Announcements

- Program #2 is available
  - its on the web page
- Reading chapter 6 (6.1 and 6.2)

forking a new process

- create a PCB for the new process
  - copy most entries from the parent
  - clear accounting fields
  - buffered pending I/O
  - allocate a pid (process id for the new process)
- allocate memory for it
  - could require copying all of the parents segments
  - however, text segment usually doesn't change so that could be shared
  - might be able to use memory mapping hardware to help
    - will talk more about this in the memory management part of the class
- add it to the ready queue

Process Termination

- Process can terminate self
  - via the exit system call
- One process can terminate another process
  - use the kill system call
  - can any process kill any other process?
    - No, that would be bad.
    - Normally an ancestor can terminate a descendant
- OS kernel can terminate a process
  - exceeds resource limits
  - tries to perform an illegal operation
- What if a parent terminates before the child
  - called an orphan process
  - in UNIX becomes child of the root process
  - in VMS - causes all descendants to be killed

Termination (cont.) - UNIX example

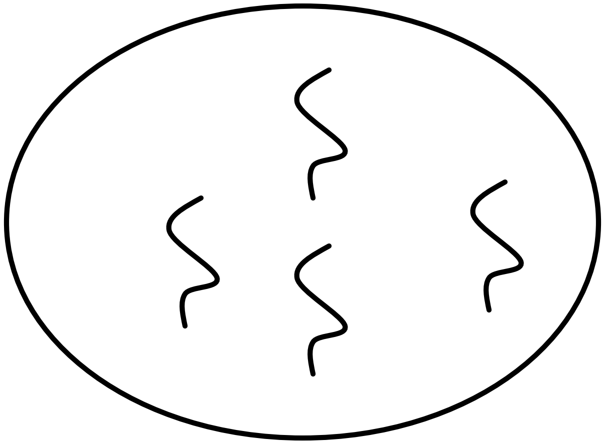
- Kernel
  - frees memory used by the process
  - moved process control block to the terminated queue
- Terminated process
  - signals parent of its death (SIGCHILD)
  - is called a zombie in UNIX
  - remains around waiting to be reclaimed
- parent process
  - wait system call retrieves info about the dead process
    - exit status
    - accounting information
  - signal handler is generally called the reaper
    - since its job is to collect the dead processes

Threads

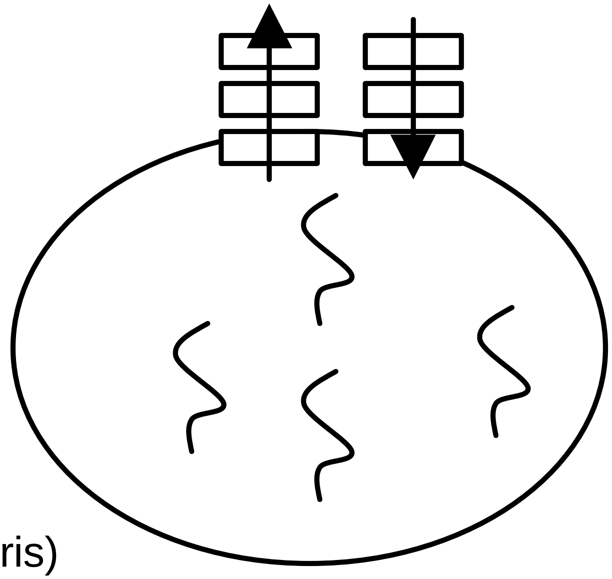
- processes can be a heavy (expensive) object
- threads are like processes but generally a collection of threads will share
  - memory (except stack)
  - open files (and buffered data)
  - signals
- can be user or system level
  - user level: kernel sees one process
    - \+ easy to implement by users
    - \- I/O management is difficult
    - \- in an multi-processor can't get parallelism
  - system level: kernel schedules threads

Thread Implementation

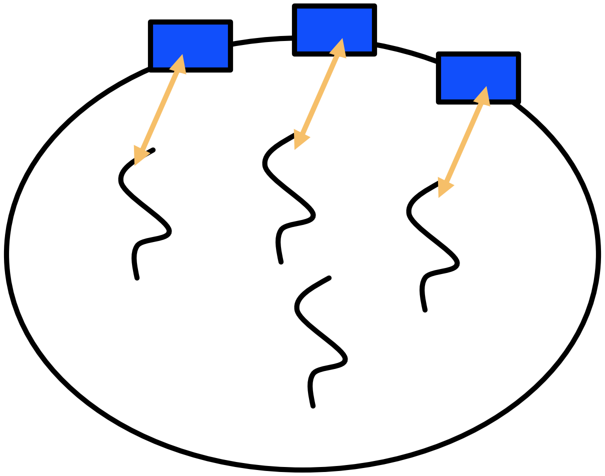
User Visible Threads



Async Kernel Calls (Digital Unix)



Light Weigth Processes (Solaris)



Cooperating Processes

- Often need to share information between processes
  - information: a shared file
  - computational speedup:
    - break the problem into several tasks that can be run on different processors
    - requires several processors to actually get speedup
  - modularity: separate processes for different functions
    - compiler driver, compiler, assembler, linker
  - convenience:
    - editing, printing, and compiling all at once

Interprocess Communication

- Communicating processes establish a link
  - can more than two processes use a link?
  - are links one way or two way?
  - how to establish a link
    - how do processes name other processes to talk to
      - use the process id (signals work this way)
      - use a name in the filesystem (UNIX domain sockets)
      - indirectly via mailboxes (a separate object)
- Use send/receive functions to communicate
  - send(dest, message)
  - receive(dest, message)

Producer-consumer pair

- producer creates data and sends it to the consumer
- consumer read the data and uses it
- examples: compiler and assembler can be used as a producer consumer pair
- Buffering
  - processes may not produce and consume items one by one
  - need a place to store produced items for the consumer
    - called a buffer
  - could be fixed size (bounded buffer) or unlimited (un-bounded buffer)