Review

- Chapter 4: Processes
- Chapter 5: Threads
- Goal: To have an concrete understanding on the process abstraction

Little detour to explain the notion of process

- Suppose you want to perform the following work from a machine. For now, ignore any notions of OS and stuff and focus on how to solve this problem
  1. read r1
  2. read r2
  3. compute r3= function(r1, r2)
  4. write r3
- Assume that r1 is available on the hard disk, r2 is available on the serial port and r3 is written into a floppy disk

What operations do we need for “read r1”

- Allocate some memory to hold r1 (r1-op1)
  - Reclamation: memory may be full
  - Exclusivity: make sure that “someone” else doesn’t inadvertently share this memory
- Get the device ready to read r1 (r1-op2)
  - Hard disk maybe “parked”, the disk arms may not be where they should be, how do you know where the arms should be (disk organization etc.) You seek to the right sector and cylinder
- Read from disk to memory (r1-op3)
  - Ask the drive to read directly into the memory location for r1
- Deal with error conditions and delays (r1-op4)
  - Once the DMA controller tells you that read finished you are done

What operations do we need for “read r2”

- Allocate some memory to hold r2 (r2-op1)
  - Reclamation: memory may be full
  - Exclusivity: make sure that “someone” else doesn’t inadvertently share this memory
- Get the device ready to read r2 (r2-op2)
  - Might be a modem
- Read from port to memory (r2-op3)
  - Serial port controller may be polled or asynchronous
- Deal with error conditions and delays (r2-op4)

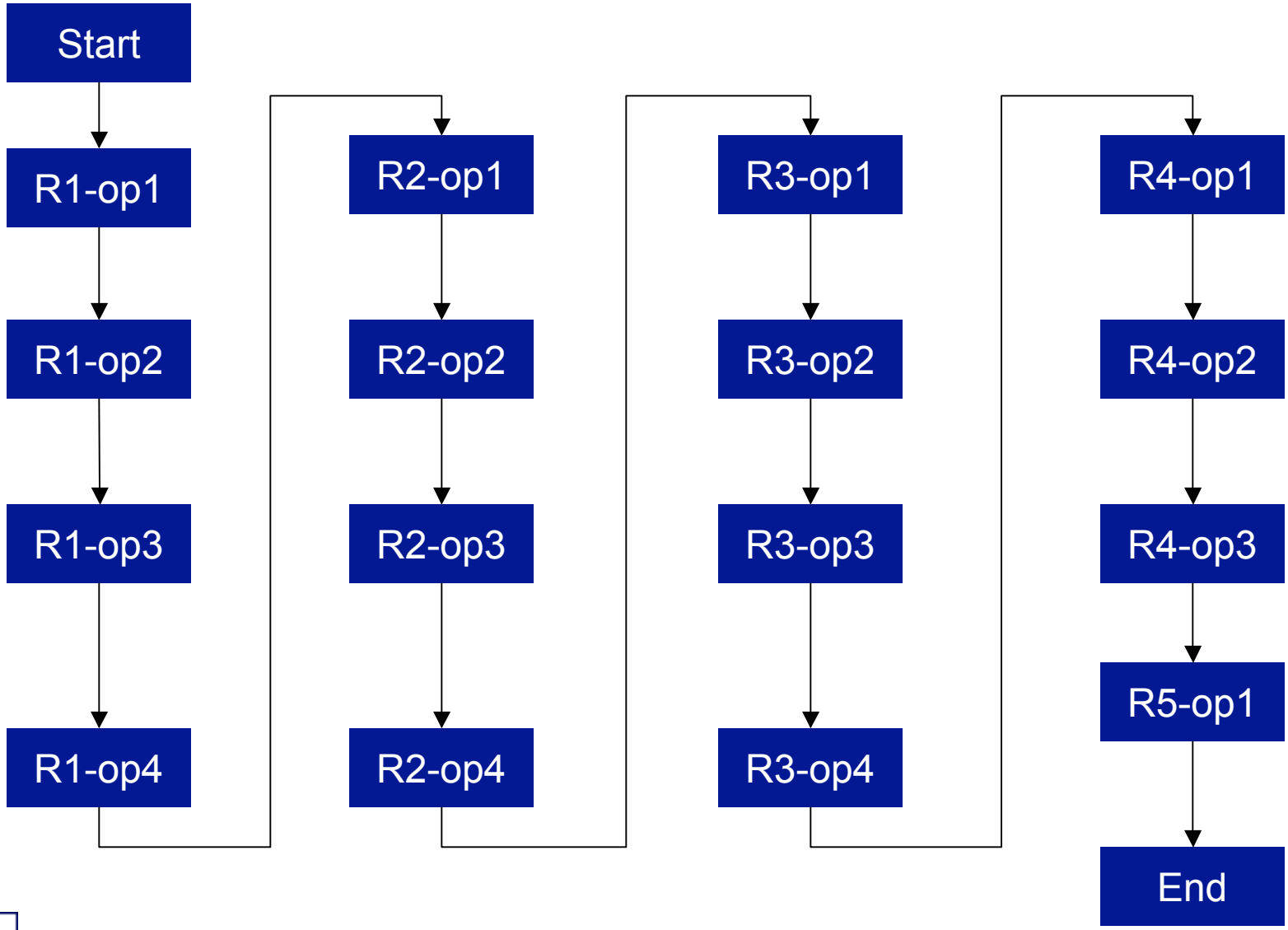
What operations do we need for “r3=function(r1,r2)”

- Allocate some memory to hold r3 (r3-op1)
  - Reclamation: memory may be full
  - Exclusivity: make sure that “someone” else doesn’t inadvertently share this memory
- Perform the *function* operation by passing r1 and r2 to *function* (call by value, call by reference etc.) (r3-op2)
  - *Function* can be shared with others (example, sqrt() function)
    - locally (math library) or
    - remotely (sqrt server)
  - You can write your own sqrt function
- Write function return value to memory r3 (r3-op3)
- Deal with error conditions (r3-op4)
  - Math overflow error

What operations do we need for “write r3”

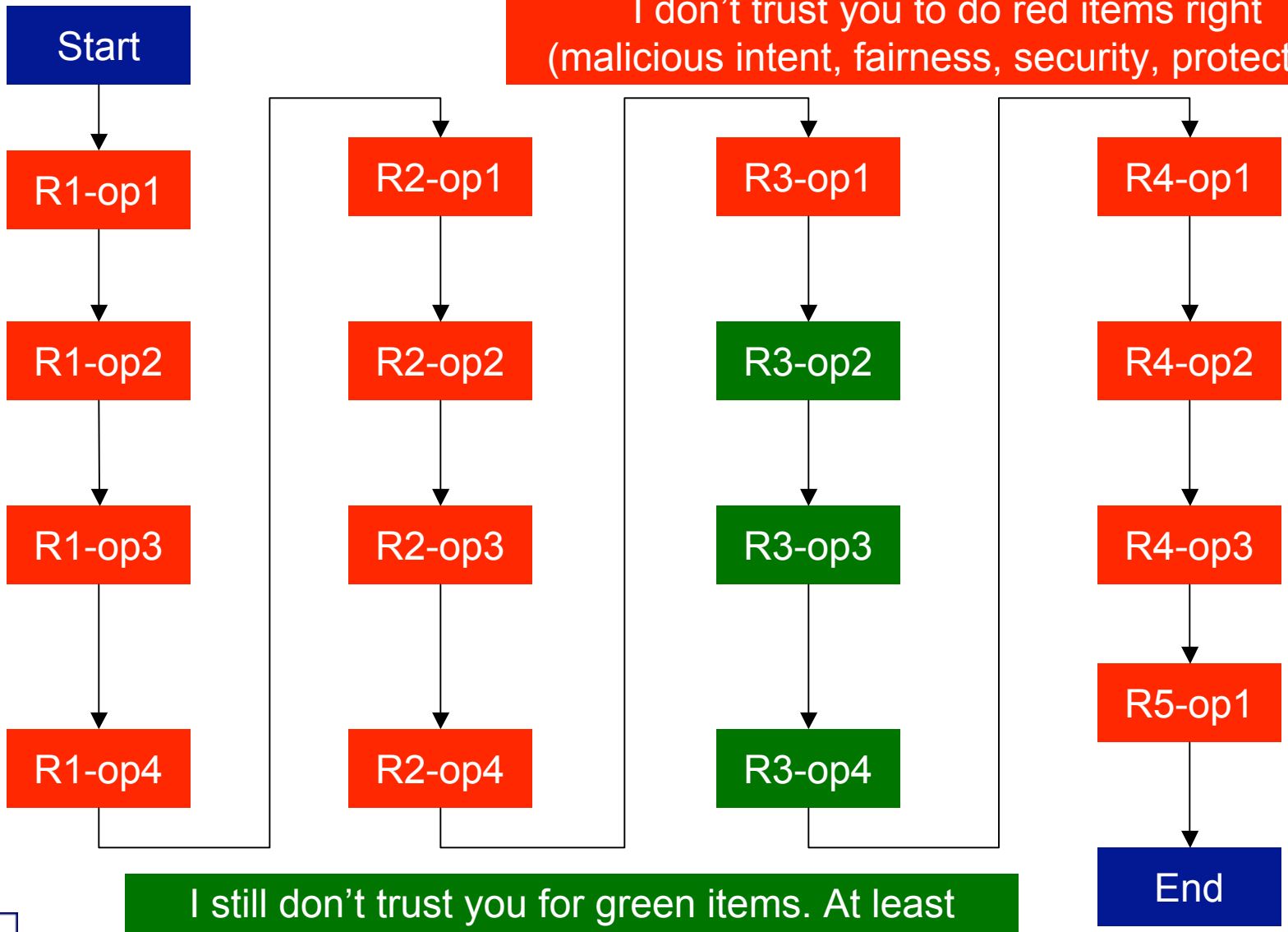
- Get the floppy device ready to write r3 (r4-op1)
  - How do we know where to write in the floppy. You can’t randomly write into any sector
- Start the write process (r4-op2)
  - Do we wait till the process finishes slowly? We don’t need to (asynchronous I/O) because we already have r3 in memory, if we need to use it again. But if we change r3 before it is written out, then the write semantics is broken (I that cast, we make a copy of r3 and write that copy)
  - We can wait just to be sure
- Deal with error conditions and delays (r4-op4)
- General cleanup (r5-op1)

Operation flow



What happens if you are not the only one in system

I don't trust you to do red items right
(malicious intent, fairness, security, protection)

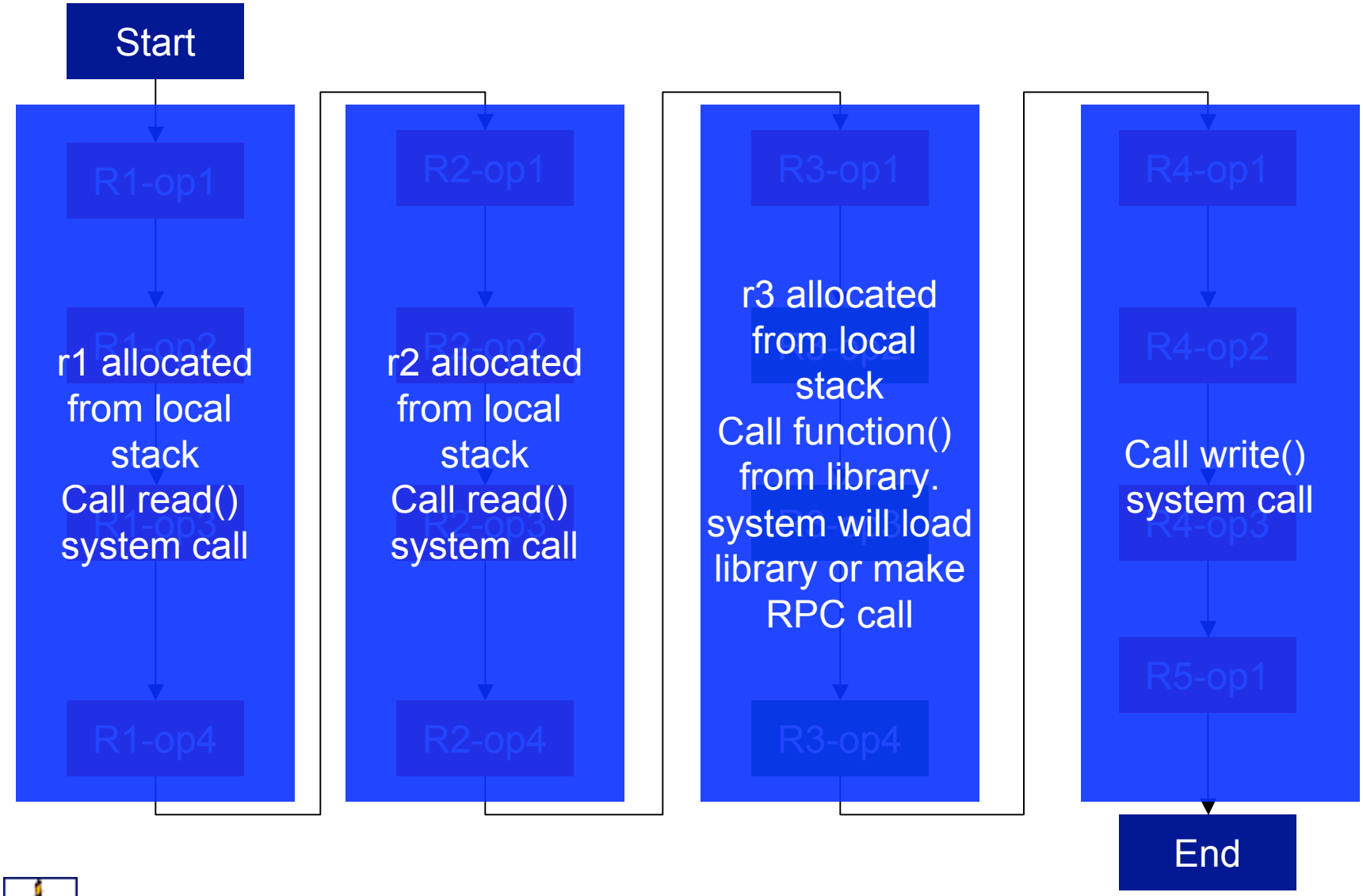


I still don't trust you for green items. At least
you wouldn't destroy the system

Solution

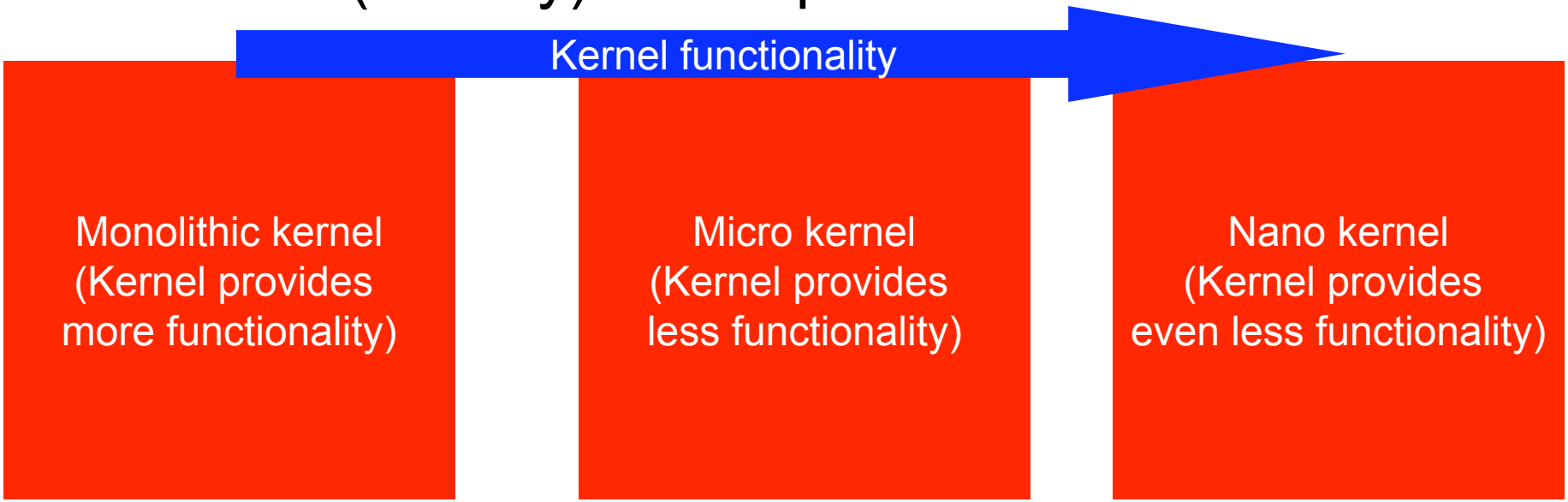
- Put everything into a process abstraction so that the kernel can know what is allowed and what is not, how much resources to allocate how to charge you for resources consumed etc.
- Put red items inside OS kernel
  - OS kernel basically does this part of what you want - items that you cannot trust users to use and allocate fairly by themselves
  - Use system calls to communicate between processes and kernel (this context switch is not free)
- Put green items inside dynamic libraries, remote procedure calls (RPC/RMI) [Swizzling arguments, return values, pointers (-endian, word size etc.)

What happens if you are not the only one in system



What do OS researchers do

- Whether r1-op1 stays inside the kernel (you pay the cost to cross from user to kernel space [protected space] or you want performance (and can deal with issues that the kernel provides you for free)
- The more the OS hides, the more easier it is for the user and (usually) lesser performance

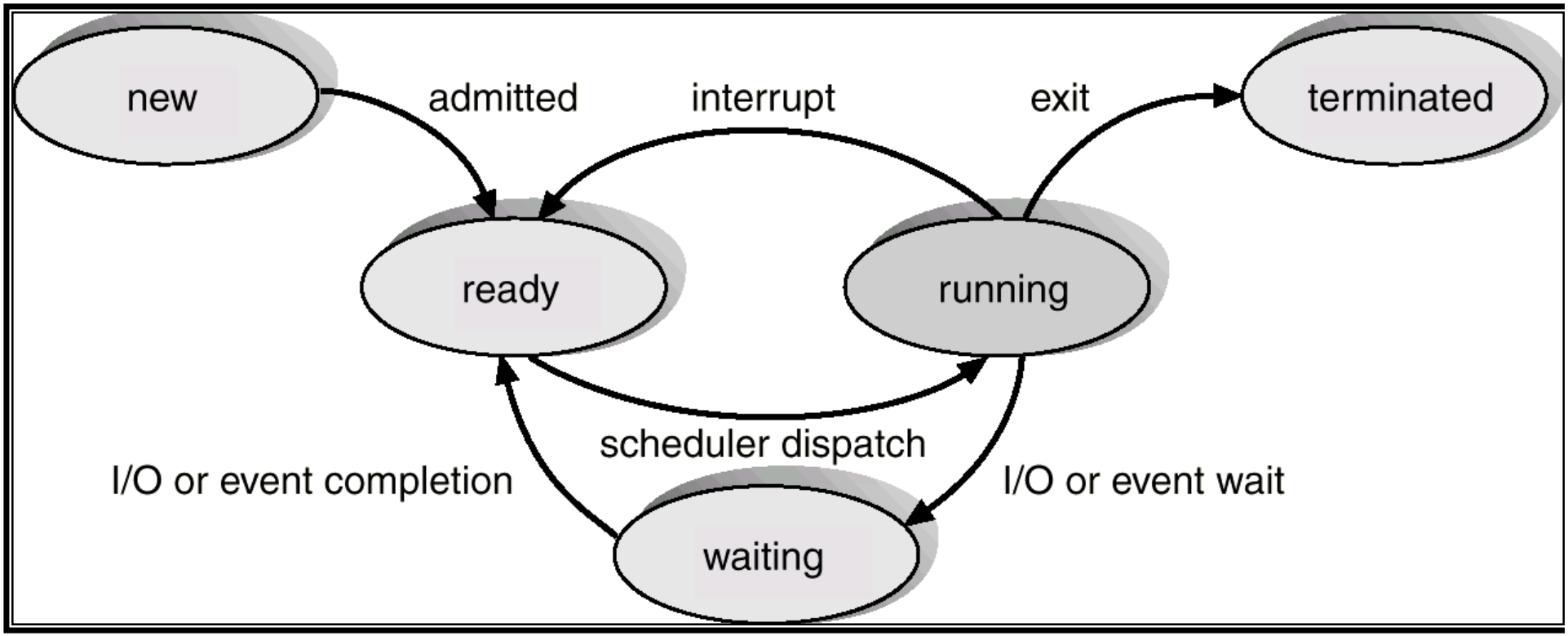


- The fundamental problem is: does the OS do it for you or do you do it in user space?

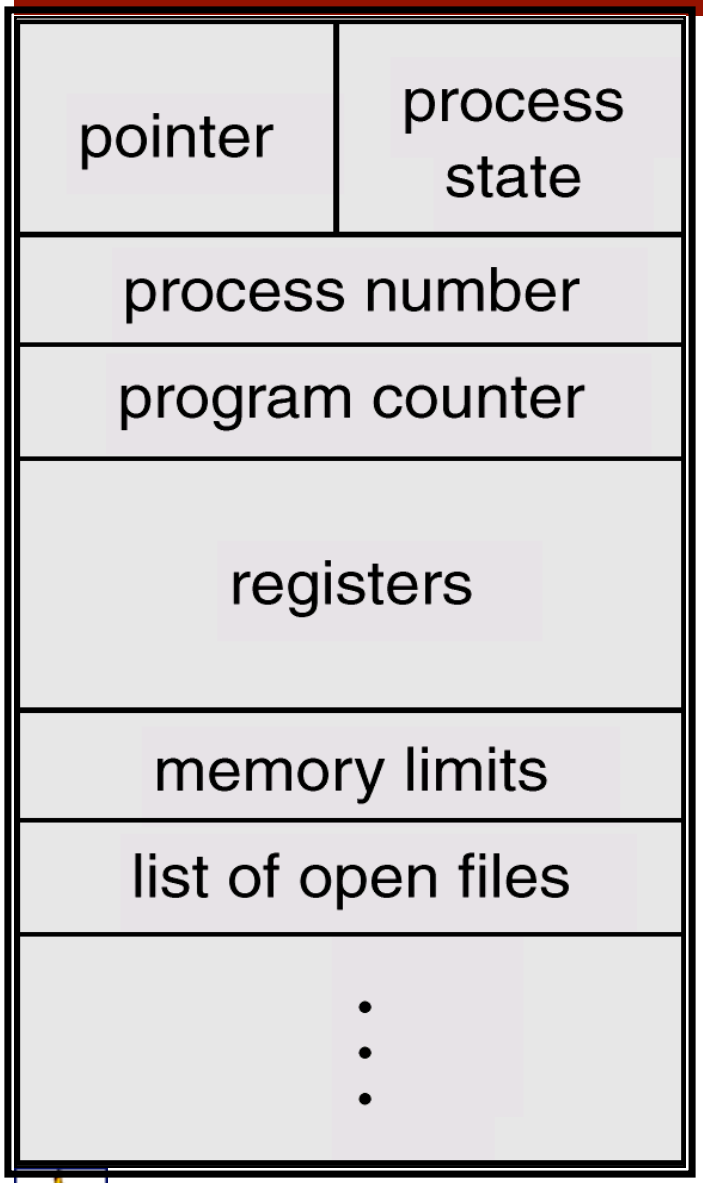
Process abstraction

- Process is a program in execution: It's the job of the OS to provide the environment required by the application to execute fruitfully
  - Program code
  - Data section
  - Execution context: Program counter, registers, stack
- Process has thread(s) of control
- Many processes "run" concurrently: Process scheduling
  - Fair allocation of CPU and I/O bound processes
    - Context switch
- Many users may run many tasks: Protect users from other users as well as from themselves.

Process States



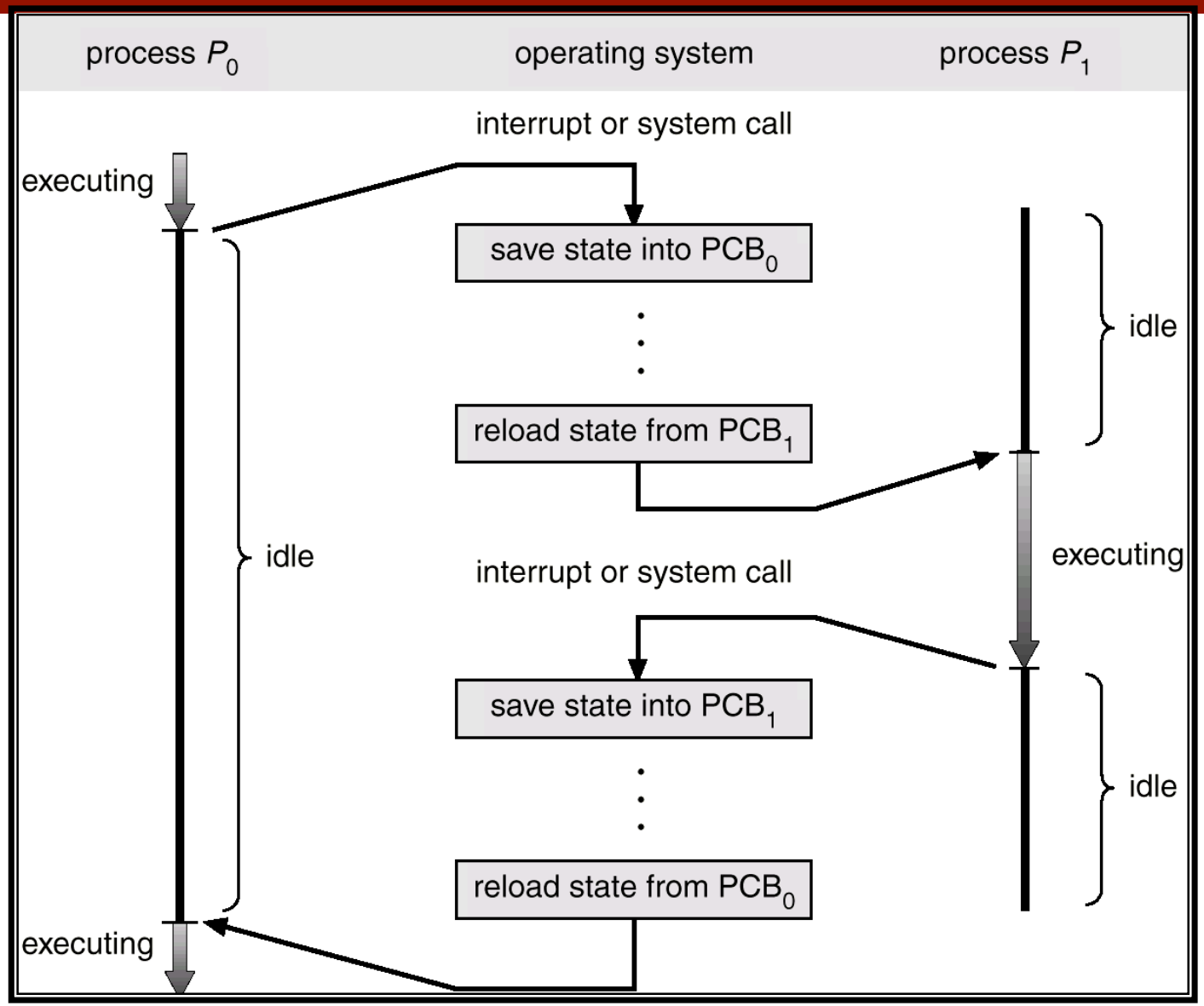
Process Control Block



Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process context switch



Process creation

- Creating new processes is expensive
  - Resource allocation issue
- Fork mechanism: UNIX, Windows NT
  - Duplicate the parent process
  - Shares file descriptors, memory is copied
  - Exec to create different process
  - Various optimizations to avoid copying the entire parent context (Copy on write (COW), etc..)
- Exec mechanism: VMS, Windows NT
  - New process is specifically loaded

Interprocess communication

- Processes need to communicate with each other
  - Naming
  - Message-passing
    - Direct (to process) or indirect (port, mailbox)
    - Symmetric or asymmetric (blocking, nonblocking)
    - Automatic or explicit buffering (capacity)
    - Send by copy or reference
    - Fixed size or variable size messages
  - Shared memory/mutexes
  - Remote Procedure Call (RPC/RMI)
- Bounded buffer problem
  - Producer buffers, consumer takes from buffer - finite buffer

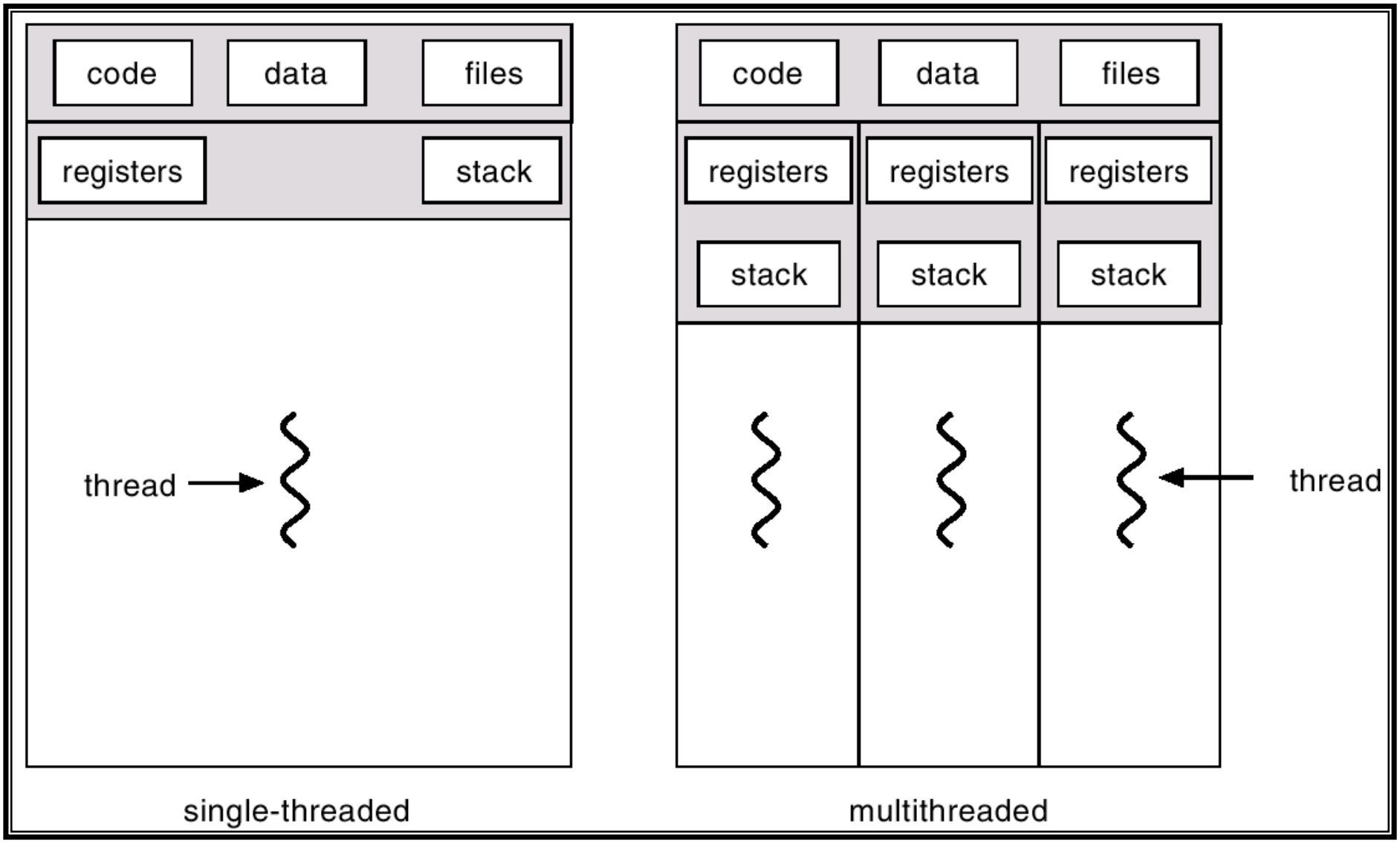
CPU scheduling

- Interleave processes so as to maximize utilization of CPU and I/O resources
- Scheduler should be fast as time spent in scheduler is wasted time
  - Switching context (h/w assists – register windows [sparc])
  - Switching to user mode
  - Jumping to proper location
- Preemptive scheduling:
  - Context switch without waiting for application to relinquish
  - Process could be in the middle of an operation
  - Especially bad for kernel structures
- Non-preemptive (cooperative) scheduling:
  - Can lead to Starvation

Threads

- Applications require concurrency. Threads provide a neat abstraction to specify concurrency
- E.g. word processor application
  - Needs to accept user input, display it on screen, spell check and grammar check
  - Implicit: Write code that reads user input, displays/formats it on screen, calls spell checked etc. while making sure that interactive response does not suffer. May or may not leverage multiple processors
  - Threads: Use threads to perform each task and communicate using queues and shared data structures
  - Processes: expensive to create and do not share data structures and so explicitly passed

Threaded application



Threads - Benefits

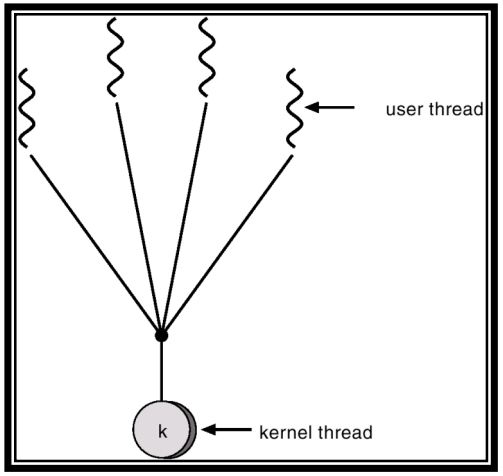
- Responsiveness
  - If one “task” takes too long, other “tasks” can still proceed
- Resource sharing: (No protection between threads)
  - Grammar checker can check the buffer as it is being typed
- Economy:
  - Process creation is expensive (spell checker)
- Utilization of multiprocessor architectures:
  - If we had four processors (say), the word processor can fully leverage them
- Pitfalls:
  - Shared data should be protected or results are undefined
    - Race conditions, dead locks, starvation (more later)

Thread types

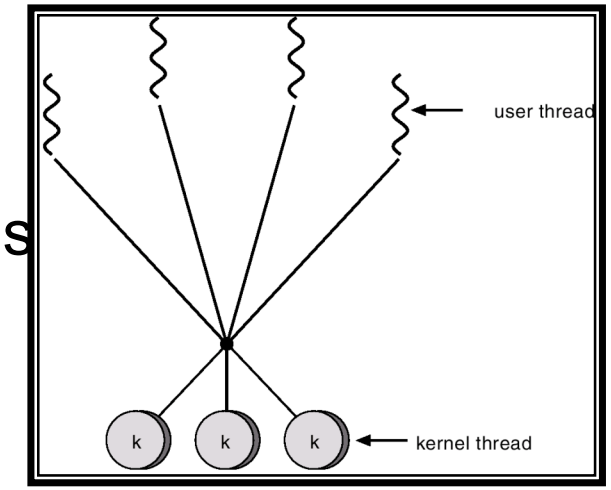
- Continuum: Cost to create and ease of management
- User level threads (e.g. pthreads)
  - Implemented as a library
  - Fast to create
  - Cannot have blocking system calls
  - Scheduling conflicts between kernel and threads. User level threads cannot do anything is kernel preempts the process
- Kernel level threads
  - Slower to create and manage
  - Blocking system calls are no problem
  - Most OS's support these threads

Threading models

- One to One model
  - Map each user thread to one kernel thread



- Many to one model
  - Map many user threads to a single kernel thread
  - Cannot exploit multiprocessors



- Many to many
  - Map *m* user threads to *n* kernel threads

Threading Issues:

- Cancellation:
  - Asynchronous or deferred cancellation
- Signal handling: which thread of a task should get it?
  - Relevant thread
  - Every thread
  - Certain threads
  - Specific thread
- Pooled threads (web server)
- Thread specific data

Wizard ‘ps -cfLeP’ output

```
UID       PID  PPID  LWP PSR   NLWP CLS PRI  STIME  TTY      LTIME CMD
root      0     0     1   -     1   SYS  96   Aug 03 ?         0:01 sched
root      1     0     1   -     1    TS  59   Aug 03 ?         7:12 /etc/init -
root      2     0     1   -     1   SYS  98   Aug 03 ?         0:00 pageout
root      3     0     1   -     1   SYS  60   Aug 03 ?       275:46 fsflush

root   477   352     1   -     1    IA  59   Aug 03 ??        0:0
                                                /usr/openwin/bin/fbconsole -d :0
root    62     1    14   -    14    TS  59   Aug 04 ?         0:00
                                                     /usr/lib/sysevent/syseventd
```

Discussion

- Constant tension between moving functionality to upper layers; involving the application programmer and performing automatically at the lower layers
- Automatically create/manage threads by compiler/ system? (open research question)