Administrivia

- Midterm: 10/26 Tue 2-3:20pm
- HW1 and PR0 due on 10/7
- Form project group by 10/7
Review

- OS is a resource manager
- OS presents an extended machine
- OS is a “giant interrupt handler”

- System calls

- Interrupt
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User level (run in user mode)

Kernel level (run in kernel mode)
### 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>restore regs (from kernel stack)</td>
<td>Run main()</td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td>move to user mode</td>
<td>...</td>
</tr>
<tr>
<td>Load program into memory</td>
<td>jump to main</td>
<td>Call system call</td>
</tr>
<tr>
<td>Setup user stack with argv</td>
<td>trap into OS</td>
<td></td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td><strong>Handle trap</strong></td>
<td>Do work of syscall</td>
</tr>
</tbody>
</table>

**return-from-trap**

**save regs (to kernel stack)**

**move to kernel mode**

**jump to trap handler**

**return-from-trap**

**restore regs (from kernel stack)**

**move to user mode**

**jump to PC after trap**

**return from main**

**trap (via exit())**

---

*Figure 6.2: Limited Direct Execution Protocol*
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?
Process Management

• This lecture starts a class segment that covers process management (processes, threads, synchronization, and scheduling)
  ♦ These topics are very important
  ♦ (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
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

• A sequential process is a program in execution
  ♦ It defines the sequential, instruction-at-a-time execution of a program
  ♦ Programs are static entities with the potential for execution
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

Program

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

foo()
{
  ...
}
```

Process

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

foo()
{
  ...
}
```

<table>
<thead>
<tr>
<th>Code</th>
<th>Data</th>
<th>heap</th>
<th>stack</th>
<th>main</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
<th>PC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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?

```
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: 16467276kB total, 14159652kB used, 2307624kB free, 171168kB buffers
Swap: 0kB total, 0kB used, 0kB free, 884340kB 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</td>
<td>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>0</td>
<td>0</td>
<td>32:06:70</td>
<td>mosh-server</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>0</td>
<td>0</td>
<td>1:26:73</td>
<td>init</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</td>
<td>0</td>
<td>0:03:13</td>
<td>kthreadd</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</td>
<td>0</td>
<td>0:04:38</td>
<td>migration/0</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>0</td>
<td>0</td>
<td>9:54:94</td>
<td>ksoftirqd/0</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</td>
<td>0</td>
<td>0:00:01</td>
<td>watchdog/0</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</td>
<td>0</td>
<td>0:04:39</td>
<td>migration/1</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>0</td>
<td>0</td>
<td>11:22:89</td>
<td>ksoftirqd/1</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</td>
<td>0</td>
<td>0:00:01</td>
<td>watchdog/1</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</td>
<td>0</td>
<td>0:18:05</td>
<td>migration/2</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>0</td>
<td>0</td>
<td>9:44:37</td>
<td>ksoftirqd/2</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</td>
<td>0</td>
<td>0:00:01</td>
<td>watchdog/2</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>0</td>
<td>0</td>
<td>0:18:06</td>
<td>migration/3</td>
</tr>
<tr>
<td>13</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>9:01:67</td>
<td>ksoftirqd/3</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</td>
<td>0</td>
<td>0:00:01</td>
<td>watchdog/3</td>
</tr>
<tr>
<td>15</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0</td>
<td>2:30:99</td>
<td>events/0</td>
</tr>
</tbody>
</table>
```
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
  - Segments, page table, stats, etc

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

- It is a **heavyweight** abstraction
struct task_struct {
  #ifdef CONFIG_THREAD_INFO_IN_TASK
  /*
   * For reasons of header soup (see current_thread_info()), this
   * must be the first element of task_struct.
   */
  thread_info thread_info;
  #endif

  /* -1 unrunnable, 0 runnable, >0 stopped: */
  volatile long state;

  /* This begins the randomizable portion of task_struct. Only
   * scheduling-critical items should be added above here.
   */
  randomized_struct_fields_start

  void *stack;
  refcount_t usage;
  /* Per task flags (PF_*) defined further below: */
  unsigned int flags;
  unsigned int ptrace;

  #ifdef CONFIG_SMP
  struct list_node wake_entry;
  int on_cpu;
  #ifdef CONFIG_THREAD_INFO_IN_TASK
  /* Current CPU: */
  unsigned int cpu;
  #endif
  unsigned int wakee_flips;
  unsigned long wakee_flip_decay_ts;
  struct task_struct *last_wakee;
  #endif

  /*
   * recent_used_cpu is initially set as the last CPU used by a task
   * that wakes away another task. Waker/wakee relationships can
   * push tasks around a CPU where each wakeup moves to the next one.
   * Tracking a recently used CPU allows a quick search for a recently
   * used CPU that may be idle.
   */
  int recent_used_cpu;
  int wake_cpu;

  #endif
  int on_rq;
  int prio;
  int static_prio;
  int normal_prio;
  unsigned int rt_priority;

  const struct sched_class *sched_class;
  struct sched_entity *rcu;
  struct sched_rt_entity rt;

  #ifdef CONFIG_CGROUP_SCHED
  struct task_group *sched_task_group;
  #endif

  struct sched_dl_entity dl;

  #ifdef CONFIG_NUMA
  /* Clamp values requested for a scheduling entity */
  struct uclamp_se uclamp_req[UCLAMP_CNT];
  /* Effective clamp values used for a scheduling entity */
  struct uclamp_se uclamp[UCLAMP_CNT];
  #endif

  #ifdef CONFIG_PREEMPT_NOTIFIERS
  /* List of struct preempt_notifier: */
  struct list_head preempt_notifiers;
  #endif

  #ifdef CONFIG_BLK_DEV_IO_TRACE
  btrace_seq;
  </nowiska>
unsigned in_user_fault;  
#undef  
#define CONFIG_COMPAT_BRK  
unsigned brk_randomized;  
#undef  
#define CONFIG_CGROUPS  
/* disallow userland-initiated cgroup migration */  
unsigned no_cgroup_migration;  
/* task is frozen/stopped (used by the cgroup freezer) */  
unsigned frozen;  
#undef  
#define CONFIG_BLK_CGROUP  
/* to be used once the psi infrastructure lands upstream. */  
unsigned use_mempdelay;  
#undef  
#define CONFIG_STACKPROTECTOR  
/* Canary value for the -fstack-protector GCC feature: */  
unsigned long stack_canary;  
#undef  
/*  
* 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 *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 list_head ptrace_entry;  
/* PID/PID hash table linkage. */  
struct pid __user *thread_pid;  
struct list_head pid_links;  
struct list_head thread_group;  
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;  
unsigned long u64 utime;  
unsigned long stime;  
#undef  
#define CONFIG_VIRT_CPU_ACCOUNTING_GEN  
struct prev_cputime cpuacct_pcpu;  
struct prev_cputime cpuacct_pcpu;  
#undef  
#define CONFIG_NO_HZ_FULL  
atomic_t tickdep_mask;  
#undef  
/* Context switch counts: */  
unsigned long nvcsw;  
unsigned long nivcsw;  
/* Monotonic time in nssecs: */  
u64 start_time;  
/* Boot based time in nssecs: */  
u64 start_boottime;  
/* MM fault and swap info: this can arguably be seen as either mm-spinlocks or  
* mm_containers, but the former is more consistent with our spinlock naming  
* conventions. */  
unsigned long min_flt;  
unsigned long maj_flt;  
/* Empty if CONFIG_POSIX_CPUTIMERS=n */  
struct posix_cputimers *posix_cputimers;  
/* Process credentials: */  
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;  
#undef  
#define CONFIG_KEYS  
/* Cached requested key. */  
struct key *cached_requested_key;  
#undef  
/*  
* 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];  
struct nameidata *nameidata;  
#undef  
#define CONFIG_SYSVIEW  
struct sysv_sema sysvsem;  
struct sysv_shm sysvshm;  
#undef  
#define CONFIG_DETECT_HUNG_TASK  
unsigned long last_switch_count;  
unsigned long last_switch_time;  
#undef  
/* Filesystem information: */  
struct fs_struct *fs;  
/* Open file information: */  
struct files_struct *files;  
/* Namespaces: */  
struct nsproxy *nsproxy;
```c
// Signal handlers: */
struct signal_struct {signal;
struct sigmask_struct {signal;
struct sigset_t* saved_sigmask;
struct sigset_t* sigact_mask;
/* Restored if set_restore_signal() was used: */
struct sigset_t* sigignore_mask;
struct sigset_t sigpending;
unsigned long sas_ss_sp;
unsigned int sas_ss_flags;
}
struct callback_head {callback_head;
}
#endif
#ifdef CONFIG_AUDIT
 struct audit_context {
audit_context;
}
#endif
#define kuid_t loginuxid;
#undef kuid_t
#define u32
#define u64
#define u32
#define u64
#endif
/* Thread group tracking: */
parent_exec_id;
self_exec_id;
#endif
/* Protection against (de-)allocation mm, files, fs, tty, keyrings, mems_allowed, memspinlock_t alloc_lock;
#endif
/* Protection of the PI data structures: */
raw_spinlock_t pi_lock;
#endif
#endif
#endif
#endif
#endif
#define TASK콱_node wake_q:
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
```

/* Protected by alloc_lock: */

short mempolicy;
short mempolicy;

/* Migration stamp: */

u64 nodestamp;

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-node

* basis. Scheduling placement decisions are made based on these

* counts. The values remain stable for the duration of a PTE scan.

* faults_cpu: Track the nodes the process was running on when a NUNA

* hinting 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.

* numa_faults is a global statistic.

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

* numa_pages_migrated;

unsigned long numa_pages_migrated;

#endif /* CONFIG_NUMA_BALANCING */

#endif /* CONFIG_KASAN */

#endif /* CONFIG_FUNCTION_GRAPH_TRACER */

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

struct ftrace_ret_stack *ret_stack;

/ * Timestamp for last schedule: */

unsigned long ftrace_timestamp;

/ * Number of functions that haven't been traced

* because of depth overrun:

* atomic_t trace_overrun;

/ * Pause tracing: */

atomic_t trace_graph_pause;

#endif /* CONFIG_TRACING */

/ * State flags for use by tracers: */

/ */

/ */ Cache last used pipe for splice(): */

/ */
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

- **Ready**
  - Create a process
  - Scheduler dispatch
  - Resource becomes available

- **Running**
  - Scheduler dispatch
  - Wait for resource
  - terminate

- **Blocked**
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?
  - How many processes can a system support?
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</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>0.0</td>
<td>32:06.70</td>
<td>mosh-server</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>0.0</td>
<td>1:26.73</td>
<td>init</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>0.0</td>
<td>0:03.13</td>
<td>kthreadd</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>0.0</td>
<td>0:04.38</td>
<td>migration/0</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>0.0</td>
<td>9:54.94</td>
<td>ksoftirqd/0</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>0.0</td>
<td>0:00.01</td>
<td>watchdog/0</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>0.0</td>
<td>0:04.39</td>
<td>migration/1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>11:22.89</td>
<td>ksoftirqd/1</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>0.0</td>
<td>0:00.01</td>
<td>watchdog/1</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>0.0</td>
<td>0:18.05</td>
<td>migration/2</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>0.0</td>
<td>9:44.37</td>
<td>ksoftirqd/2</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>0.0</td>
<td>0:00.01</td>
<td>watchdog/2</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>0.0</td>
<td>0:18.06</td>
<td>migration/3</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>9:01.67</td>
<td>ksoftirqd/3</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>0.0</td>
<td>0:00.01</td>
<td>watchdog/3</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>root</td>
<td>RT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0.0</td>
<td>2:30.99</td>
<td>events/0</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

- Ready Queue
- Firefox PCB
- X Server PCB
- Emacs PCB
- Disk I/O Queue
- Emacs PCB
- ls PCB

Console Queue
Sleep Queue

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
Process Creation

• A process is created by another process
  ♦ Parent is creator, child is created ( Unix: ps “PPID” field)

• The parent defines (or donates) resources and privileges for its children
  ♦ Unix: Process User ID is inherited – children of your shell execute with your privileges
Process Creation

- Compiling a program:
  - Who created gcc? (What does gcc internally consist of?)
  - Who created process cpp?
  - Who created cc1?

- Who created shell?

- Who creates the first process (Unix: init (PID 0 or 1))?
Process Tree Example

Figure 3.9 A tree of processes on a typical Solaris system.
Process Creation/Termination

• Who should *actually* create/terminate processes? (bring program to life)
  ♦ Who manages processes?

• But user process decides when and what

• So what should OS provide user process?

• What should the semantics of the interface be?
Process Creation – interface design options

• The system call creates a child process

• Execution possibilities?
  ♦ Parent and child execute concurrently
  ♦ Parent waits till child finishes

• Address space possibilities?
  ♦ Child duplicate of parent (code and data)
  ♦ Child has a new program loaded into it
Process Creation — UNIX interfaces (1)

- `fork()` system call creates a duplicate of the original process

```
main()
{
  ...
  foo()
  ...
  I = fork()
}
foo()
{
  ...
}

Process
```

```
main()
{
  ...
  foo()
  ...
  I = fork()
}
foo()
{
  ...
}

Process
```
Process Creation -- UNIX interfaces (2)

- `fork()` system call creates a duplicate of the original process
  - What is the major benefit?
  - How to disambiguate who is who?

```c
main()
{
...  
foo()
...
I = fork()
}

foo()
{
...
}
```

Process

```c
main()
{
...
foo()
...
I = fork()
}

foo()
{
...
}
```

Process
Process Creation: Unix

- fork()
  - Creates and initializes a new PCB
  - Creates a new address space (but with duplicate content)
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue

- Fork returns twice
  - Huh?
  - Returns the child’s PID to the parent, “0” to the child
What does this program print?
Example Output

alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}

Parent

Child

child_pid = 0
child_pid = 0
child_pid = 486
Divergence

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```
Example Continued

alpenglow (18) ~/tmp> cc t.c
alpenglow (19) ~/tmp> a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> a.out
Child of a.out is 498
My child is 498

Why is the output in a different order?
**Why fork()?**

- Very useful when the child...
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task

- Example: Web server

```c
while (1) {
    int sock = accept();
    if (((child_pid = fork()) == 0) {  // Handle client request and exit
        Handle client request and exit
    } else {  // Close socket
        Close socket
    }
}
```
Process Creation: Unix interfaces (2)

• Wait a second. How do we actually start a new program?

    \[
    \text{int exec(char *prog, char *argv[])}
    \]

• `exec()` system call used after a `fork` to replace the process’ code/address space with a new program
  ♦ Important: BOTH code and data, i.e., the whole address space is replaced!
exec("b.out")

Afterwards, only one thing about the process was kept, which is?
**Process Creation: Unix interfaces (2)**

- `exec()`
  - Stops the current process
  - Loads the program “prog” into the process’ address space
  - Initializes hardware context and `args` for the new program
  - Places the PCB onto the ready queue
  - **Note:** It *does not* create a new process

- What does it mean for `exec` to return?
Process Creation – UNIX interfaces

UNIX system calls related to process creation/termination:

- **fork** – create a copy of this process
  - Clone would have been a better name!
- **exec** – replace this process with a new program
- **fork()** is used to create a new process, **exec** is used to load a program into the address space
- **exit/kill** – (potentially) end a running process
- **wait** – wait for child process to finish
Process Termination

• All good processes must come to an end. But how?
  ♦ Unix: `exit(int status)`

• Essentially, free resources and terminate
  ♦ Terminate all threads (next lecture)
  ♦ Close open files, network connections
  ♦ Release allocated memory (and swapped out pages)
  ♦ Remove PCB from kernel data structures, delete

• Note that a process does not need to clean up itself
  ♦ Why does the OS have to do it?
What happens when a parent process disappears? all child processes are killed by the OS, or all child processes reset init as their parent
wait() a second...

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Unix `wait()`
  - Suspends the current process until any child process ends
  - `waitpid()` suspends until the specified child process ends
More exercises

```c
#include <stdio.h>
void main()
{
    int pid;        int was = 3;
    pid = fork();  /* fork another process */

    if (pid == 0) { /* child process */
        sleep(2);  printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);}
    else { /* pid > 0; parent process */
        was = 4;
        printf("parent: child process id = %d; was=%d\n", pid, was);
        wait(NULL); exit(0);
    }
}
```
#include <stdio.h>

void main()
{
    int ret_of_fork;  int was = 3;
    ret_of_fork = fork(); /* fork another process */

    if (ret_of_fork == 0) { /* child process */
        sleep(2); was = 9; printf("child: was = %d\n", was);
        execlp("/bin/ls", "ls", NULL);  was = 10;
        printf("It's me, your child was = %d\n", was);
    }
    else { /* pid > 0; parent process */
        was = 4;
        printf("parent: child process id %d was=%d\n", pid, was);
        wait(NULL);  exit(0);
    }
}
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes,
        redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}
Process Summary

• What are the units of execution?
  ♦ Processes

• How are those units of execution represented?
  ♦ Process Control Blocks (PCBs)

• How is work scheduled in the CPU?
  ♦ Process states, process queues, context switches

• What are the possible execution states of a process?
  ♦ Running, ready, waiting

• How does a process move from one state to another?
  ♦ Scheduling, I/O, creation, termination

• How are processes created?
  ♦ fork/exec
Next time...

- Read Chapters 26, 27
Backup Slides
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.
Context Switch in Nachos

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);
}

disables interrupts. Why do we do this?

NULL if the ready queue is empty

re-enables interrupts. When does this execute?
Context Switch in Nachos

```c
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);
}
```

what happens when this executes?
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)?