Part 2
Processes and Threads

This presentation contains materials distributed with
• Stallings, Operating Systems, 7th ed., Prentice-Hall

Relevant Reading Assignments
• These notes cover Chapter 3 (3.1-3.7) of Stallings
• For fun, see Torvalds vs. Tanenbaum (1992)
  (link on 410 Resources page)
• NOTE: Early Linux code available at:
  – http://www.cse.msu.edu/~mckinley/410/linux/

Process Topics
• Process model and process states
• Process control block (example: Linux task_struct)
• System calls, reentrant kernel
• Interrupts, signals and their interaction
• Suspend/resume (sleep/wakeup)
• Address space, page tables
• Process creation/termination (fork/exec)
  • Scheduling and dispatching (covered as separate topic)
  • Open file table (we will discuss this later with file systems)
  • Loading and linking (discuss later with boot and virtualization)

Linux Examples
• To see how a real OS implements this functionality, we show example code from an early Linux release.
• Full source code for this version of the Linux kernel is available here:
  http://www.cse.msu.edu/~mckinley/410/linux/
• This version is also described in Rusling’s online “text” on an early version of the Linux kernel, at:
  http://tldp.org/LDP/tlk/tlk.html

OS Requirements for Processes
• All multiprogramming operating systems are built around the concept of processes
• OS supports
  – creation/termination of processes
  – interleaving (i.e., timeslicing) the execution of multiple processes to maximize CPU efficiency and “virtualize” CPU
• For each process, the OS
  – constructs/maintains an address space
  – provides access to system services via system calls
  – supports interprocess communication
  – is alerted (exception) if the process behaves “improperly”

Multiprogramming (multitasking/timeslicing)
(a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Compared to earlier batch systems, multiprogramming virtualizes the CPU for each user process.
Each process thinks it has the machine to itself.
OS Execution

- Three main types of OS have been studied/developed
  - (a) OS kernel is a process itself
    - Context switch upon system call
  - (b) OS executes within context of current user process
    - System call trap switches mode, but only some aspects of context
  - (c) OS is largely a set of user-level processes
    - name for this design?

Preemptive Kernel

- Typically means process executing in kernel can involuntarily lose control of processor to another user process (not just interrupt handlers)
- Unix/Linux kernel historically was not preemptive
- Linux 2.6 kernel and beyond can be configured to be preemptive.
- For general-purpose computing, non-preemption is generally fine.
- For what type of systems is preemption needed?

Critical Sections

- Unlike protecting against interrupts, a multithreaded kernel requires a more general mechanism for locking and unlocking critical sections of code
- Kernels rely on support from hardware for atomic operations (e.g., test-and-set, SWP, CAS, etc.)
- True or false: A multithreaded kernel could work correctly without such support.

Types of processes

- User processes
  - run in user mode when executing user code
  - run in kernel modes for system calls, traps, and interrupts
- Kernel processes (*not* just a user process in kernel mode)
  - Sometimes called system processes or kernel daemons
  - Just like other processes, except never run in user mode
  - Stay in kernel mode, wake up occasionally to perform OS housekeeping functions, e.g.,
    - management of free space
    - writing dirty (modified) pages to disk
    - etc.

Typical User Process Components

- Compiled application code (e.g., code you wrote)
- Library functions (including system call stubs)
  - Either statically compiled into program
  - Or dynamically linked during process creation or run time
- Global data
  - Usually private to a process
  - Sometimes shared with other processes
- Stack to support execution of the code
  - Activation records pushed and popped during function call and return, respectively
  - Store local variables, return address, processor context

Typical User Process Components (cont.)

- Kernel data structures specific to the process
  - Most important of these: process control block (PCB)
    - contains (or points to) “everything there is to know” about that process
    - PCB is allocated and filled in when the process is created
- Also, kernel buffers containing data related to this process (e.g., network packets, blocks of files)
- On many systems, the ENTIRE kernel address space is part of the address space of every user process (accessible only when executing in kernel mode)
Processes

All operating systems have a structure to maintain information on each process

Example information:

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment info</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment info</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment info</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Manipulation of PCBs (important!)

- Array of pointers to PCBs maintained in kernel
- When a process is created, a new PCB is allocated
- An PCB will reside on various lists during its lifetime
  - Typically exist several different “blocked” lists
  - Waiting for disk
  - Waiting for network packet
  - Waiting for action by another process…

Typical (General Purpose) OS Design

- In most modern operating systems, most OS code gets executed in the context of a user process.
  - What do we mean by context?
  - Exceptions to this statement?
- On system calls and other traps: the CPU switches to kernel mode to execute OS routine within the context of user process (mode switch)
- Before returning to user level, the OS might relinquish the processor to a different process.
  - E.g., the first process blocks, exits, or its time slice expires
  - The scheduler code selects next process, the dispatcher switches the contexts, including PC/PSW

Illustration

- But, if the process is blocked, who executes the scheduler?
- A process “blocks” by changing its state to BLOCKED, after which it relinquishes the processor.
- What if a process reaches the end of its time slice?

Reentrant Kernel

- Reentrant code: process (thread) executing the code can safely yield the processor to another
- A reentrant kernel is written so that processes can give up control (voluntarily or involuntarily) while executing in the kernel
- If kernel is not reentrant, process can be blocked only in user mode
  - If executing in the kernel, must “back out” to user level
- Hardware interrupts have priority over duties such as servicing system calls.
- Consequence?

Thread of Execution

- And another important component of a process…
- One or more threads that execute the code
  - But what is a thread???
  - Anything tangible?
Evolution of PCB in Unix

- Historically, Unix had **two** kernel structures for every process:
  - **u area** (open file table, real/effective UIDs, kernel stack, etc.)
  - **proc structure** (process state, mmgt info, pointer to u area, etc.)
- In modern systems, there is only one (e.g., Linux **task_struct**) containing elements of both.
- Why were there two?

Process States

- Once a process has been created, it sequences through a set of states, for example, it can be:
  - Ready to run (sitting in a run queue)
  - Running (actually executing on a processor)
  - Blocked (waiting for a resource, before it can run)
    - Examples?
  - Exiting (in Linux this is called ZOMBIE)

Process States (simplified)

- Scheduler selects process to run
- Once running, process holds the processor until:
  - Time slice expires, exits, faults, blocks (waiting on IO)
- Actually, invoking any system call will likely relinquish control...

Example: Linux Process States

Linux task_struct

- In early operating systems, a static array of PCBs was allocated at boot time
- Modern systems typically declare an array of pointers, with actual space allocated on demand
- Each OS has a different name for its PCBs
- In Linux, the process control block is called a **task_struct**
Pointers to task_structs

- A given task_struct (actually a pointer to it) typically appears on MANY lists and other data structures
  - all processes (task list)
  - parents/children/siblings
  - run queue
  - blocked/waiting lists
  - etc.

task_struct contents

- But, operating systems are not made of boxes!
- For fun (?), let's take a look at the code for the task_struct in that early version of Linux.
- Don't worry, you are not responsible for details of the code, but rather the concepts/mechanisms realized by the code.

Linux task_struct  
(include/linux/sched.h)

```
struct task_struct {
   ... fields ...
   ... various fields ...
   struct linux_list next_task, prev_task;
   struct task_struct *next, *next_run;
   struct task_struct *parent;
   ... variables ...
   ... fields ...
   ... variables ...
}
```
Some key fields

• volatile long state;
  – State of the process (running, blocked, etc)
  – volatile keyword?
• int pid;
  – process identifier (shows up with ps command)
• unsigned short uid...
  unsigned short gid...
  – user and group ids
• char comm[16];
  – Name of the file containing the process code (command)
  – E.g., ”gcc” ”g++” ”emacs”

More fields

• long counter;
  – Keeps track of how much time is left in timeslice
  – E.g, initialized to 20 (jiffies), decremented with clock tick
• unsigned long kernel_stack_page;
  – Address of kernel stack (only a single page)
  – For 4KB pages, lower bits of address are 0
• int errno;
  – stores error value associated with failed system call
  – copied into user address space upon return
  #include <errno.h>
  …
  if (somecall() == -1) {
    printf("somecall() failed, error code is %d\n", errno);
  }

Lists of task_structs

• struct task_struct *next_task, *prev_task;
  – list of all existing processes
  – how used?
• struct task_struct *next_run, *prev_run;
  – these links realize the run queue
  – list of tasks ready to run (i.e., not blocked)
  – scheduler will look through this list and identify process
    with highest priority (early algorithm)
  – that process will be dispatched onto the processor

Address Space

• struct mm_struct *mm;
  – Points to structures describing vm_areas
    • Start and end addresses, permissions etc.
    • Idea of sparse virtual address space
    • not originally in Unix, borrowed from Mach. More later.
  – Pages of VM areas are mapped to physical pages (page frames) by page tables. Details shortly.

Signals

• unsigned long signal;
  – Supports a very important type of interprocess communication (in Unix since the beginning)
  – Each of 32 bits acts as flag indicating important information
  – We say a signal is posted against a process
  – If running at user level*, process will be interrupted
  – At various points during kernel execution, the process
    checks these flags and responds accordingly
• Example:
  – Bit 9, SIGKILL: if posted, will terminate the program
  – Bit 2, SIGINT: interrupt from keyboard, can be caught
  – Bit 14, SIGALRM: process-initiated timer has fired

File system information

• struct fs_struct *fs;
  – Every process has the concept of executing within a file system (current working directory)
  – fs_struct defines root and current working directory
Open files

- `struct files_struct *files;`
- A process may have many files open at one time
- This data structure points to any array, where each entry points to another structure describing the open file
  - Type (regular file, directory, socket, pipe, etc.)
  - Location on secondary storage
  - State (to what point has it been read/written)
  - And so on

Process Activities of the OS

- Process creation
  - In Unix processes are created through system call `fork()` or derivatives such as `vfork()`, `clone()`.
  - In all versions of Unix, certain processes created at boot time (daemons) have special significance. We will discuss these later in the context of Linux.
- Process termination through `exit()`.
- Process suspension and resumption
  - Processes suspend themselves (sleep) when they are awaiting the occurrence of some event.
    - Sleeping processes do not consume CPU resources
    - Traditional Unix: `sleep(addr)` and `wakeup(addr)`
    - Linux: Wait queues (include/linux/wait.h)

Process Creation

- Events that cause process creation:
  - System boot
  - One process spawns another (e.g., `fork()` system call)
    - User creates a new process (e.g., typing a command).
- Basic steps of process creation (e.g., `fork()`)
  - Allocate and initialize kernel data structures
  - Configure address space (text, data, stack)
  - Update kernel data structures for shared resources such as files
  - Add new process to scheduler queue
- We will look at the details of `fork()` shortly…

Process Termination

- Events that cause process termination:
  - Normal exit (voluntary).
  - Error exit (voluntary).
  - Fatal error (involuntary).
    - Examples?
    - What happens?
  - Killed by another process (involuntary).
    - Who can do it?
    - How does it work?

User Level vs. Kernel Level

- Processes “trap” into the kernel by executing a specific instruction (or due to an exception)
- For a monolithic kernel, once a process has entered the kernel, it has access not only to all information about itself, but about all other processes and system resources.

  - Again, how do we know the process won’t do something it shouldn’t, while executing in the kernel?

System Calls

- User-level process traps to kernel
  - Invokes library routine (linked to its address space)
  - Library routine contains trap instruction (software interrupt)
  - Kernel “interrupt handler” runs, identifies which system call was invoked (how?), and calls a corresponding routine by indexing into the system call table
- System call routine runs
  - might voluntarily relinquish control
  - in any case, scheduler is invoked before process returns to user level.
Types of System Calls (hundreds)

- **Process control**: creation, termination, synchronization, communication, manage resources including memory, threads, events
- **I/O**: file open/close, reading, writing, management of peripheral devices, network operations
- **Status**: time/date info, get/set system data, devices or system attributes
- **Protection**: get/set permissions for resources

For a list of all Linux system calls, see: [http://man7.org/linux/man-pages/man2/syscall.2.html](http://man7.org/linux/man-pages/man2/syscall.2.html)

(Early) Linux System Call Implementation

- System call implementation is architecture-specific
- i386 Implementation in earlier Linux
  - execution of a system call is invoked by a programmed interrupt, caused by the instruction "int 0x80"
  - interrupt vectors set up at boot time to transfer control to specific addresses in the kernel
  - see arch/*/kernel/entry.S for syscall, exception entry points
- each system call has a "stub" in libc library
- each stub is a syscallX() macro, where X is the number of parameters used by the actual routine

Example...

- This system call stub implements all system calls that take one parameter

  ```c
  #define _syscall1(type, name, type1, arg1)
  \{
  _asm volatile ("int $0x80"
  : "a"(_res)
  : "0"(_NR_##name), "b"((long)(arg1)));
  if (_res >= 0)
    return (type) __res;
  errno = -__res;
  return -1;
  }
  ```

Entry to the kernel

- See arch/*/kernel/entry.S, which defines entry points for interrupts/exceptions set up a boot (segmentation error, divide by zero, system call, ...) 
- ENTRY(system call): this code is responsible for saving all registers, checking to make sure a valid system call was invoked and then ultimately transferring control to the actual system call code via the offsets in the system call table
- Invokes system call via system call table
- ret_from_sys_call() checks to see if the scheduler should be run, and if so, calls it

Entering the Kernel (earlier than 2.0.33)
System Call Table

- Table in kernel space
- Each entry is a pointer to the function implementing a particular system call
- Index of entry placed in a register prior to the trap instruction

Unix/Linux Signals

- Very early form of interprocess communication
- Extremely simple
  - In fact, as simple as you can get: One bit!
- Uses
  - inform a process of an asynchronous event (e.g., interrupt)
  - may be sent by other user processes using kill() system call
  - may be sent by the kernel to a user process
- Signals are NOT interrupts!
  However, some signals are triggered by interrupts.

List of Signals (may be variations among OS versions)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Default Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Term</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Core</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Core</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>Core</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Core</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>Core</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Core</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>Term</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Term</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Term</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Term</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>Core</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>Core</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>Core</td>
</tr>
</tbody>
</table>

System Call Table

- data
- ENTRY(sys_call_table)
  - long SYMBOL_NAME(sys_setup) /* 0 */
  - long SYMBOL_NAME(sys_exit)   /* 1 */
  - long SYMBOL_NAME(sys_fork)   /* 2 */
  - long SYMBOL_NAME(sys_open)   /* 3 */
  - long SYMBOL_NAME(sys_write)  /* 4 */
  - long SYMBOL_NAME(sys_ioctl)  /* 5 */
  - long SYMBOL_NAME(sys_close)  /* 6 */
  - long SYMBOL_NAME(sys_unlink) /* 7 */
  - long SYMBOL_NAME(sys_read)   /* 8 */
  - long SYMBOL_NAME(sys_mknod)  /* 9 */
  - long SYMBOL_NAME(sys_clone)  /* 10 */
  - long SYMBOL_NAME(sys_waitpid)/
  - long SYMBOL_NAME(sys_setup)  /* 11 */
  - long SYMBOL_NAME(sys_execve) /* 12 */
  - long SYMBOL_NAME(sys_setup)  /* 13 */
  - long SYMBOL_NAME(sys_setup)  /* 14 */
  - long SYMBOL_NAME(sys_setup)  /* 15 */

Signal Features

- Classes of signals
  - indicating termination of a process
  - process induced exceptions
  - unrecoverable conditions during system call
  - unexpected error during system call
  - user mode signals and terminal-related signals
  - tracing-related signals
- Signals are “posted” against a process by kernel
- Signals are “received” by a process running in the kernel

Signal Default Actions

Default actions
- Term: Default action is to terminate the process.
- Ign: Default action is to ignore the signal.
- Core: Default action is to terminate the process and dump core.
- Stop: Default action is to stop the process.
- Cont: Default action is to continue the process if it is currently stopped.

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Note: “core dump” is an historical term referring to storing the entire main memory when a program faults. Nowadays it refers to storing the image of a process that has faulted. By default is working directory of the process or a standard system directory (e.g., /cores on Mac OS X).
Kernel and Signals

- Sending a signal to a process
  - Simply, set a bit in signal field of PCB (e.g., task_struct)
  - if process asleep and interruptible, kernel wakes it up
  - processes have no memory of how many times a signal fired
- Kernel handling of signals
  - kernel code checks for received signals when the process is ready to return to user mode
  - note: signal has no instant effect on process running in kernel mode. Why?
  - if process is running in user mode, it is interrupted; signal takes effect when returning from the interrupt

User process sending a signal

- kill(pid, signum) system call
  - pid - identifies set of processes to receive the signal
  - signum - signal being sent
  - pid can specify a process or a process group
- raise(signum) library routine
  - simply posts signal against current process
  - actually invokes kill() system call
  - could be invoked if a particular condition is detected...
- In the kernel (kernel/exit.c)
  - simply post the signal (set appropriate bit) against the process or each member of a process group
  - typically will wake up process if sleeping

Response to signal

- Possible actions taken by process receiving signal
  - exit in kernel mode (default for most signals)
  - ignore the signal (can do this only for certain signals)
  - “catch” a signal (execute a user-defined “handler”). This is only allowed for certain signals.
- User-defined signal handlers enable a program to contain asynchronous logic:
  - Regardless of what the process was doing at the time, execute a particular function in response to an event such as Ctrl-C, timer firing, floating point exception, ...

sigaction() System Call

- int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
- void (*signal(int signum, void (*handler)(int)))(int);
- signum is the number of the signal
- act is a user-defined function to be called upon receipt of the signal, or special flag:
  - if handler is SIG_IGN, the signal will be ignored if it is posted against the process
  - if handler is SIG_DFL, default action for the signal is reinstated.
  - If act points to user function, that function will run asynchronously when signal is processed by the kernel
- task_struct field (signal_struct) keeps track of how each signal should be handled:
  ```c
  struct signal_struct {
    int count;
    struct sigaction action[32]
  }
  ```

Simple Example (catch ctrl-C)

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>

void sig_handler(int signo) {
    if (signo == SIGINT) {
        printf("received SIGINT\n");
    }
}

int main(void) {
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        printf("can't catch SIGINT\n");
    }
    else {
        // go into a sleep loop
        while(1) sleep(1);
    }
    return 0;
}
```
So, how does asynchronous code run?

- Remember: kernel has complete control over a user process.
- Basically, the kernel “gaslights” the target process
  - manipulates user stack
  - changes user context, including program counter
  - starts process executing in signal-handling function
  - when finished, process returns to where it had been
- Resumed Process: “Hmm, I could have sworn I was executing in function foo() when I last had the processor.”
- Kernel: “No, no. You were here in sighand(). Are you feeling ok? Maybe you should lie down.”

To execute user-defined handler...

- Kernel activities (called from do_signal in signal.c)
  - access the saved register context to get PC and SP
  - set signal handler field to default state
  - create new stack frame on user stack, so it “looks like” user program directly called the signal handler
  - change PC and SP in saved register context so they reflect execution in the new function
  - when kernel returns to user mode, the process will
    - execute the signal handling function
    - when finished, return to place in code where interrupted

Illustration

- Process X was “executing” in function foo() when a signal was posted against it
- Note: X may not have been actually running at the time!
  (on a uniprocessor, impossible)
- Function sighand() had been designated to catch this signal

Hardware Interrupts

- Of course, the other way to “pull the rug out from under” a process (or even the kernel!) is a hardware interrupt.
- Some examples:
  - I/O - disk controller, keyboard, printer, network card
  - Clock (timer) interrupt
  - Hardware failure
  - Inter-processor interrupt (on multiprocessor machine)
- Execution transfers to “pre-wired” interrupt handlers (a.k.a., interrupt service routines) executing on a special stack called the interrupt stack

Interrupt Handlers

- Interrupts have priorities
- During interrupt handling, lower priority interrupts are blocked
- Upon return from interrupt handler, any pending interrupts will fire (in order) and be handled
- In many cases, the hardware interrupt (which has to be handled promptly) indicates some processing needs to be done
  - But that processing (e.g., handling a network packet) might be lower priority than what kernel was doing already
  - We delay other “half” of interrupt handling till time is right...

Bottom Half Handling

- Handle the tasks associated with an interrupt when the time is right
- Often, this means right before returning to user level
- In Linux, “bottom half handling” originally was static:
  - Set of flags, similar to signal flags, only kernel-wide
  - Flags are set by “top half” of interrupt handlers
  - Another array has pointers to the BH handler routines
- Recent versions use more flexible methods (tasklets)
- In Windows, this is called deferred procedure calls
  - A separate interrupt level is reserved for DPC/dispatch
  - Handled at same interrupt level as in Linux
Interrupts and Signals

- **Reminder**: Signals are **not** interrupts!
- Signals are just **flags** in the PCB that are checked and result in a function call to a signal handler
  - either in a kernel handler or,
  - by stack manipulation, a user-level handler
- In contrast, for interrupt handlers
  - Addresses stored in interrupt vector (set up during boot)
  - Execution triggered directly by interrupt (soft or hard)
- **However**, some signals are certainly tied to interrupts
  - SIGINT, SIGFPE, SIGILL, SIGALRM
  - Interrupt handler (or bhh) posts signal, handled later

Timesharing (a.k.a., timeslicing)

- Clock (timer) interrupt fires periodically (in Linux, jiffy)
- Typical jiffy time is still relatively long (e.g., 1ms)
- Typically, each clock tick the clock interrupt handler
  - decrements the counter variable of current process
  - if 0, set global flag to invoke scheduler before returning to user level
  - also checks various interval timers and sets bh flags as necessary (e.g., SIGALRM fires)

Sleeping and Waking Up

- Besides **involuntary** relinquishing of processor through time slicing...
- Since beginning of Unix (and most other general purpose OS) has been ability of a process to **voluntarily** relinquish control of processor
  - User-level primitive (such as waiting for an event)
    - We will study these later (concurrency, semaphores, mutexes, etc.)
  - Kernel-level blocking (unknown to user level program)
    - E.g., waiting for incoming network data, disk read, etc.
    - Note: real-time OS may not allow this...

Wait Queues

- Kernel data used in suspending and resuming processes
- Process “sleeps” by putting itself on a wait queue
- At some other point in time, the kernel removes the process from the wait queue and adds it to the run queue
- Process can then be scheduled to execute
- Wait queues and their usage are scattered throughout the Linux kernel.

A few (of many) example uses in the kernel

```c
#include <linux/sched.h>

static void __sleep_on(struct wait_queue **p, int state) { ...
    unsigned long flags;
    struct wait_queue wait = { current, NULL }; ...

    if (current == task[0]) ...
        return;

    if (!p) ...
        panic("task[0] trying to sleep");
    current->state = state;
    save_flags(flags);
    cli();
    __add_wait_queue(p, &wait);
    schedule();
    cli();
    __remove_wait_queue(p, &wait);
    restore_flags(flags);
```

Sleeping on a wait_queue

- current process adds itself, calls scheduler
  - Note: does not stop/block here!
Wake Up!

* wake_up doesn't wake up stopped processes - they have to be awakened with signals or similar.
* Note that this doesn't need cli-cli pairs: interrupts may not change the wait-queue structures directly, but only call wake_up() to wake a process. The process itself must remove the queue once it has woken.

```c
void wake_up(struct wait_queue **q) {
  struct wait_queue *next;
  struct wait_queue *head;
  if (!q || !(next = *q))
    return;
  head = WAIT_QUEUE_HEAD(q);
  while (next != head) {
    struct task_struct *p = next->task;
    next = next->next;
    if (p) {
      if (p->state == TASK_UNINTERRUPTIBLE ||
          p->state == TASK_INTERRUPTIBLE)
        wake_up_process(p);
      if (!next)
        goto bad;
    }
  }
  return;
bad:
  printk("wait_queue is bad (eip = %p)\n", __builtin_return_address(0));
  printk("q = %p\n", q);
  printk("       *q = %p\n", *q);
}
```

Wake Up (cont)

• Actually change state of process and put on run queue

• Wake up a process. Put it on the run-queue if it's not already there. The "current" process is always on the run-queue (except when the actual re-schedule is in progress), and as such you're allowed to do the simpler "current->state = TASK_RUNNING" to mark yourself runnable without the overhead of this.

```c
inline void wake_up_process(struct task_struct *p) {
  unsigned long flags;
  save_flags(flags);
  cli();
  p->state = TASK_RUNNING;
  if (!p->next_run)
    add_to_runqueue(p);
  restore_flags(flags);
}
```

Review of control flow (for uniprocessor)

• Continuous flow of execution, from boot, through setup of OS, through (timesliced) execution of processes
• Interrupts and their return “hopping” from one piece of code to another, in some cases switching modes
• Also, processes can explicitly hand control of the processor to other processes
  – steps are carried out by the yielding process
  – in the context of the yielding process

Example

• As a process, when I make a request of the kernel (e.g., reading from a file):
  – I trap to the kernel, and if the data I need is not in memory
  – I issue a request to the disk controller
  – I put my PCB on a wait queue
  – I call the scheduler and select the next process to run
  – I call the routine to switch from me to the next process
• And that is exactly where I wake up when I am scheduled to run again later.

Details of Virtual Memory

• Addresses seen by the process are virtual, not physical.
• Again, the advantages?
  • But, the system (mostly hardware nowadays) needs to translate every memory reference
  • Job of the OS: set up and maintain the address space, handle page faults and other events

Process Address Space

• Three main areas (segments): text, data, stack
• Data segment divided into initialized data and uninitialized data (BSS)
  – BSS does not have to be part of image on disk (other than declaring the size needed)
  – Data segment can grow via brk() system call (which is used in malloc())
• Processes executing the same image will share text segment
  – Processes can also share data segments through special mechanisms, described later
Mapping Address Space to Physical Mem

- Note 1: Unlike figure above, in paged system, pages comprising a segment need not reside in contiguous physical memory
- Note 2: Lots of “empty” virtual address space

VM Details

- Virtual and physical memory divided into pages (4K by default on most systems)
- Page tables map virtual pages to page frames
- Each page table entry defines
  - access permissions (read/write, kernel mode only, etc.)
  - whether the page is currently in main memory (one bit)
  - if so, location of page in main memory (page frame number)
  - if not?
- Bits in the virtual address are indices into the page table

Address Translation with Paging

- Problem with this approach? Solution?

Caching the PTEs

- Will work, but still has a major problem. What?

Multi-level Page Tables

- Key point: bits of virtual address used as indices
- Key point: each page table fits in (exactly) one page

Example: Two-Level Page Tables on x86

- Two levels of indirection in address translation
- Top level is historically called page “directory”
  - contains (up to) pointers to 1024 page tables
  - some (many) entries can be null
- Second level page tables
  - each contains (up to) 1024 pointers to physical pages
  - again, many entries can be null
- When the process runs, the CR3 register loaded with the physical base address of the page directory
- When process is not running, CR3 value stored as part of the process’s context (in task_struct)
Virtual to Physical Address Translation

- 32-bit virtual address is divided as follows:
  - bits 31-22 DIRECTORY; bits 21-12 TABLE; bits 11-0 OFFSET
- Physical address is then computed (in hardware):
  - compute CR3 + DIR*4
    - The top 20 bits of the entry at this address point to second-level page table base
  - Compute page table base + TABLE*4
    - The top 20 bits of the entry at this address point to the base of physical page (page frame) containing referenced word
  - physical address = page frame base + OFFSET

When page is swapped out?

- Bit 0 is 0.
- What is in bits 1-31?

Maximum size of swap space?

Now, consider a 64-bit architecture!

- Addresses are 48 bits long. Hmm. Why?
- What do the page tables look like?

Page Table Summary

- A page table is a data structure in the kernel
- But, not just any data structure!
- Implementation is directly tied to the architecture.
  - address translation details, page size, etc.
- And vice versa!!
  - architecture needs to be able to “walk” the page tables
  - requires a rigorously defined and implemented data structure

Page Table Contents, x86 architecture

- Page table entry
  - each page table entry is 32 bits long
  - only need 20 bits to address a 4KB page
  - upper 20 bits contain address information
  - lower 12 bits are used to store useful information about the page table (or page) pointed to by the entry.

<table>
<thead>
<tr>
<th>31-12</th>
<th>11-9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>OS/3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>U/S R/W P</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- D = 1 means page is dirty (undefined for page directory entry);
- R/W = 0 means read-only for user.
- U/S = 1 means user page.
- P = 1 means page is present in memory.
- A = 1 means page has been accessed (set to 0 by aging).
- GS = bits can be used for LRU etc, and are defined by the OS.

Example
Linux VM Areas

VM area: contiguous piece of address space (one or more pages) to be treated/managed as a unit.

When does the kernel search through this structure?

User and Kernel Space (32-bit arch.)

- Kernel space mapped into each process
- Kernel runs in virtual memory, mostly direct mapped
- Protection bit (U/S) indicates privileged mode only

Standard Linux Process Layout (32-bit)

Spawning a New Process

- In modern operating systems processes are (almost) always created by other processes
- Exception is an initial process (init, in the case of Unix/Linux), which is “manually” constructed by kernel upon boot.

Evolution of fork()

- Traditional fork() system call in **swapped** systems
  - complete copy of data and stack segments created for new process
  - text segment could be shared as read-only
- BSD optimization
  - full copy, as in swapping system, is very wasteful
  - in paging system, just copy page tables and update page frame data table reference counts
  - for data pages, remain shared until written to, at which time copy occurs (copy-on-write)

Simple fork() example

```c
int main()
{
    std::cout << "Current process Id : " << getpid() << std::endl;
    pid_t pid = fork();
    if (pid < 0) {
        std::cout << "Failed to Create a new Process" << std::endl;
    }
    else if (pid == 0) {
        std::cout << "I am the child process.";
        std::cout << " My ID : " << getpid() << std::endl;
        std::cout << " My parent's ID : " << getppid() << std::endl;
    }
    else if (pid > 0) {
        std::cout << "I am the parent process.";
        std::cout << " My ID : " << getpid() << std::endl;
        std::cout << " My child's ID : " << pid << std::endl;
    }
}
```
Linux Process Creation

- **Fork() System Call**
  - `pid = fork();`
  - `pid` - process id of child returned to parent, 0 returned to child
  - child process differs from the parent process only in its PID and PPID; file locks and pending signals are not inherited.
- **A few (of many) fork procedures of kernel**
  - allocate new task struct
  - assign unique id to child
  - get free page for kernel stack
  - make "logical" copy of parent process – share page frames
  - increment file and inode counters for files
  - **copy-on-write**: data is shared until one of the processes attempts to write to it, then a separate copy is allocated to the process.
  - How implemented?

Linux do_fork (kernel/fork.c)

```c
p->pid_exec = 0;
p->nextable = 0;
kernel_stack_page = new_stack;
unsigned long p->kernel_stack_ptr = STACK_MAGIC;
state = TASK_RUNNING;
flags = PF_FORKING;
p->pid = get_pid_table();
p->prev = NULL;
p->next = NULL;
p->p_gdt = p->pgdt = new_stack;
p->g_ssr = NULL;
p->g_tss = NULL;
p->g虮 tolerate current;
p->g_cpu = NULL;
jit_welcome[0] = wait_chريد();
p->signal = 0;
p->fildes = 0;
p->fsf.fix = p->fsf.value = p->fsf.value = 0;
p->fsf.value = 0;
p->fsf.value = 0;
p->fsf.value = 0;
p->Real_term_data = 0;
p->leader = 0;
/task_lnr = 0;
nt_tasks++;
```

Linux clone() system call

- **Synopsis**: `pid clone(void *sp, unsigned long flags)`
- clone is an alternate interface to fork(), with more options for sharing memory, files, stack, etc. with the parent
  - fork() is equivalent to clone(0, SIGCLD—COPYVM).
  - If sp is non-zero, the child process uses sp as its initial stack pointer
  - COPYVM flag: if set, child pages are copy-on-write images of the parent pages, else share with parent
  - COPYFD flag: is set, the child’s file descriptors are copies of the parent’s file descriptors. If not set, the child’s file descriptors are shared with the parent.

execve()

- **execve(2) system call**
  - `int execve (const char *filename, const char *argv[], const char *envp[])`
    - executes the program pointed to by a filename, either a binary executable or a shell script
    - on success, does not return
    - text, data, bss, and stack of the calling process are overwritten by that of the program loaded.
    - new program invoked inherits the calling process’s PID and any open file descriptors that are not set to close on exe
    - signals pending on the parent process are cleared
int main()
{
    char *newargv[] = {"/bin/ls", "/etc/netconfig", NULL};
    std::cout<<"Current process Id : "<<getpid()<<std::endl;
    pid_t pid = fork();

    // check return value
    if(pid < 0) {
        std::cout<<"Failed to Create a new Process"<<std::endl;
    }
    else if (pid == 0) { // This code will be executed in Child Process Only
        std::cout<<"I am the child process."<<std::endl;
        std::cout<<"  My ID : "<<getpid()<<std::endl;
        std::cout<<"  My parent's ID : "<<getppid()<<std::endl;
        execv("/bin/ls", newargv); // turn the child into "ls"
    }
    else if (pid > 0) { // This code will be executed in Parent Process Only
        std::cout<<"I am the parent process."<<std::endl;
        std::cout<<"  My ID : "<<getpid()<<std::endl;
        std::cout<<"  My child's ID : "<<pid<<std::endl;
    }
}