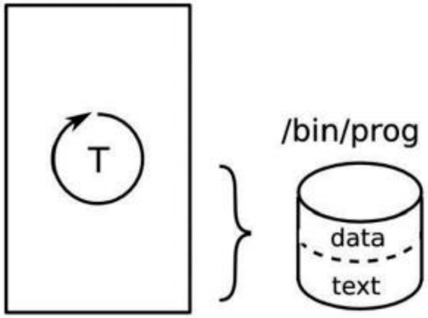
EMBEDDED OPERATING SYSTEMS

Embedded Linux on Beaglebone Black

Process



- A process is a combination of
  - Memory address space
  - Thread of execution
- Address space is private to the process
  - Other processes / threads cannot access it
  - Created by memory management subsystem in kernel
- Process has resources
  - Memory (stack, heap), file descriptors, etc.
- Kernel reclaims all resources when process ends
- Processes communicate among themselves using IPC
  - Inter-process communication

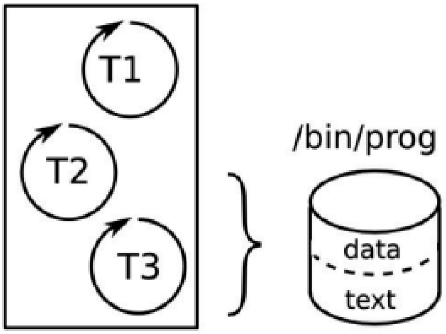
Thread

- A thread is an executional entity within a process

- All processes begin with *main()* thread
  - They may create more threads using **pthread** APIs

- Multiple threads in same process share
  - Same memory address space
  - Other resources like file descriptors

- Communication is easier
  - Needs to be properly synchronized

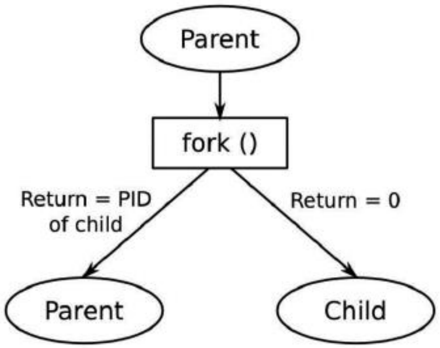


Creating a process: fork()

- Processes are created using the ***fork()*** call

  #include <sys/types.h>
  #include <unistd.h>
  pid_t fork(void);

  - Here pid_t is the Process ID



- But fork() is a strange function
  - 2 returns on success
    - First return is 0 in the new process created (child)
    - Second return is a number in the original process (parent)
- Child is an exact copy of parent
  - But in separate address spaces!

Terminating a process: exit()

- Process terminates when
  - It calls exit() function – ***voluntary***
  - It gets a signal (like SIGKILL) – *involuntary (is killed)*

- Return value of terminating process is either
  - Argument to exit() call
  - Signal value if killed

- Return value can be collected by parent
  - Using *wait()* or *waitpid()* calls
  - Kernel can notify parent using a SIGCHLD

- If parent is killed / not waiting
  - Child process becomes a zombie
  - Attached to init / kernel

Forking exercise

- *forking.c*
  - Notice how the code is written
    - Common codebase for child and parent
    - How child / parent process can be distinguished
      - Trapping the return value of the fork() call
    - How child process exits with an exit code
    - How parent can wait for child termination
      - And get access to exit code returned by child

Forking different program: exec()

- What if we want a fork() but want to run something else, instead of parent code?
  - Use the exec() family of functions
    - Get ***location*** of the executable as argument
      - *execl()*
      - *execlp()*
      - *execle()*
    - Get a ***vector*** of arguments
      - *execv()*
      - *execvp()*
      - *execve()*

The exec() family

- *int execl(const char *pathname, const char *arg, ..., NULL);*
- *int execlp(const char *file, const char *arg, ..., NULL);*
- *int execle(const char *pathname, const char *arg, ..., NULL, char *const envp[]);*

- *int execv(const char *pathname, char *const argv[]);*
- *int execvp(const char *file, char *const argv[]);*
- *int execve(const char *pathname, char *const argv[], char *const envp[]);*

exec() exercise

- Use the shell script (run_all_execs.sh)
  - To run all types of exec() calls
- Note the different calls
  - Input parameters and usage!
- Terminology
  - *l ➔ location*
  - *v ➔ vector*
  - *p ➔ path*
  - *e ➔ environment (shell)*

Inter-Process Communication

- Each process is an island of memory

- Inter-Process Communication (IPC) happens
  - By copying info from one address space to another
    - Using a queue / buffer to hold and pass messages
      - Examples: **sockets, pipes, message queues**
    - Using a **shared file** and **file locks**
  - By creating a memory area that all can access
    - Using shared memory; needs access synchronization
      - Examples: **shmem**

- Message and shared file based methods are easier to program and debug
  - But slow for large number of messages
- Shared memory based methods are more efficient
  - But need careful synchronization

IPC using shared files, file locks

- Writer and reader process
  - Share a common file on file system (file.dat)

- Either operation needs a file lock (***flock***)

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

- Lock types:
  - Writer needs a F_WRLCK (exclusive)
  - Reader needs a F_RDLCK (shared / non-exclusive)
  - After the operation, process needs to F_UNLCK

File locking using fcntl()

- Locks are obtained using ***fcntl()***

  #include <unistd.h>

  #include <fcntl.h>

  int fcntl(int fd, int cmd, ... / arg */);*

- Commands for fcntl (*cmd*):
  - GETLK
    - Get the current lock; returns immediately
  - SETLK
    - Set the lock (3^{rd} argument is &lock); no waiting
  - SETLKW
    - Set the lock (3^{rd} argument is &lock), wait until you get the lock

File lock exercise

- *flock-writer.c*
  - Implements the writer
  - Creates the file, and takes a write lock (F_WRLCK) – with wait
  - Writes data to file
  - Exits after releasing the file lock (F_UNLCK)

- *flock-reader.c*
  - Implements the reader
  - Takes a read lock (F_RDLCK) on the file
    - Waits till it gets it!
  - Reads data from file and displays on stdout
  - Exits after releasing the lock (F_UNLCK)

- Run *flock-writer.out* and *flock-reader.out* on different terminals at once
- Try running 2 writer processes (*flock-writer.out*) at once

IPC using shared memory

- POSIX API for sharing memory
  - *shm_open()*
    - Create/open a new/existing shared memory object
  - *ftruncate()*
    - Set the size of the shared memory object (viewed as fd)
  - *mmap() / munmap()*
    - Map/unmap devices or files into process memory space
  - *shm_unlink()*
    - Unlink (delete) share memory object

- POSIX maintains a backing file for shared memory
  - Located in /dev/shm/…

- We need to use primitives like *semaphore/mutex*
  - For controlling exclusive access to memory object

- Linking flags: ***-lrt -lpthread***

POSIX shared memory API

- Open/create and close

  #include <sys/mman.h>
  #include <sys/stat.h>   / For mode constants */*
  #include <fcntl.h>   / For O_* constants */*
  *int **shm_open**(const char *name, int oflag, mode_t mode);*
  *int **shm_unlink**(const char *name);*

- Set size

  #include <unistd.h>
  #include <sys/types.h>
  *int **ftruncate**(int fd, off_t length);*

- Memory map/unmap

  *void ***mmap**(void *addr, size_t length, int prot, int flags, int fd, off_t offset);*
  *int **munmap**(void *addr, size_t length);*

POSIX semaphore API

- Open/create

  #include <fcntl.h> / For O_* constants */*
  #include <sys/stat.h> / For mode constants */*
  #include <semaphore.h>
  *sem_t ****sem_open***(const char *name, int oflag, mode_t mode, unsigned int value);*

  - Semaphore mode
    - **O_CREAT | O_RDWR** for writer
    - **O_RDWR** for reader

- Wait on / Post a semaphore

  int ***sem_wait****(sem_t *sem);*
  int ***sem_post****(sem_t *sem);*

- Close/unlink

  int ***sem_close****(sem_t *sem);*
  int ***sem_unlink****(const char *name);*

Shared memory exercise

- *shmem-writer.c*
  - Implements the writer
  - Opens/creates and mmaps the shared mem segment
    - Sets the size
  - Creates the semaphore
    - Writes to mem segment while holding it
  - Releases (posts) the semaphore
  - Cleans up (semaphore deletion, unlinking backing file, etc.)

- *shmem-reader.c*
  - Implements the reader
  - Opens and mmaps the shared mem segment
  - Tries to take the semaphore
    - Waits till it gets the semaphore
  - Reads from the segment and outputs to stdout
  - Unmaps the mem segment and exits

Threads on Linux

- Linux uses the POSIX threads library
  - Commonly called ***pthread***
- Threads are called 'light-weight processes' (LWP)
  - Share same resources (memory space, file descriptors) as the parent process
- A process starts by creating a single thread
  - Called main thread (from the C ***main()*** entrypoint)
- Inter-thread communication
  - Is simpler and has less overhead
  - But needs to be synchronized and controlled!
- Linking: ***-lpthread***

What threads share (& don't!)

- Threads in a process share:
  - Process ID (PID)
  - Parent Process ID (PPID)
  - User ID (UID) and Group ID (GID)
  - Controlling terminal
  - File descriptors and locks

- Distinct for each thread:
  - Thread ID (TID)
  - Signal mask
  - ***errno*** variable
  - RT scheduling policy and priority

POSIX thread API

- Open/create

  #include <pthread.h>

  *int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);*

- Termination options

  *void **pthread_exit**(void *retval);*

  *int **pthread_cancel**(pthread_t thread);*

  *int **pthread_join**(pthread_t thread, void **retval);*

  - Return from start_routine()

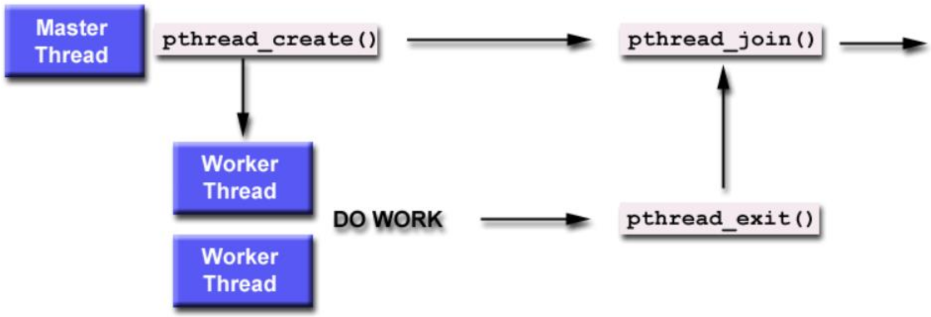
- Thread attributes

  *int **pthread_attr_init**(pthread_attr_t *attr);*

  *int **pthread_attr_destroy**(pthread_attr_t *attr);*

Joinable vs. Detachable threads

- Usual lifecycle of threads



- By default, threads are joinable
  - ***Master*** waits for ***worker*** using ***pthread_join()***
  - ***Worker*** resources not freed up till ***master*** joins, even if ***worker*** exits
- But what if we want to free ***worker*** resources early on?
  - Then ***worker*** can be created ***detachable***

Creating detachable threads

- Create a ***pthread_attr_t attr;***
- Set/get detach state using these APIs:

  *int **pthread_attr_setdetachstate**(pthread_attr_t *attr, int detachstate);*
  *int **pthread_attr_getdetachstate**(const pthread_attr_t *attr, int *detachstate);*

- Detach states:
  - *PTHREAD_CREATE_DETACHED*
  - *PTHREAD_CREATE_JOINABLE*

- Create the thread using this ***attr***

  *int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);*

- A joinable thread can be changed to detached using the API

  *int **pthread_detach**(pthread_t thread);*

Joinable/detached exercise

- Run ***join-detach.out***

- Observe the output
  - When threads are created
  - As threads execute
  - When main tries to join the threads

- Notice that
  - Joinable thread can be joined
  - Detached threads cannot be joined any more

Mutex

- Mutex is an exclusive lock
  - Used for controlling / serializing access
  - To a common (shared) resource
  - Between multiple threads
- Only the thread holding the lock
  - Can access the shared resource
  - It then releases (unlocks) the mutex
- The other thread has to wait till the mutex is free

Conditional variable (condvar)

- Threads need to be able to alert other threads
  - Something has changed / needs attention
  - This is called a condition
  - The alert is sent through a ***conditional variable (condvar)***
- One thread (alerter) signals the other (alertee)
  - By signaling using the ***condvar***
- The other thread sleep-waits on the ***condvar***
  - Waiting for the signal from the 1st thread
- Since both threads use/access the ***condvar*** at one time
  - We need a ***mutex*** to serialize access to the ***condvar***

POSIX mutex and condvar API

- Mutex API
  - Creation

    pthread_mutex_t *mutex = PTHREAD_MUTEX_INITIALIZER;*
  - Lock/unlock

    *int **pthread_mutex_lock**(pthread_mutex_t *mutex);*
    *int **pthread_mutex_unlock**(pthread_mutex_t *mutex);*

- Conditional variable (condvar) API
  - Creation

    pthread_cond_t *cond = PTHREAD_COND_INITIALIZER;*
  - Wait/signal

    *int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex);*
    *int **pthread_cond_signal**(pthread_cond_t *cond);*

Mutex-Condvar exercise

- Run ***mutex-condvar.out***

- Observe
  - Producer and Consumer share a global char buffer
  - Producer enters data into the buffer
    - Signals this to Consumer
  - Consumer waits for signal from Producer
    - Retrieves data from buffer; resets it to “empty”
  - They use a *condvar* and a *mutex*
    - For serialized and controlled access to the *condvar*

THANK YOU!