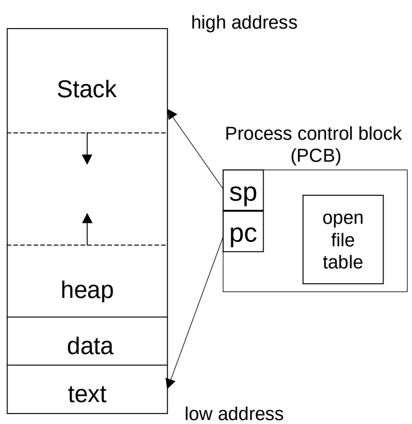
Threads

lightweight processes

Motivation

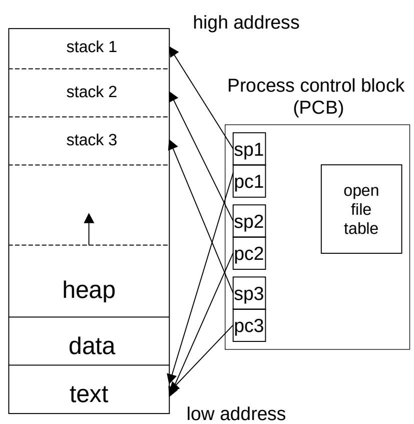
- Processes are expensive to create.
- It takes quite a bit of time to switch between processes
- Communication between processes must be done through an external kernel structure
  - files, pipes, shared memory
- Synchronizing between processes is cumbersome.
- *Is there another model that will solve these problems?*

Processes



- Each process has its own
  - program counter
  - stack
  - stack pointer
  - address space
- Processes may share
  - open files
  - pipes

Threads



- Each thread has its own
  - program counter
  - stack
  - stack pointer
- Threads share
  - address space
    - variables
    - code
  - open files

Advantages

- Communication between threads is cheap
  - they can share variables!
- Threads are “lightweight”
  - faster to create
  - faster to switch between
- Synchronization avoids kernel

Threaded design

- Several common models for threaded programs exist:
  - Manager/worker: a single manager thread assigns work to other threads, the workers. The manager typically handles all input and parcels out work to the workers.
  - Pipeline: a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. Is like an automobile assembly line.
  - Peer: similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Pthreads

- POSIX threads (pthreads) is the most commonly used thread package on Unix/Linux

pthread_create

```
int pthread_create(pthread_t *tid,
        pthread_attr_t *attr,
        void *(*func)(void*), void *arg);
```

- `tid` uniquely identifies a thread within a process and is returned by the function
- `attr` sets attributes such as priority, initial stack size
  - can be specified as NULL to get defaults
- `func` - the function to call to start the thread
  - accepts one `void *` argument, returns `void *`
- `arg` is the argument to func
- returns 0 if successful, a positive error code if not
- does not set `errno` but returns compatible error codes
- can use `strerror()` to print error messages

pthread_join

```
int pthread_join(pthread_t tid,
                 void **status)
```

- `tid` - the tid of the thread to wait for
  - cannot wait for any thread (as in `wait()`)
- `status`, if not NULL returns the `void *` returned by the thread when it terminates.
- a thread can terminate by
  - returning from `func`
  - the `main()` function exiting or `exit()` called
  - `pthread_exit()`
  - `pthread_cancel()`

More functions

- `void pthread_exit(void *status)`
  - a second way to exit, returns `status` explicitly
  - `status` must not point to an object local to the thread, as these disappear when the thread terminates.
- `int pthread_detach(pthread_tid);`
  - if a thread is detached its termination cannot be tracked with `pthread_join()`
  - it becomes a daemon thread
- `pthread_t pthread_self(void)`
  - returns the thread ID of the thread which called it
  - often see `pthread_detach(pthread_self())`

Passing Arguments to Threads

```
pthread_t thread_ID;  int fd, result;
fd = open("afile", O_RDONLY);
result = pthread_create(&thread_ID, NULL,
                myThreadFcn, (void *)&fd);
if(result != 0)
  printf("Error: %s\n", strerror(result));
```

- We can pass any variable (including a structure or array) to our thread function.
- It assumes the thread function knows what type it is.
- This example is **bad** if the main thread alters `fd` later.

Solution

- Use malloc() to create memory for the variable
  - initialize variable's value
  - pass pointer to new memory via pthread_create()
  - thread function releases memory when done.
- Example:

```
typedef struct myArg {
  int fd;
  char name[25];
} MyArg;

int result;
pthread_t thread_ID;
```

Example (cont'd)

```
MyArg *p = (MyArg *)malloc(sizeof(MyArg));
p->fd = fd;  /* assumes fd is defined */
strncpy(p->name, "CSC209", 7);
result = pthread_create(&threadID, NULL,
                  myThreadFcn, (void *)p);
void *myThreadFcn(void *p) {
   MyArg *theArg = (MyArg *) p;
   write(theArg->fd, theArg->name, 7);
   close(theArg->fd);
   free(theArg);
   return NULL;
}
```


Thread-safe functions

- Not all functions can be called from threads
  - many use global/static variables
  - new versions of UNIX have thread-safe replacements like `strtok_r()`
- Safe:
  - `ctime_r(), gmtime_r(), localtime_r(), rand_r(), strtok_r()`
- Not Safe:
  - `ctime(), gmtime(), localtime(), rand(), strtok(), gethostxxx()`
- Could use semaphores to protect access but will generally result in poor performance.

Pthread Mutexes (Semaphores)

```
int pthread_mutex_init(pthread_mutex_t *mp,
      const pthread_mutexattr_t *attr);

int pthread_mutex_lock(pthread_mutex_t *mp);
int pthread_mutex_trylock(pthread_mutex_t *mp);
int pthread_mutex_unlock(pthread_mutex_t *mp);
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

- easier to use than `semget()` and `semop()`
- only the thread that locks a mutex can unlock it
- mutexes often declared as globals

Example

```
pthread_mutex_t myMutex;
int status;

status = pthread_mutex_init(&myMutex, NULL);
if(status != 0)
   printf("Error: %s \n", strerror(status));
pthread_mutex_lock(&myMutex);
/* critical section here */
pthread_mutex_unlock(&myMutex);
status = pthread_mutex_destroy(&myMutex);
if(status != 0)
  printf("Error: %s\n", strerror(status));
```