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);

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.

2 policies are **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.

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)

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 multiprocessing; 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

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

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);

Things to do in 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
  + - The interruption could be caused by an internal action such as a jump or call, or by an external action such as a hardware interrupt or signal. Once the reentered invocation completes, the previous invocations will resume correct execution.
* (special case) only safe multithreading, asynchronous-signal-safe (aka semaphore)

Things we cant do in signal handler

* attempt to use global data structure
* anything with dynamically allocated memory => malloc’ed 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

Setting up a signal : signal(SIGNAME,(void)(\*fn\_ptr))

**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.

**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**

**&> = redirect both to a certain file**

**1>&2 = redirect to standard error**

**grep : looks through files for a specific text and returns those files with that text**

for varName in list ; do

done

if [ ]

fi

takes output of this | puts it here

new = /temp/busybody$$ (set new to /temp/busybody.PID)

trap echo cleaning up ; rm -rf $old $new ; exit “ 0 1 2 9 (If codes 1 2 or 9 are hit do whats in the trap)

echo trap returned $? (print out what trap returned)

while :

do

who > $new (put output of who command into new file)

diff $old $new (Whats the difference between old and new)

mv $new $old (move whats in new into old)

sleep 20 (Sleep of 20 seconds)

done

case $# in (matches the number of arguments to a case)

0) set date ; m=$2 y=$6;;

1) m=$1 ; set ‘date’ ; y=$6 ;;

esac

* Event driven architectures
  + An incident significant to our program, take non-zero time to process
  + Different handlers for events
  + Happen asynchronously
  + Should handle all events in the order they occur
    - add asych. events to queue as they occur
  + not all events will be caught, need to minimize the loss of events

main event processing loop

while(true) {

sem\_wait (&pullsem) //decrement the semaphore

event = pull(&queue) //pull from queue

handle(event) //handle the event

discard(event) //get rid of event

}

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)

Multiprogramming (Forking creating another identical process except pid) returns the id of the new process

(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

Do signal dispositions in a process change after an execlp() calls another process to complete the current process. No unless the additional process specifies a change in the signal. Otherwise it is inherited.

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.

Spacial locality : things near this memory will be called again

Temporal locality : something called will be called again