Object files contains machine code not necessarily binary.
Linking connects the functions definitions to the locations of the actual functions. Linking takes more than one object file making a single executable. Complains if the function is undefined. Linking can find the functions between different files where the compiler can't.

Consistency of data through caches memory and disk.
write through: updates through each level; slower but easier to implement, more reliable.
Write back: only updates through different levels when data is evicted, faster harder to implement, less reliable.
Spacial locality : things near this memory will be called again
Temporal locality : something called will be called again

Kernel : creation, destruction, schedule run by kernel, one blocked other threads are ok (good for IO intensive), slower (has to access lower level kernel)
User : run by user,if blocked all other threads blocked, faster (more control of schedule)

Statically:Precompiled with all the functions
Advantage : Simple to implement,reliable
Disadvantages : All the functions loaded at once , bad for security
Dynamically: Not all loaded into memory
Advantage : not all are loaded at once, can limit which ones, good for security
Disadvantage : not simple to implement

Atomicity : Executes uninterrupted by anything. pthread_cond_wait() is important to be atomic. It unlocks the mutex and waits for a signal to be sent. It is important its atomic because those two must happen because it could miss a signal or because it could miss a wait.

Multiprogramming (Forking creating another identical process except pid) returns the id of the new process
•anything that calls it what results is a child process with the everything exactly the same; what is different is that the child process gets its own process ID
•One process calls fork, it returns to both parent and child though
•if it returns -1, it fails
•if it returns 0, we are in the child process
•if it returns a number larger than 0, it is returning the process id of the child process
•keeps All file descriptors
•Child inherits signal dispositions
•Pending signals of the parent process are not pending in the child
•parent and child can do different things
•Any process can split into a perfect duplicate. Parent can wait for any of the child processes to return. If the parent doesn't wait, it's a zombie process. Child is formed but can't do anything.
- orphan: parent die before child, and init (parent of all processes) becomes the new parent, the child process is an orphan
pid_t fork();
getppid = get parent id
pid_t wait(int * status); // Tells me the FIRST of any one of the children that stopped
pid_t waitpid(pid_t, int * status, int options); //wait for a specific child

Exec (executes the process and gives return value, stops the program at the exec and starts the function not completing the calling function)
• use exec to make a child process execute a new program after it has been forked.
• Normally does not return, -1 on failure
• Fail when: too big, ACCESS, get into loop, name too long, exe doesn't exist
• New process keeps: set of blocked signals, pending signals,timer, any open file descriptors
• doing an exec does not change the relationship between a parent and a child process
• caught signals return to default values

```
// Signal handler demonstration.  Handles user-caused SIGINT, prints blocked signals.
static void signal_handler( int signo ) {
    printf( "Signal handler invoked.  Delivered signal is %s.\n", _sys_siglist[signo] );
    printBlocked( "signal handler" );
}
int main(){
              struct sigaction        action;
              printf( "main() invoked in process %d.\n", getpid() );
              printBlocked( "main()" );
              action.sa_flags = 0;
              action.sa_handler = signal_handler; /* short form */
              sigemptyset( &action.sa_mask ); /* no additional signals blocked */
              sigaction( SIGINT, &action, 0 );
              pause();     /* wait for a signal.  any signal. */
              printBlocked( "back in main()" );
              printf( "Normal end.\n" );
              return 0;
}
```

Events
•something significant to the program
•take non-zero time
•asynchronous
•there can be many different types of events: for each type, we need a different handler
•single event loop/queue: everything goes through one queue no matter how complicated the program is; add asynchronous events to queue as they occur
•event handler adds more events to queue
•do our best to handle events in order they occur
•we always have the possibility to lose events
Test and Set : An atomic instruction that writes to disk.
call something from C, save from conflict; atomic operation solves racing issues, but still losing things
BUSY WAIT--wait until the queue is available(lock)

Signal Handlers are asynronous and can happen any time
One signal handler can hancle multiple signals
They are handled by the operating system
SAFE signal handler
asynchronous-signal-safe: operations guaranteed not to interfere with operations that are being interrupted.
set/read global flag (value)
use boolean flag to protect data structure, access
use local variables/parameters passed to signal handlers
call other reentrant functions
reentrant functions: if it can be interrupted in the middle of its execution and then safely called again ("re-entered") before its previous invocations complete execution
post to semaphore

UNSAFE signal handler
use global data structure
malloc or free:internal struct might be inconsistent
no buffered io (fread)
create, exit, join; cancel threads, especially user-level threads
do not lock mutex: signal handlers can be invoked during a thread, but they don't know which one; might create deadlock
do not unlock mutex: might happen in thread that locked (because data was inconsistent) the mutex => inconsistent state as a result
don't unlock someone else's mutex

Threads are cheaper than making a process. all threads in the process share the same address space.
every thread has its own: set of registers,call stack,errno, threadID
· no idea of parent thread id
· signals happen to processes, not threads
· each thread has its own set of registries, like in multipro-cessing; swapped in and out on machine with single processor(cheaper to swap out threads than entire processes)
· advantages: responsiveness: program can keep going and doing other things even if a thread is blocked by IO; shared resources
Why don't you kill threads : This means sockets and files won't be closed, dynamically-allocated memory will not be freed, mutexes and semaphores won't be released, etc. Killing a thread is almost guaranteed to cause resource leaks and deadlocks.

void pthread_init (void)
int pthread_create(pthread_t *thread, NULL,void *(*fntptr) (void *), void *arg)
int pthread_detach (pthread_t * thread_ptr)
int pthread_join (pthread_t thread , any_t * status);
void pthread_exit (any_t status);

int pthread_cond_broadcast (pthread_cond_t * cond);
int pthread_cond_destroy (pthread_cond_t * cond);
int pthread_cond_init (pthread_cond_t * cond , NULL);
int pthread_cond_signal (pthread_cond_t * cond);
int pthread_cond_wait (pthread_cond_t * cond , pthread_mutex_t * mutex);

int pthread_mutex_init (pthread_mutex_t * mutex , pthread_mutexattr_t * attr);
int pthread_mutex_destroy (pthread_mutex_t * mutex);
int pthread_mutex_lock (pthread_mutex_t * mutex);
int pthread_mutex_trylock (pthread_mutex_t * mutex);
int pthread_mutex_unlock (pthread_mutex_t * mutex);

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);
int sem_getvalue(sem_t *sem, int *valp);

Bash syntax :
$$ = PID
$# = Number of arguments
$? = return value of last command called
$1 = return the first argument of script
$0 = name of script called
$@ = all args except 0
$* = all args
m>n = m file Descriptor, n FileName
m>&n = both file descriptors
1>&2 = redirect stdout to standard error
&> = redirect both to a certain file
command >> file = append to file
command<file = contents of file to stdin
grep search-word <filename
&&: if left fails, stop, else return 0 and do right
|| : if left succeeds don't do right
grep [options] regex files...: returns files matching regex; -i: ignore case; -v: print unmatched; -l print filenames not lines;
ls -a(all) -l(long form)
find path_name expr
expr can be -name, -type, -exec cmd {}\; , -print, -depth
ps -e(all process IDs on machine) -u(all IDs on user)
sh :invokes another shell

```
// gcc -g -o thread3 thread3.c -lpthread
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int value = 0;
void * worker( void * ignore ){
    int i,  x;
    for ( i = 0 ; i < 1000 ; i++ ){
         pthread_mutex_lock( &mutex );
           x = value;
           sched_yield();
           value = x + 1;
           pthread_mutex_unlock( &mutex );
    }
    printf( "Thread %d ends, value is %d.\n", pthread_self(), value );
    return 0;
}
int main(){
   pthread_t tid; int i;
    printf( "Cogito ergo sum.\n" );
    pthread_mutex_init( &mutex, NULL );
    for ( i = 0 ; i < 2 ; i++ ){
           pthread_create( &tid, 0, worker, 0 );
    }
    pthread_exit( 0 );
}
```

Events
Main Event Processing Loop

```
for (;;)
{
    sem_wait(&pullsem)
    event = pull(&queue)
    handle(event)
    discard(event)
}
```

Asynchronous Event Response

```
while( testandset(&notAvailable)
!=0)
{
    push(&queue, event)
    notAvailable = 0
    sem_post(&pullsem)
}
```

BUT THIS IS NOT EFFICIENT busy wait this is probably still better though

If we have more queues, we get a lot more reliability:

```
for(i=0;i<4;i++)
{
  initEmpty(&queue[i])
  notAvailable[i]=0
}
sem_init(&pullsem,0,0)
//only implement semaphore once
still
```

```
int count = 0, input, *numbers=NULL0, *more_numbers;
do{
    scanf("%d" , input);
    count ++;
    more_numbers = realloc(numbers, count*sizeof(int));
    if (numbers !=NULL) {
        numbers = more_numbers;
        numbers[count-1] = input;
    } else {
        free(numbers,more);
    }
} while(input != 0);
```

```
new=/tmp/busybody1.$$           #$$ is the process
id of the script
old=/tmp/busybody2.$$
touch $old                      #make empty file

trap "echo cleaning up ; rm -rf $old $new ; exit 0"
1 2 9
echo trap returned $?

while:                  #while true
do
    who > $new          #who = last login
    diff $old $new
    mv $new $old        #move new file to old file
    sleep 20
done
```

Producer/Consumer

```
void *consumer( void * arg ){
struct SharedData *d = (struct SharedData *)arg;
char buffer[200],  *      p2;
int i;
pthread_detach( pthread_self() );
while ( d->isopen ){
    pthread_mutex_lock( &d->mutex );
    while ( d->count == 0 ){
        pthread_cond_signal( &d->notFull );
        printf( "BKR consumer() waits when shared buffer is empty.\n" );
        pthread_cond_wait( &d->notEmpty, &d->mutex );
    }
    sleep( 2 );                              // pretend to do something
    printf( "BKR consumer() now takes %d bytes from buffer.\n", d->count );
    p2 = buffer;
    for ( i = 0 ; (d->count > 0) && (i < sizeof(buffer) ) ; i++ ){
        p2[i] = d->buf[d->front];
        d->front = (d->front + 1) % d->bufsize;
        --d->count;
    }
    printf( "%s\n", p2 );
    pthread_cond_signal( &d->notFull );
    pthread_mutex_unlock( &d->mutex );
    }
return 0;
}
void *producer( void * arg ){
    struct SharedData * d = (struct SharedData *)arg;
    int i, limit, back;
    char p2[100];
    while ( printf( "Enter something->" ), scanf( " %[^\n]", p2 ) > 0 ){
        pthread_mutex_lock( &d->mutex );
        while ( d->count == d->bufsize ){
              pthread_cond_signal( &d->notEmpty );
              printf( "BKR producer() waits when shared buffer is full.\n" );
              pthread_cond_wait( &d->notFull, &d->mutex );
        }
        limit = strlen( p2 );
        for ( i = 0 ; i < limit ; i++ ){
            if ( d->count < d->bufsize ){
              back = (d->front + d->count) % d->bufsize;
              d->buf[back] = p2[i];
              ++d->count;
            }
            else{
              pthread_cond_signal( &d->notEmpty );
              printf( "BKR producer() waits when shared buffer is full.\n" );
              pthread_cond_wait( &d->notFull, &d->mutex );
              }
        }
        pthread_cond_signal( &d->notEmpty );
        pthread_mutex_unlock( &d->mutex );
    }
    d->isopen = 0;                        // is this bad?
    return 0;
}
```

```
case $# in #counting number of arguments
0)    set 'date' ; m=$2 y=$6         ;; #get 2nd and 6th arg from date
1)    m=$1 ; set 'date' ; y=$6       ;; #one arg passed
*)     m=$1 ; y=$2       ;; #anything more than 1 arguments; default
esac

case $m in
jan*|Jan*|JAN*) m=1   ;; #anything that looks like January, make m = 1
...
```

```
//execl() envarg test program
int main(int argc, char ** argv)
{
    printf ("Process %s PID %d invoking envarg\n", argv[0], getpid());
    execl("/grad/users/morbius/cs214/envarg", "envarg", 0);
    return 0;
}
int main(int argc, char ** argv)
{
    pid_t pid;
    switch(pid = fork())
    {
        case -1: /*error case*/
              printf("fork() failed -- errno %s]n", strerror(errno));
              break;
        case 0: /*child*/
              printf("process %s PID %d invoking envarg\n", argv[0], getpid());
              execl("/grad/grad_users/morbius/cs214/envarg", "envarg", 0);
              break;
        default: /*process id of child*/
              printf("parent PID %d, waiting for child %d\n", getpid(), pid);
              pid = wait(0); /*returns process id of child*/
              printf("parent PID %d, child %d exited.\n", getpid(),pid);
              return 0;
    }
return 0;
}
```