Processes, Execution, and State

3A. What is a Process?
3B. Process Address Space
3C. Process Operations
3D. Implementing Processes
3E. Asynchronous Exceptions
3U. User-mode Exceptions
3X. Object modules and Load modules
3Z. Linkage conventions

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 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

Address Space: Code Segments

• 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

Characteristics of Libraries

- Many advantages
  - Reusable code makes programming easier
  - A single well written/maintained copy
  - Encapsulates complexity ... better building blocks
- Multiple bind-time options
  - Static ... include in load module at link time
  - Shared ... map into address space at exec time
  - Dynamic ... choose and load at run-time
- It is only code ... it has no special privileges

Sharable executables

- Code segments are usually read-only
  - one copy could be shared by multiple processes
  - allow more process to run in less memory
- Code has been relocated to specific addresses
  - all proc must use shared code at the same address
- Only the code segments are sharable
  - each process requires its own copy of writable data
  - data must be loaded into each process at start time

Advantages of Shared Libraries

- Reduced memory consumption
  - one copy can be shared by multiple processes/programs
- Faster program start-ups
  - if it is already in memory, it need not be loaded again
- Simplified updates
  - library modules are not included program load modules
  - library can be updated (e.g. new version w/ bug fixes)
  - programs automatically get new version when restarted

Processes, Execution and State

Sharing libraries

- Library modules are usually added to load module
  - each load module has its own copy of each library
    - this dramatically increases the size of each process
    - program must be re-linked to incorporate new library
      - existing load modules don't benefit from bug fixes
  - 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

Address space – shared executable

- Shared code
- Private data
- Private stack

Shared Libraries

- Code segments are usually read-only
  - one copy could be shared by multiple processes
  - allow more process to run in less memory
- Code has been relocated to specific addresses
  - all proc must use shared code at the same address
- Only the code segments are sharable
  - each process requires its own copy of writable data
  - data must be loaded into each process at start time
address space – shared libraries

Implementing Shared Libraries
- multiple code segments in a single address space
  - one for the main program, one for each shared library
  - each sharable, and mapped in at a well-known address
- deferred binding of references to shared libs
  - applications are linkage edited against a stub library
    - stub module has addresses for each entry point, but no code
    - linkage editor resolves all refs to standard map-in locations
  - loader must find a copy of each referenced library
    - and map it in at the address where it is expected to be

Stub modules vs real shared libraries

Indirect binding to shared libraries

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

Loading and Binding w/DLLs
(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

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);
  ```

Variations on Process Creation

• tabula rasa — a blank slate
  — a new process with minimal resources
  — a few resources may be passed from parent
  — most resources opened, from scratch, by child
• run – fork + exec
  — create new process to run a specified command
  — a cloning fork is a more expensive operation
• 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

Windows Process Creation

• The CreateProcess() system call
• A very flexible way to create a new process
• Numerous parameters to shape the child
  — name of program to run
  — security attributes (new or inherited)
  — open file handles to pass/inherit
  — environment variables
  — initial working directory
Process Forking

- The way Unix/Linux creates processes
  - child is a clone of the parent
  - the classical Computer Science fork operation
- Occasionally a clone is what you wanted
  - likely for some kinds of parallel programming
- Program in child process can adjust resources
  - change input/output file descriptors
  - change working directory
  - change environment variables
  - choose which program to run

What Happens After a Fork?

- There are now two processes
  - with different process ID numbers
  - but otherwise nearly identical
- How do I profitably use that?
  - two processes w/same code & program counter
  - figure out which is which
  - parent process goes one way
  - child process goes another
    - perhaps adjust process resources
    - perhaps load a new program into the process
    - this code takes the place of (CreateProcess) parameters

Process Operations: exec

- load new program, pass parameters
  - address space is completely recreated
  - open files remain open, disabled signals disabled
  - 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);
  ```

How Processes Terminate

- Perhaps voluntarily
  - by calling the exit(2) system call
- Perhaps involuntarily
  - as a result of an unhandled signal/exception
    - a few signals (e.g. SIGKILL) cannot be caught
  - Perhaps at the hand of another
    - a parent sends a termination signal (e.g. TERM)
    - a system management process (e.g. INT, HUP)

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);
  }
  ```

The State of a Process

- Registers
  - Program Counter, Processor Status Word
  - Stack Pointer, general registers
- Address space
  - size and location of text, data, stack segments
  - size and location of supervisor mode stack
- System Resources and Parameters
  - open files, current working directory, ...
  - owning user ID, parent PID, scheduling priority, ...
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
  - copyinitialize a stack segment
  - set up initial registers (PC, PS, SP)
- return from supervisor mode into new process

Forking and the Data Segments

- Forked child shares parent’s code segment
  - a single read only segment, referenced by both
- Stack and Data segments are private
  - each process has its own read/write copy
  - child’s is initialized as a copy of parent’s
  - copies diverge w/subsequent updates
- Common optimization: Copy-on-Write
  - start with a single shared read/only segment
  - make a copy only if parent (or child) changes it
Forking a New Process

Loading (exec) a Program

- We have a load module
  - The output of linkage editor
  - All external references have been resolved
  - All modules combined into a few segments
- A computer cannot “execute” a load module
  - Computers execute instructions in memory
  - An entirely new address space must be created
  - Memory must be allocated for each segment
  - Code must be copied from load module to memory

Loading a new program (exec)

Process Virtual Address Space

Process Terminated

- Reclaim any resources it may be holding
  - memory
  - locks
  - access to hardware devices
- Inform any other process that needs to know
  - those waiting for interprocess communications
  - parent (and maybe child) processes
- Remove process descriptor from process table

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

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
Hardware: Traps and Interrupts
• Used to get immediate attention from S/W
  – Traps: exceptions recognized by CPU
  – Interrupts: events generated by external devices
• The basic processes are very similar
  – program execution is preempted immediately
  – each trap/interrupt has a numeric code (0-n)
  – that is used to index into a table of PC/PS vectors
  – new PS is loaded from the selected vector
  – previous PS/PC are pushed on to the (new) stack
  – new PC is loaded from the selected vector

Review (User vs. Supervisor mode)
• the OS executes in supervisor mode
  – able to perform I/O operations
  – able to execute privileged instructions
    • e.g. enable, disable and return from interrupts
  – able update memory management registers
    • to create and modify process address spaces
  – access data structures within the OS
• application programs execute in user mode
  – they can only execute normal instructions
  – they are restricted to the process’s address space

Direct Execution
• Most operations have no security implications
  – arithmetic, logic, local flow control & data movement
• Most user-mode programs execute directly
  – CPU fetches, pipelines, and executes each instruction
  – this is very fast, and involves zero overhead
• A few operations do have security implications
  – h/w refuses to perform these in user-mode
  – these must be performed by the operating system
  – program must request service from the kernel

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

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

System Call Trap Gates
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
- system call handler stack frame

(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 ...

(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

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 */
... fault_happened = 0; fault_expected = 1;
... /* code that might cause a segmentation fault */
```
Stacking a signal delivery

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

stack (at time of exception)

PC/PS (at time of exception)

signal handler sees a completely standard appearing stack frame.

Supplementary Slides

Assignments

- Project
  - get started on P1A
    - many will have problems with terminal modes
    - many will have problems with non-blocking I/O
    - many will have problems with fork/exec

- Reading
  - A/D 7 (Scheduling)
  - A/D 8 (Adaptive Scheduling)
  - Real-Time Scheduling

(Compilation/Assembly)

- compiler
  - reads source code and header files
  - parses and understands “meaning” of source code
  - optimizer decides how to produce best possible code
  - code generation typically produces assembler code

- assembler
  - translates assembler directives into machine language
  - produces relocatable object modules
    - code, data, symbol tables, relocation information

Typical Object Module Format

- each code/data section is a block of information that should be kept together, as a unit, in the final program
(Relocatable Object Modules)

- code segments
  - relocatable machine language instructions
- data segments
  - non-executable initialized data, also relocatable
- symbol table
  - list of symbols defined and referenced by this module
- relocation information
  - pointers to all relocatable code and data items

Libraries

- programmers need not write all code for programs
  - standard utility functions can be found in libraries
- a library is a collection of object modules
  - a single file that contains many files (like a zip or jar)
  - these modules can be used directly, w/o recompilation
- most systems come with many standard libraries
  - system services, encryption, statistics, etc.
  - additional libraries may come with add-on products
- programmers can build their own libraries
  - functions commonly needed by parts of a product

Linkage Editing

- obtain additional modules from libraries
  - search libraries to satisfy unresolved external references
- combine all specified object modules
  - resolve cross-module references
  - copy all required modules into a single address space
  - relocate all references to point to the chosen locations
- result should be complete load module
  - no unresolved external addresses
  - all data items assigned to specific virtual addresses
  - all code references relocated to assigned addresses

Load Modules (ELF)

- ELF header
  - version info
  - target ISA
- section 1 header
  - load addr.
  - alignment
  - flags
- section 2 header
  - load addr.
  - alignment
  - flags
- section 3 header
  - symbol table
program loading – executable code

- 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)
    - each to be loaded, continguously, at a specified address
- a computer cannot "execute" a load module
  - computers execute instructions in memory
  - memory must be allocated for each segment
  - code must be copied from load module to memory
    - in ancient times this involved an additional relocation step

program loading – data segments

- code segments are read-only & fixed size
- programs include data as well as code
- data too must be initialized in address space
  - memory must be allocated for each data segment
  - initial contents must be copied from load module
  - BSS: segments to be initialized to all zeroes
- data segments read/write & variable size
  - execution can change contents of data segments
  - program can extend data segment to get more memory

Processes – stack frames

- modern programming languages are stack-based
  - greatly simplified procedure storage management
- each procedure call allocates a new stack frame
  - storage for procedure local (vs global) variables
  - storage for invocation parameters
  - save and restore registers
    - popped off stack when call returns
- most modern computers also have stack support
  - stack too must be preserved as part of process state

Simple procedure linkage conventions

calling routine

push p1; push first parameter
push p2; push second parameter
call foo; save pc, call routine

add =8,sp ; pop parameters
...

called routine

foo:
push r2-r6 ; save registers

sub =12,sp ; space for locals
...add r12,0
pop r12-r6
return

mov rslt, r0 ; return value
add =12,sp
pop r2-r6 ; restore regs
return ; restore pc

Sample stack frames

Process Stacks

- size of stack depends on activity of program
  - 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
UNIX stack space management

- Data segment starts at page boundary after code segment
- Stack segment starts at high end of address space
- Unix extends stack automatically as program needs more.
- Data segment grows up; Stack segment grows down
- Both grow towards the hole in the middle. They are not allowed to meet.

Traps vs. Interrupts

- Traps originate within the CPU
  - illegal instruction, invalid address, loss of power, ...
  - each represents a critical one-time event
- Interrupts come from devices on I/O bus
  - I/O completion, device added/removed, ...
  - they can be (temporarily) blocked/disabled
  - they may represent a continuing condition
  - may block acceptance of other similar interrupts
  - may not clear until they are acknowledged