**Spin()** repeatedly checks the time

and ret once it has run for a second //c2 – 2

**getpid()** returns the process identifier (PID)

进程标识符of the calling process.

PID is unique per running process

(often used by routines that generate

unique temporary filenames.) //c2 - 6

**int fprintf**(FILE \*stream, const char \*format, ...)

sends formatted output to a stream

stream = sequence of data elements (conveyor belt)

//c2 - 7

**int atoi**(const char \*str)

"array to int"

converts the string argument str to

an integer

This function returns the converted

integral number as an int value.

If no valid conversion could be performed,

it returns zero

//c2 - 7

**int pthread\_create**(pthread\_t \*restrict thread,

const pthread\_attr\_t \*restrict attr,

void \*(\*start\_routine)(void \*),

void \*restrict arg);

starts a new thread in the calling process

the new thread starts execution by

invoking start\_routine()

arg is passed as the sole argument of

start\_routine()

//c2 - 7

**int pthread\_join**(pthread\_t thread, void \*\*retval)

waits for the thread specified by *thread* to terminate

if *thread* has already terminated, then **pthread\_join()**

returns immediately //c2 – 7

**pid\_t fork**(void) creates a new process by duplicating

the calling process

new process = *child* process

calling process = *parent* process

child process and parent process run in separate memory

spaces; at time of **fork**() both memory spaces have the

same content. Memory writes performed by one of the

processes do not affect the other

if(success) child process’s PID is returned in the parent,

and 0 is returned in the child

if(failure) -1 is returned in the parent,

no child process is created //c5 - 1

**fork**() geek: fork() syscall creates a new process

(child process), which runs concurrently

with the process that makes the

fork() call (parent process)

after the fork() instruction, 2 process will

run the same program

values returned by fork():

negative: creation of child failed

zero: returned to newly created child process

positive: returned to parent or caller

**pid\_t wait**(int \*wstatus) wait for state changes

in a child of the calling process,

and obtain info about the child whose state has changed

state change = child terminated, child stopped by a signal,

or child was resumed by a signal

in case of a terminated child, performing a wait

allows the system to release the resources associated

with the child; if wait is not performed, then the

terminated child remains in a “zombie” state

(zombie process = a process that has

completed execution but still has an entry

in the process table) //c5 - 4

**wait**() geek: blocks the calling process until its child

process exits or a signal is received

after child process terminates, parent continues its

execution after wait syscall

child process terminate due to:

* It calls exit()
* It returns (an int) from main
* It receives a signal (from OS or another process)

whose default action is to terminate //c5 - 4

**exec**() family of functions replaces the current process

image with a new process image

(process image = an executable file required

while executing the program: code, data, stack, heap segments)

(image = an exact replica of the contents of

a storage device) //c5 - 4

**int execve**(const char \*pathname, char \*const argv[],

char \*const envp[])

//*envp* is array of pointers to strings

executes the program referred to by pathname

causes the program that is currently being run by

the calling process to be replaced with a new program,

with newly initialized stack, heap, and data segments

geek:

exec family can be used to run a C program using another

C program

**int execvp**(const char \*file, char const argv[]), using execvp,

created child does not have to run the same program as the

parent process does

a successful call to exec() never returns

**char \*strdup**(const char \*s) duplicate a string,

returns a pointer to a new string which is a

duplicate of the string s

//c5 – 5

**int pipe**(int pipefd[2]) creates a pipe,

a unidirectional data channel that can be

used for interprocess communication

array *pipefd* is used to return two file

descriptors:

pipefd[0]: read end of pipe

pipefd[1]: write end of pipe

//c5 - 7

pipe() geek: connection between 2 processes,

ex: standard output from one process becomes

the standard input of the other process

fd = file descriptor

**grep** – print lines that match patterns

searches a file for a particular pattern and

displays all lines that contain that pattern

grep [options] pattern [files]

//c30 – 6

**Locks**: an object that can only be owned by

a single thread at any given time

**acquire**: mark the lock as owned by the current thread;

if some other thread already owns the lock,

then first wait until the lock is free.

Lock typically includes a queue to keep track of

multiple waiting threads

**release**: mark the lock as free

(the lock must currently be owned by the calling thread)

<https://web.stanford.edu/~ouster/cgi-bin/cs140-spring14/lecture.php?topic=locks>

//c28