = scheduler->FindNextToRun();
    if (nextThread != NULL) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    }
    (void) interrupt->SetLevel(oldLevel);
}
void Scheduler::ReadyToRun (Thread *thread) {
    thread->setStatus(READY);
    readyList->Append((void *)thread);
}

Thread * Scheduler::FindNextToRun () {
    return (Thread *) readyList->Remove();
}

void Scheduler::Run (Thread *nextThread) {
    Thread *oldThread = currentThread;
    currentThread = nextThread;
    currentThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
}
Context Switch in Nachos

This is called *round-robin scheduling*

- A thread runs for a *quantum* (the time it takes for the interval timer to fire)
- How long should a quantum be? Why?
- How long after it finishes with a quantum does a thread run again?
- What happens if there is only one thread (or all the other threads are waiting)?