**THREADS:**

-Most modern applications are multithreaded

-Threads run within application

-Multiple tasks with the application can be implemented by separate threads

*(Update display, Fetch data, Spell checking)*

*-*Answer a network request

-Process creation is heavy-weight while thread creation is light-weight

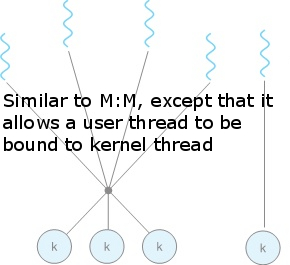
-Can simplify code, increase efficiency

-Kernels are generally multithreaded

***PROS***

**Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces

**Resource Sharing –** threads share resources of process, easier than shared memory or message passing

**Economy –** cheaper than process creation, thread switching lower overhead than context switching

**Scalability –** process can take advantage of multiprocessor architectures

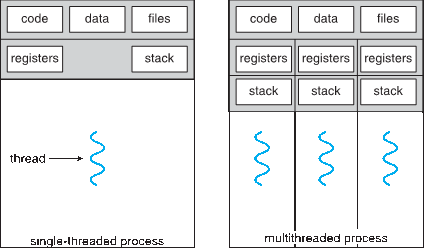
**CONS**

Semantics of **fork()** and **exec()** system calls

Signal handling, Synchronous and asynchronous

Thread cancellation of target thread

*(Asynchronous or deferred, Thread-local storage, Scheduler Activations)*

****

**Amdahl’s Law**

Both share Code, Data, and files.

*S* is serial portion

*N* processing cores

I.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

As *N* approaches infinity, speedup approaches 1 / *S* **Serial portion of an application has disproportionate effect on performance gained by adding additional cores**

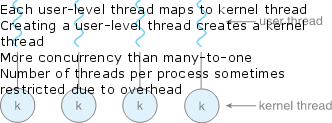
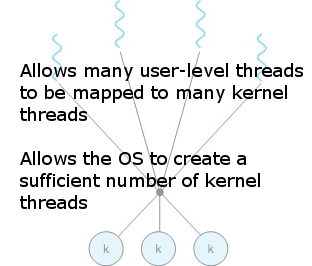
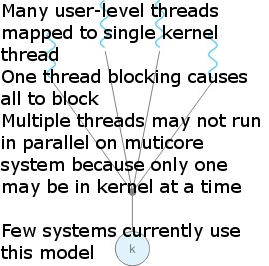
**User threads** - management done by user-level threads library

Three primary thread libraries:

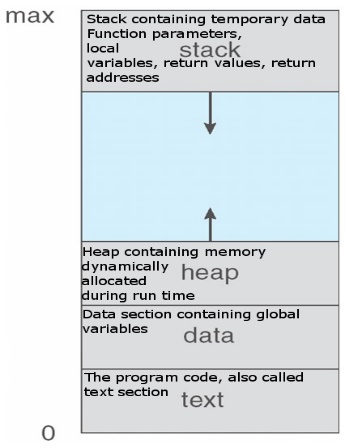
*(POSIX* ***Threads,*** *Windows threads, Java threads)*

**Kernel threads** - Supported by the Kernel

(virtually *all general purpose OS)*



May be provided either as user-level or kernel-level

A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

***[Specification***, not ***implementation]***

[API specifies behavior of the thread library; implementation is up to development of the library]

Common in UNIX operating systems (Solaris, Linux, Mac OS X)

**Implicit Threading**

Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads

Creation and management of threads done by compilers and run-time libraries rather than programmers *(Thread Pools, OpenMP)*

OpenMP = #pragma  
**Context Switching**  
When CPU switches to another process, the system must save the state of the current process and load the saved state for the new process via a context switch

Context of a process represented in the PCB

Context-switch time is overhead; the system does no useful work while switching

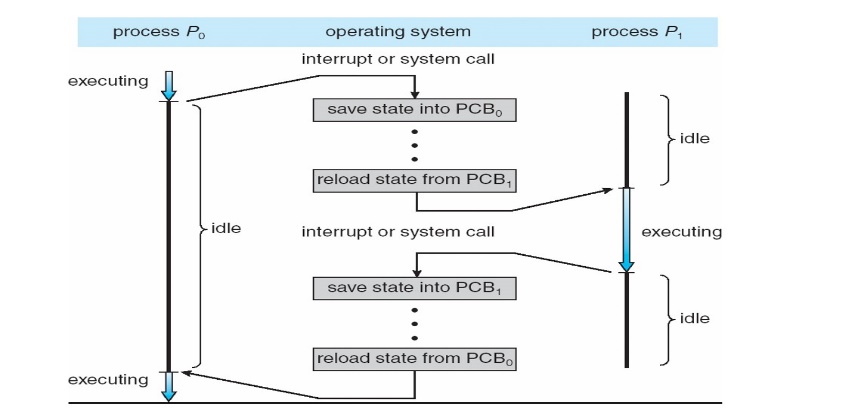
Time depends on OS complexity and design (e.g. memory management techniques)

Time dependents on hardware support

Memory speed, number of registers, existence of special instructions, such as a single instruction to load and store all registers  
Some hardware provides multiple sets of registers per CPU Typical switching speed is a few milliseconds  
Independent process cannot affect or be affected by the execution of another process

Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation

*****(Information sharing, Computation speed-up, Modularity, Convenience)*

**new**: The process is being created

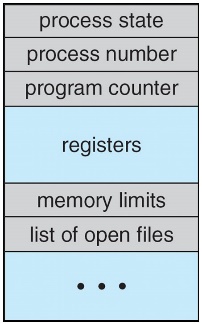
**running**: Instructions are being executed on CPU

**waiting**: The process is waiting for some event to occur (such as I/O completion, semaphore, message, child termination)

**ready**: The process is waiting to be assigned to a processor

**terminated**: The process has finished execution

(also called **task control block**)

**Process state** – running, waiting, etc.

**Program counter** – address of next instruction to execute

**CPU registers** – contents of all **CPU registers**: general purpose, stack reg, ...

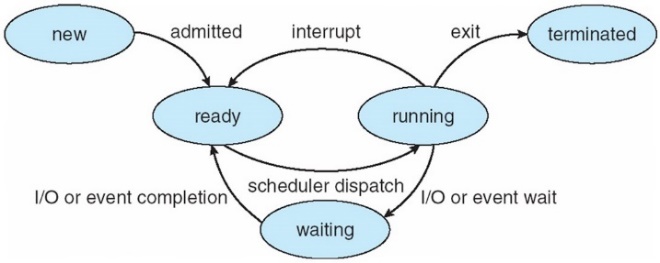
CPU scheduling information- priority, scheduling queue pointers

**Memory-management information** – memory allocated to the process

**Accounting information** – CPU time used, clock time elapsed since start, time limits

**I/O status information** – I/O devices allocated to process, list of open files

Maximize CPU use, quickly switch processes onto CPU for time sharing

**Scheduling**

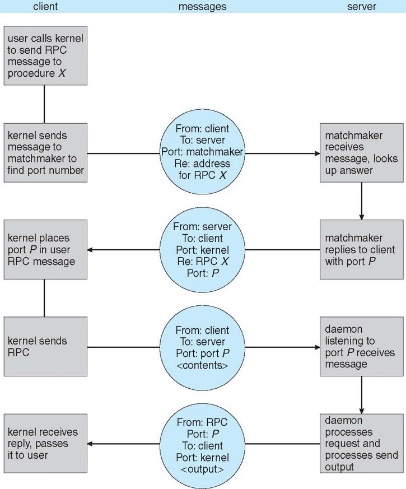
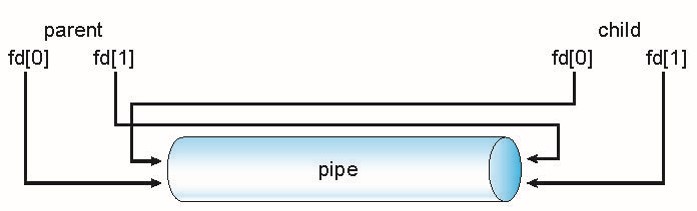
**Process scheduler** selects among available processes for next execution on CPU

Maintains **scheduling queues** of processes

**Job queue** – list of all processes in the system

**Ready queue** – processes residing in main memory, ready to execute.bData structure depends on the scheduling algorithm

**Device queues** – processes waiting for an I/O devicebEach device has its own queue Processes migrate among the various queues

Data representation handled via **External Data Representation** (**XDL**) format to account for different architectures  
**Big-endian** vs **little-endian**.  
Remote communication has more failure scenarios than local Messages can be delivered ***exactly once*** rather than ***at most once*** OS typically provides a **matchmaker**) service to connect client and server  
**Pipes**  
Acts as a conduit allowing two processes to communicate  
Issues:  
Is communication unidirectional or bidirectional?  
In the case of two-way communication, is it half or full-duplex?  
Must there exist a relationship (i.e., parent-child) between the communicating processes?  
Can the pipes be used over a network?  
Ordinary pipes – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.   
Named pipes – can be accessed without a parent-child relationship  
Ordinary Pipes allow communication in standard producer-consumer style  
Producer writes to one end (the write-end of the pipe)  
Consumer reads from the other end (the read-end of the pipe)  
Ordinary pipes are therefore unidirectional  
Constructed using pipe(int fd[]);  
fd[0] is the read end and fd[1] is the write end  
Require parent-child relationship between communicating processes  
Windows calls these anonymous pipes  
Named Pipes are more powerful than ordinary pipes  
Communication is bidirectional  
No parent-child relationship is necessary between the communicating processes  
Several processes can use the named pipe for communication  
Provided on both UNIX and Windows systems  
**Scheduling**  
Message passing may be either blocking or non-blocking  
Blocking is considered synchronous  
Blocking send -- the sender is blocked until the message is received  
Blocking receive -- the receiver is blocked until a message is available  
Non-blocking is considered asynchronous  
Non-blocking send -- the sender sends the message and continues  
Non-blocking receive -- the receiver receives:  
 A valid message, or Null message  
Different combinations possible  
If both send and receive are blocking, we have a rendezvous, and the solution to the producer-consumer problem becomes trivial  
**Message Passing**If processes P and Q wish to communicate, they need to:  
Establish a communication link between them  
Exchange messages via send/receive  
Implementation of communication link  
Physical *(Shared memory, Hardware bus, Network )*  
Logical\* *( Direct or indirect, Synchronous or asynchronous, Automatic or explicit buffering )***Synchronization Hardware**:  
Many systems, provide hardware support for implementing the critical section.  
Critical section is the section of the code that may cause deadlocks or starvation. Usually because of a shared variable.  
Locking. This is a concept that hardware uses to prevent interrupts from happening.  
However, this may be inconvenient or inefficient.  
Another concept to fix this could be the use of atomic operations. For example:  
"This right here will be used as an atomic function. That means it cannot stop halfway thru."  
**"test\_and\_set"**   
*bool test\_and\_set(bool \*target)*{ Boolean rv = \*target; \*target = TRUE; return rv; }  
***USE*** *do {* while(test\_and\_set(&lock)){ ; }/\*\* Critical section code \*/  
lock = false; /\*\* Other section code \*/ *} while(true);***"compare\_and\_swap"** //This is set at 0 no process is active  
*int compare\_and\_swap(int \*value, int expect, int new\_value)*{  
int temp = \*value; *if(\*value == expect){* \*value = new-value;  
return temp;}  
**USE** *do { while(compare\_and\_swap(&lock,0,1) != 0){* ; } /\*\* Critical section code \*/ lock = false; /\*\* Other section code \*/ } *while(true);*These however, do not satisfy the bounded waiting. *(Progress, Mutual Exclusion, Bounded Waiting)*Time scheduling and time quanta are the factors that do not guarantee the bounded waiting problem.  
We cannot assume the time scheduling is fair and enforces fairness.  
It is in regards in turn based waiting. Waiting[h]  
waiting[i] = true, // i is waiting there is a lock in place at this point.  
After the process has gained access to its critical section it sets the waiting to false.  
Now it has to search all processes and check if it is waiting for the process. If a process is waiting it is set as false, and since it will be stuck in the while loop, it can now pass the while loop and access its critical section.  
**Mutex Locks**Hardware solutions use is complex and hard to use directly. A simpler usage is required. OS's use Mutex locks and semaphores. the main functions are acquire() and release()  
We will be implementing this using pthreads.  
pthread\_lock and pthread\_unlock  
When working with these latter functions.  
pthread\_lock  
 access shared var / critical section  
pthread\_unlock  
This style is called a spinlock. While it is waiting...it is "spinning" and just waiting.  
A waiting process blocks (waiting state)  
**Semaphore**semaphore s=3  
Number of available instances of a resource.  
Wait -> if the number of instances <= 0  
signal/post -> make one more instance available s++  
**Buffering**Queue of messages attached to the link.  
implemented in one of three ways  
1. Zero capacity – no messages are queued on a link.  
Sender must block until the recipient receives the message (rendezvous)  
2. Bounded capacity – finite length of n messages  
Sender must wait if and only if link is full  
3. Unbounded capacity – infinite length   
Sender never waits