What is a Process?

- an executing instance of a program
  - how is this different from a program?
- a virtual private computer
  - what does a virtual computer look like?
  - how is a process different from a virtual machine?
- a process is an \textit{object}
  - characterized by its properties (state)
  - characterized by its operations

What is “state”?

- the primary dictionary definition of “state” is
  - “a mode or condition of being”
- an object may have a wide range of possible states
- all persistent objects have “state”
  - distinguishing it from other objects
  - characterizing object’s current condition
- contents of state depends on object
  - complex operations often mean complex state
  - we can save/restore the aggregate/total state
  - we can talk of a subset (e.g. scheduling state)

Program vs Process Address Space

- load module (output of linkage editor)
  - all external references have been resolved
  - all modules combined into a few segments
  - includes multiple segments (text, data, BSS)
- code must be loaded into memory
  - a virtual code segment must be created
  - code must be read in from the load module
  - map segment into virtual address space
- code segments are read/only and sharable
  - many processes can use the same code segments
Address Space: Data Segments
- data too must be initialized in address space
  - process data segment must be created
  - initial contents must be copied from load module
  - BSS: segments to be initialized to all zeroes
  - map segment into virtual address space
- data segments
  - are read/write, and process private
  - program can grow or shrink it (with sbrk syscall)

Address Space: Stack Segment
- size of stack depends on program activities
  - grows larger as calls nest more deeply
  - amount of local storage allocated by each procedure
  - after calls return, their stack frames can be recycled
- OS manages the process's stack segment
  - stack segment created at same time as data segment
  - some allocate fixed sized stack at program load time
  - some dynamically extend stack as program needs it
- Stack segments are read/write and process private

Address Space: Shared Libraries
- static libraries are added to load module
  - each load module has its own copy of each library
  - program must be re-linked to get new version
- make each library a sharable code segment
  - one in-memory copy, shared by all processes
  - keep the library separate from the load modules
  - operating system loads library along with program
- reduced memory use, faster program loads
- easier and better library upgrades

Other Process State
- registers
  - general registers
  - program counter, processor status
  - stack pointer, frame pointer
- processes own OS resources
  - open files, current working directory, locks
- processes have OS-related state
  - Process ID, User ID, Group ID, scheduling priority
  - registered signal handlers, queued events, ...

Process Operations: fork
- parent and child are identical:
  - data and stack segments are copied
  - all the same files are open
- code sample:
  ```c
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr, "Fork failed\n");
  } else if (rc == 0) {
    fprintf(stderr, "Child\n");
  } else
    fprintf(stderr, "Fork succeeded, child pid = %d\n", rc);
  ```

Process Operations: wait
- await termination of a child process
  - collect exit status
- code sample:
  ```c
  int rc = waitpid(pid, &status, 0);
  if (rc == 0) {
    fprintf(stderr, "process %d exited rc=%d\n", pid, status);
  }
  ```
Process Operations: exec

- load new program, pass parameters
  - address space is completely recreated
  - all open files remain open
  - available in many polymorphisms
- code sample:
  ```c
  char *myargs[3];
  myargs[0] = "wc";
  myargs[1] = "myfile";
  myargs[2] = NULL;
  int rc = execvp(myargs[0], myargs);
  ```

Variations on Process Creation

- tabula rasa – a blank slate
  - a new process with minimal resources
  - it must set up all resources for itself
- run – fork + exec
  - create new process to run a specified command
- a cloning fork is a more expensive operation
  - much data and resources to be copied
  - convenient for setting up pipelines
  - allows inheritance of exclusive use devices

Representing a Process

- all (not just OS) objects have descriptors
  - the identity of the object
  - the current state of the object
  - references to other associated objects
- Process state is in multiple places
  - parameters and object references in a descriptor
  - app execution state is on the stack, in registers
  - each Linux process has a supervisor-mode stack
    - to retain the state of in-progress system calls
    - to save the state of an interrupt preempted process

Resident and non-Resident State

(resident process descriptor)

- state that could be needed at any time
- information needed to schedule process
  - run-state, priority, statistics
  - data needed to signal or awaken process
- identification information
  - process ID, user ID, group ID, parent ID
- communication and synchronization resources
  - semaphores, pending signals, mail-boxes
- pointer to non-resident state

(non-resident process state)

- information needed only when process runs
  - can swap out to free memory for other processes
- execution state
  - supervisor mode stack
  - including: saved register values, PC, PS
- pointers to resources used when running
  - current working directory, open file descriptors
- pointers to text, data and stack segments
  - used to reconstruct the address space
Creating a new process

• allocate/initialize resident process description
• allocate/initialize non-resident description
• duplicate parent resource references (e.g. fds)
• create a virtual address space
  — allocate memory for code, data and stack
  — load/copy program code and data
  — copy/initialize a stack segment
  — set up initial registers (PC, PS, SP)
• return from supervisor mode into new process

Limited Direct Execution

• CPU directly executes all application code
  — punctuated by occasional traps (for system calls)
  — with occasional timer interrupts (for time sharing)
• Maximizing direct execution is always the goal
  — for Linux user mode processes
  — for OS emulation (e.g. Windows on Linux)
  — for virtual machines
• Enter the OS as seldom as possible
  — get back to the application as quickly as possible

Asynchronous Exceptions

• some errors are routine
  — end of file, arithmetic overflow, conversion error
  — we should check for these after each operation
• some errors occur unpredictably
  — segmentation fault (e.g. dereferencing NULL)
  — user abort (^C), hang-up, power-failure
• these must raise asynchronous exceptions
  — some languages support try/catch operations
  — computers support traps
  — operating systems also use these for system calls

System Call Trap Gates

(Trap Handling)

• hardware trap handling
  — trap cause as index into trap vector table for PC/PS
  — load new processor status word, switch to supv mode
  — push PC/PS of program that caused trap onto stack
  — load PC (w/addr of 1st level handler)
• software trap handling
  — 1st level handler pushes all other registers
  — 1st level handler gathers info, selects 2nd level handler
  — 2nd level handler actually deals with the problem
    • handle the event, kill the process, return ...

Using Traps for System Calls

• reserve one illegal instruction for system calls
  — most computers specifically define such instructions
• define system call linkage conventions
  — call: r0 = system call number, r1 points to arguments
  — return: r0 = return code, cc indicates success/failure
• prepare arguments for the desired system call
• execute the designated system call instruction
• OS recognizes & performs requested operation
• returns to instruction after the system call
Stacking and unstacking a System Call

User-mode Stack
- stack frames from application computation
- resumed computation

Supervisor-mode Stack
- user mode PC & PS
- saved user mode registers
- parameters to system call handler
- return PC
- system call handler stack frame

direction of growth

(Returning to User-Mode)
- return is opposite of interrupt/trap entry
  - 2nd level handler returns to 1st level handler
  - 1st level handler restores all registers from stack
  - use privileged return instruction to restore PC/PS
  - resume user-mode execution at next instruction
- saved registers can be changed before return
  - change stacked user r0 to reflect return code
  - change stacked user PS to reflect success/failure

Asynchronous Events
- some things are worth waiting for
  - when I read(), I want to wait for the data
- sometimes waiting doesn’t make sense
  - I want to do something else while waiting
  - I have multiple operations outstanding
  - some events demand very prompt attention
- we need event completion call-backs
  - this is a common programming paradigm
  - computers support interrupts (similar to traps)
  - commonly associated with I/O devices and timers

User-Mode Signal Handling
- OS defines numerous types of signals
  - exceptions, operator actions, communication
- processes can control their handling
  - ignore this signal (pretend it never happened)
  - designate a handler for this signal
  - default action (typically kill or coredump process)
- analogous to hardware traps/interrupts
  - but implemented by the operating system
  - delivered to user mode processes

Signals and Signal Handling
- when an asynchronous exception occurs
  - the system invokes a specified exception handler
- invocation looks like a procedure call
  - save state of interrupted computation
  - exception handler can do what ever is necessary
  - handler can return, resume interrupted computation
- more complex than a procedure call and return
  - must also save/restore condition codes & volatile regs
  - may abort rather than return

Signals: sample code
```c
int fault_expected, fault_happened;
void handler( int sig) {
    if (!fault_expected) exit(-1); /* if not expected, die */
    else fault_happened = 1; /* if expected, note it happened */
}
signal(SIGHUP, SIGIGNORE); /* ignore hang-up signals */
signal(SIGSEGV, &handler); /* handle segmentation faults */
```
Stacking a signal delivery

- p1: parameters
- p0: return address
- p1: saved registers
- p0: local variables
- p0: computation

PC/PS (at time of exception)

handler: saved registers
handler: local variables

stack frame (pushed by signal)

Supplementary Slides

Indirect binding to shared libraries

code segment (read only)

jump foo

redirection table

shared library (read only, at well known location)

Loading and Binding w/DLLs

code segment (read only)

call foo

new code segment

Procedure Linkage Table (read/write)

run time loader

Limitations of Shared Libraries

- not all modules will work in a shared library
  - they cannot define/include static data storage
- they are read into program memory
  - whether they are actually needed or not
- called routines must be known at compile-time
  - only the fetching of the code is delayed 'til run-time
  - symbols known at compile time, bound at link time
- Dynamically Loadable Libraries are more general
  - they eliminate all of these limitations ... at a price

assignments

- reading for the next lecture
  - Arpaci ch 7 ... CPU Scheduling
  - Arpaci ch 8 ... Multi-Level Feedback
  - Arpaci ch 10 ... Multi-CPU Scheduling (skim)
  - real-time scheduling

Quiz 4 is due before the lecture!

Start project 1 before lab session

Supplementary Slides
(run-time binding to DLLs)

- Load module includes a Procedure Linkage Table
  - Addresses for routines in DLL resolve to entries in PLT
  - Each PLT entry contains a system call to run-time loader (asking it to load the corresponding routine)
- First time a routine is called, we call run-time loader
  - Which finds, loads, and initializes the desired routine
  - Changes the PLT entry to be a jump to loaded routine
  - Then jumps to the newly loaded routine
- Subsequent calls through that PLT entry go directly

Shared Libraries vs. DLLs

- Both allow code sharing and run-time binding
- Shared libraries
  - Do not require a special linkage editor
  - Shared objects obtained at program load time
- Dynamically Loadable Libraries
  - Require smarter linkage editor, run-time loader
  - Modules are not loaded until they are needed
    - Automatically when needed, or manually by program
  - Complex, per-routine, initialization can be performed
    - E.g., allocation of private data area for persistent local variables

Dynamic Loading

- DLLs are not merely “better” shared libraries
  - Libraries are loaded to satisfy static external references
  - DLLs are designed for dynamic binding
- Typical DLL usage scenario
  - Identify a needed module (e.g., device driver)
  - Call RTL to load the module, get back a descriptor
  - Use descriptor to call initialization entry-point
  - Initialization function registers all other entry points
  - Module is used as needed
  - Later we can unregister, free resources, and unload