Processes & Threads
SGG: Chapters 3 & 4

I. Overview of Challenges

A. OS is a tool that manages resources so that we can work done. But, how do we get work done? We write programs.
B. What defines a program (on disk and in memory)?
C. How does it get resources?
D. How do we run multiple programs?
   1. Who gets to run and for how long?
   2. When's a good time to start/stop running?
E. How do processes communicate?
F. How does a process efficiently get concurrency?

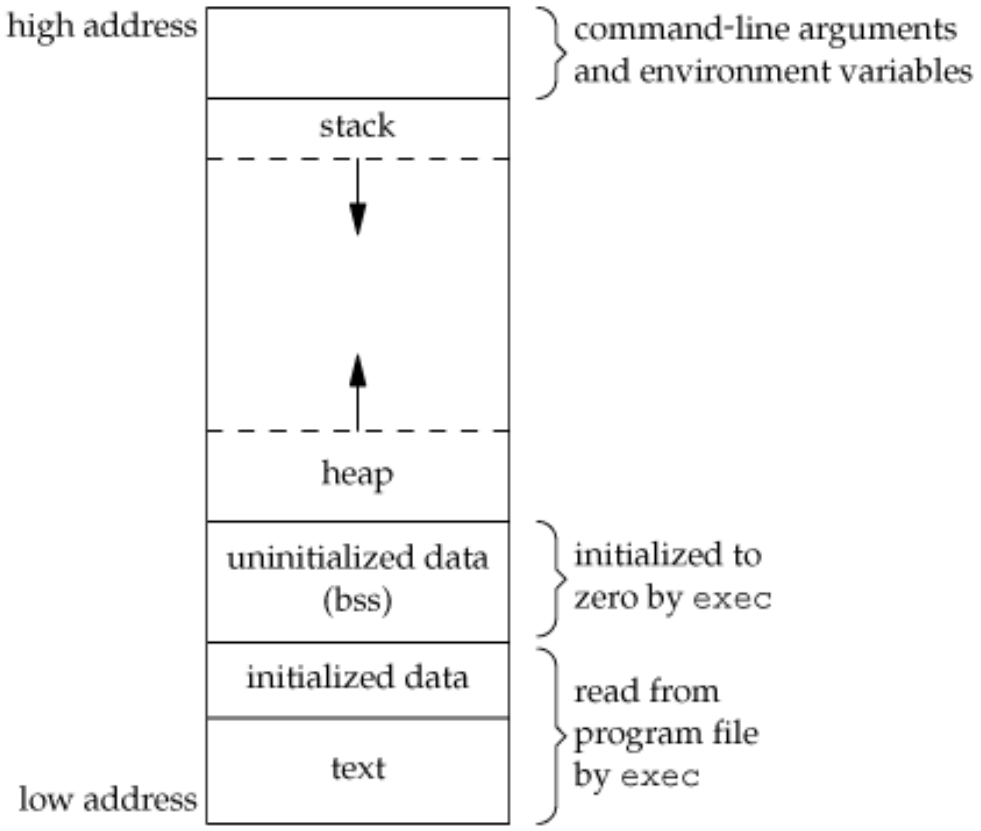
II. Process Overview

A. We start with processes, because, really, everything else in the OS is supportive of processes (except for maybe storage).
B. Definition: an **abstraction** of a **running program**
   1. A process is a program in execution
   2. It is the abstraction that allows for **multiprogramming**: the illusion of concurrency
   3. More than the code & data the describe execution; **Process control block** (metadata) used for controlling flow, saving state, accounting information
      a) Like in our lab: some "global" knowledge of process metadata
         (1) e.g. Linux: pid, state, parent, children, open files, address space

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

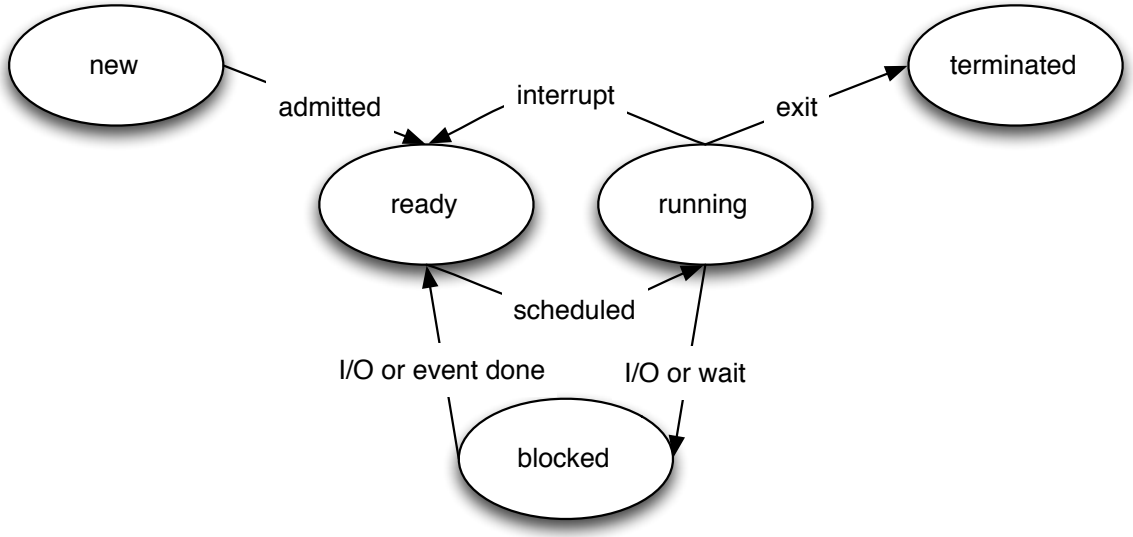
Figure (b). Some of the fields of a typical process table entry.

4. Process POV
   a) Own memory with consistent addressing (divorced from physical addressing)
   b) It has exclusivity over the CPU: It doesn't have to worry about scheduling;
   c) conversely, it doesn't know when it will be scheduled, so real time events require special handling
   d) Has some identity: pid, gid, uid
   e) Has a set of services available to it via the OS
      (1) Data (via file system)
      (2) Communication (sockets, IPC)
      (3) More resources (e.g. memory)
5. OS POV
   a) Partitioned memory: dedicated & shared address space; perhaps non-contiguous
   b) Process table
   c) Memory Layout
      (1) Text segment: machine instructions; shareable between identical processes; read-only
      (2) Data segment: for initialized data; e.g. int count = 99;
      (3) BSS (block started by symbol) segment: uninitialized data; int sum[10];
      (4) Heap: dynamic memory allocation
      (5) Stack: initial arguments and environment; stack frames



d) Other segments in a **program**: symbol table, debugging info, linkage table for dynamic library: not loaded to memory (not part of a process)

III. Process States



A. States:
1. **New**: Process is created, typically immediately moved to ready queue
2. **Running**: In memory, instructions are being executed
3. **Blocked**: Process waits for some event to occur (e.g. IO or signal)
4. **Ready**: Await processing (runnable, unblocked, but not running)
5. **Terminated**: Process is killed

B. Example: Running process can move from running to terminated (exit or killed), moved to ready (time slice up), or blocked (signaled to wait, I/O)
1. Compute a new RSA key?
2. Find the largest value in a 1TB of data?

IV. Intro. to Scheduling & Degree of Multiprogramming

A. **Scheduler** usually makes the transitions from ready to running; hides the details from the process/user

B. Blocked process cannot run until some event takes place
1. IO Interrupt, Signal

C. A process can block itself with a call like pause() or sleep()

D. Processes often have two types, characterized by what "state" they spend most of their time in
1. **I/O bound**: work is dependent on I/O; e.g. web browser, database, media streaming
2. **CPU bound**: work is dependent on CPU; e.g. scientific apps, cryptography

E. Example: cat foo.txt bar.txt baz.txt | grep mustang
1. grep may be ready to run first, but must wait for cat's IO
2. cat may run first, but must wait for the disk to return data

3. Q: What state does grep have? What state does cat have?

F. Scheduler must balance CPU- & I/O-bound processes
   1. Goal: to maximize CPU utilization

G. Modern schedulers have multiple queues
   1. ready queue, from ready processes are drawn; sometimes multiple ready queues based on process **priority**
   2. device queues

V. Process Execution

A. UNIX: fork() system call:creates an exact clone of the running process
   1. terminology: **child** & **parent**
   2. same memory images, same environment settings, same open files
   3. child often invokes exec() to change its memory image to a new program
   4. two steps allows the child to change file descriptors and other settings before exec()

B. Windows: CreateProcess() handles both creating and loading, (with 10 arguments to control altering the settings)

C. In both UNIX and Win, both parent and child have unique address spaces (isolated from each other, allowing for independent processing) (code may be shared, since it's read-only)

D. Different execution models
   1. Parent & child may execute independently
   2. Parent may wait for child
   3. Child may create more children (Process hierarchies)
   4. Parent may kill children

E. Process memory: code (text seg), variable (data seg) and the stack.

VI. Concurrency

A. One goal of a GP-OS is multiprogramming (concurrency)

B. The process model helps us achieve this

C. When a process moves from ready to running: **context switch** (and it's expensive)

1. save the current context: PCB, registers, process state, and some memory management info
2. Context-switch time is pure overhead - on the order of milliseconds
3. Some hardware support can help (e.g. multiple register sets)
4. If not done intelligently, you can spend more time context-switching than actual processing

D. **Q**: Who is in charge of context switching? Why shouldn't processes control context switching?

1. The could refuse to give up CPU (processes are greedy)
2. They're intentionally isolated, and don't have enough information about other processes
3. It would cause too much complication (every process would have to implement its own context switch code)

E. Context switching is **expensive**, and risks lowering utilization

1. **Example 1**: consider a web server as a single process; could only handle one connection at a time; how could a web sever support multiple connections? multiple processes; lots of overhead but maybe 99% of the memory is identical; context switching between processes decreases responsiveness; how would we manage a shared cache?
2. **Example 2**: consider the modern word processor, which "concurrently" 1) accepts user input, 2) performs auto-correct/spell checking; 3) auto-detects formatting 4) auto-save (nothing could be entered until save was complete); all impossible (or at least very inconvenient) without threads.

F. Concurrency within a process: **threads**

VII. Threads

A. Processes within a process; doesn't require kernel support to manage

1. Share code, data, files
2. Independent registers and stacks (logical)
3. Multiple **threads of execution** in a single process

B. Why threads?

1. To support multiple activities within a single process
2. **Responsiveness**: a process may continue to run, even if part of it is blocked (depends on thread implementation)
3. **Resource sharing**: Process are isolated so can only communicate with help from the kernel; threads share a memory space and a process's resources (e.g. files).
4. **Economy**: Allocating resources for multiple processes is costly, because it involves the kernel. Threads can be handled entirely in user space and are much faster to allocate and destroy. Allows for **pop-up threads**, dynamically created threads to support load.
5. **Hardware support**: modern CPUs support **multithreading** (hyperthreading), where threads can run in parallel (true concurrency)

C. Why not threads?

1. Share the same address space and global variable: loss of isolation: A runaway thread can wipe out other threads
2. However, these aren't different processes from different users: they are mutually trusted and working toward a common "goal"
3. Cannot signal threads (this is a process service)
4. No built-in mechanism for pre-emption, so threads should be well behaved (unlike greedy processes)
5. n threads see 1/n of a single CPU; CPU-bound processes

make thread moot, or detrimental

D. Other examples of thread usefulness?
   1. Web browser (tabs, simultaneous connections, javascript)
   2. Video games
   3. Anything with user input and processing or IO

E. **Q:** Why do threads need their own stack?
   1. They have different execution histories

F. **Q:** How do processes get their share of the CPU?
   1. Scheduler, in the kernel, based on timers (or something)

G. **Q:** How do threads get their share of the CPU?
   1. Thread libraries allow for voluntary yielding
   2. Thread can "join" another thread; wait for another thread to complete

H. **Q:** What happens to threads when you fork()?
   1. Should a child inherit the threads of their parent?
   2. What happens if two threads are competing for the same resource? E.g. who gets the input from a keyboard or established network connection?
   3. What happens if a thread closes a file another expects to be open?
   4. In reality (Linux): a child process, only the calling thread is replicated.

I. POSIX Threads (Pthreads)
   1. In addition to system calls, IEEE has a standard for threaded programs
      a) Portability, ease of use, correctness
   2. 60 calls, but the important ones are: pthread_*
      a) create, exit, yield, join (Look familiar?)

J. Pthreads can be a **user space** library: all functionality happens in user space, and the kernel knows nothing about them.
   1. can run on kernels or CPUs with no thread support;
   2. orders of magnitude faster than a kernel trap (Everything is a

local function call)
3. custom schedulers
4. **Q:** What happens when a thread makes an IO-bound system call?
   a) All threads will block, because the process is blocked
   b) there is a select() system call which will tell us if a read() will block
   c) sometimes a thread blocking a process is unavoidable: page fault.
5. Where we typically want threads is where there is a lot of IO blocking or system calls: the web server example above
   a) if we're trapping to the kernel, are we really gaining anything?

K. Threads libraries can also have **kernel** support (most modern OSes support kernel threads)

1. There is no user space runtime environment or thread table -- all is managed within the kernel
2. Support for multiple cores (hyperthreading)
3. All thread interfaces are system calls handled by the kernel
4. Instead of blocking on a system call, the kernel now has the intelligence to schedule another thread
5. Creating and destroying threads comes at a higher cost (the kernel trap we were trying to avoid above)
   a) Thread pools: a collection of pre-allocated threads, assigned as needed (an idea not exclusive to kernel threads though)

L. Hybrid Approach: A mapping of user-level threads to a kernel threads; ultimate flexibility at the cost of complication

1. One-to-One (Linux & Windows), Many-to-one; many-to-many

VIII. Interprocess Communication (IPC)

A. Motivating Example

1. cat foo.txt bar.txt | grep “Peterson” | sort

B. Benefits

1. **Information Sharing**: selectively control isolation
2. **Computation speedup**, with isolation
3. **Modularity**: e.g. microkernel
4. Allows for process **cooperation** (cooperative processes)

C. Two Strategies

1. Message Passing
   a) An high-level abstraction for exchanging packets of information over some interconnect
2. Shared Memory
   a) Region of memory available to different processes; writable by at least one process

D. Shared Memory

1. kernel plays a role in establishing and attaching the address space, but does not control read/write access beyond that
   a) can be **fast**
   b) how that memory is shared, and kept consistent, is left up to the processes
   c) limited to intra-computer communication
2. Procedurally:
   a) Sharer process allocates a region in their address space
   b) Other processes attach to that region
3. POSIX shared memory calls
   a) id = shmget(key, size, perms)
   b) ptr = shmat(id, addr, flag (ro or wo))
   c) shmdt(ptr)
   d) shmctl()
4. Example: Produce Consumer (see challenges ahead)

E. Message Passing

1. kernel establishes and oversees all communication
2. two primitives: send() and recv()

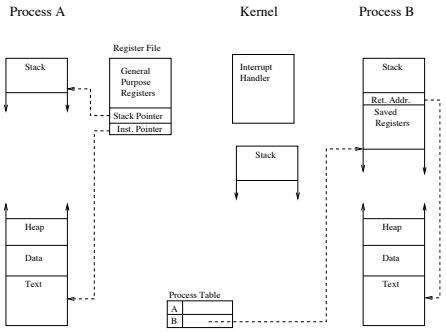
3. beyond intra-computer communication, facilitates processes over a network; link implementation is unimportant
4. Addressing: **direct** (to process) and **indirect** (to mailbox)
5. Challenges & Design Decisions:
   a) **synchronization**: do send and recv block or nonblock?
      (1) Q: What are the advantages/disadvantages of this decision?
   b) **buffering**: how much and how long do we store messages?
   c) **overhead**: every message requires two system calls (send & recv); passing a message is always slower than controlling access to shared memory (messages are copied)
   d) **reliability**: e.g. acknowledgement in the face of lost messages; out-of-order of delivery
   e) **authentication** of messages
   f) **addressing**: unicast, multi/broadcast; client-server; many-to-many
6. sockets are the common link between inter-computer processes

F. Pipes

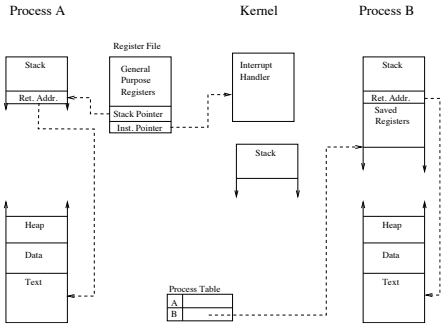
1. A special conduit between process (originally from UNIX); **anonymous pipe**: pipe(int fd[2]) creates a unidirectional pipe with read-end and write-end; using syscall read() and write()
2. Cannot be accessed outside the creating process
3. Since they are file descriptors, a child process can inherit the pipe
4. Fun: a pipe who has one end closed is **widowed**
5. **Named pipes**: bidirectional, w/o parent-child relationship (FIFOs);
   a) Creates a persistent file-like name
   b) Requires a writer and reader
6. form of message passing, but really a data stream (ordering, buffering, reliability, authentication all implied)

IX. Challenges (a preview of what's to come: scheduling & synchronization)

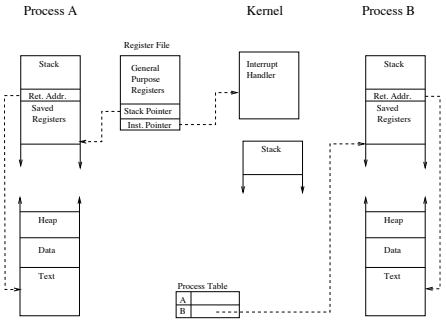
1. Example 1: In an airline reservation system, two processes want to reserve the last seat (shared data); what happens if they get there at the same time (race condition); very difficult to debug
   a) Solution: identify **critical regions/section**; prevent the reading and writing of data at the same time
   b) Mutual exclusion: a way to isolate access to a shared resource
   c) Many solutions for achieving mutual exclusion
2. Example 2: Producer/Consumer: a class of problem where two (or more!) processes, one produces something, the other consumes it (web browser, data base, etc.)
   a) How do we resolve proper sequencing when there are dependencies?
   b) What happens if a producer doesn't complete a write before losing the CPU to the consumer?
   c) How might we do it with message passing?
   d) How might we do it with shared memory?



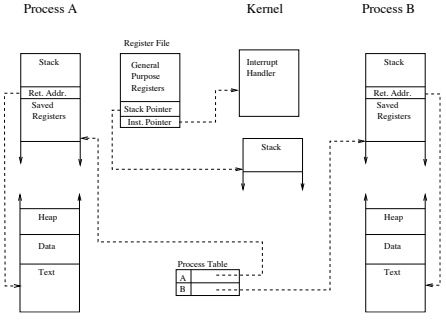
(a) While process A is running



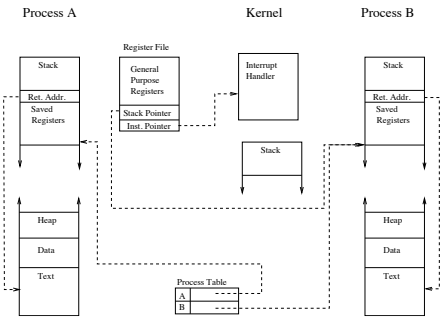
(b) After the interrupt



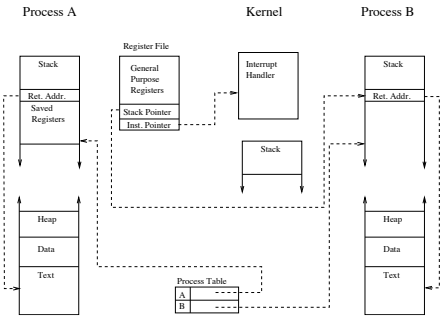
(c) Saving A's registers



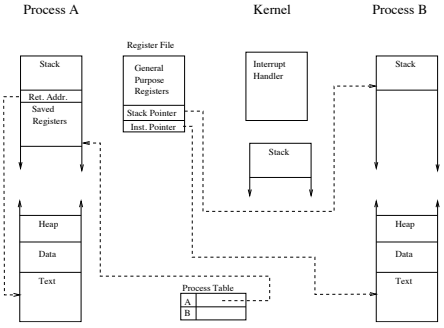
(d) switch to the operating system (kernel) stack



(e) Restore B's SP from process table.



(f) Pop B's registers



(g) After return from interrupt