Synchronization
POSIX Semaphores

- **Unnamed Semaphore**
  - Provides synchronization between threads and between related processes
  - Thread-shared or process-shared
  - Use `sem_init`

- **Named Semaphore**
  - Provides synchronization between unrelated/related processes as well as between threads
  - Kernel persistence
  - System-wide and limited in number
  - Use `sem_open`
POSIX Semaphores

- Data type
  - Semaphore is a variable of type `sem_t`
- Include `<semaphore.h>`
- Semaphore operations
  ```c
  int sem_init(sem_t *sem, int pshared, unsigned value);
  int sem_destroy(sem_t *sem);
  int sem_wait(sem_t *sem);
  int sem_post(sem_t *sem);
  int sem_trywait(sem_t *sem);
  ```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);

- Initialize an unnamed semaphore
- Returns
  - 0 on success
  - -1 on failure, sets errno
- Parameters
  - sem: pointer to target semaphore
  - pshared:
    - 0: only threads of the creating process can use the semaphore
    - Non-0: other processes can use the semaphore
  - value: Initial value of the semaphore

You cannot make a copy of a semaphore variable!!!
Sharing Semaphores

- Sharing semaphores between threads within a process is easy, use `pshared==0`.
- A non-zero `pshared` allows any process that can access the semaphore to use it.
  - Semaphore is shared between processes, and *should be located in a region of shared memory*.
  - Forked children inherit the shared memory mapping of parent; so, a semaphore mapped in shared memory by parent, it is visible to the children.
- Note: unnamed semaphores are not shared across unrelated processes.
sem_init can fail

- On failure
  - sem_init returns -1 and sets errno

<table>
<thead>
<tr>
<th>errno</th>
<th>cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>value exceeds SEM_VALUE_MAX</td>
</tr>
<tr>
<td>ENOSYS</td>
<td>pshared is nonzero, but the system does not support process-shared semaphores</td>
</tr>
</tbody>
</table>

```c
sem_t semA;

if (sem_init(&semA, 0, 1) == -1)
    perror("Failed to initialize semaphore semA");
```
Semaphore Operations

```c
#include <semaphore.h>
int sem_wait(sem_t *sem);
```

- **Lock a semaphore**
  - If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.

- **Returns**
  - 0 on success
  - -1 on failure, sets `errno` (== EINTR if interrupted by a signal)

- **Parameters**
  - `sem`: pointer to target semaphore
#include <semaphore.h>
int sem_post(sem_t *sem);

- Unlock a semaphore
- Returns
  - 0 on success
  - -1 on failure, sets errno (== EINVAL if semaphore doesn’t exist)
- Parameters
  - sem: pointer to target semaphore
Semaphore Operations

```c
#include <semaphore.h>

int sem_trywait(sem_t *sem);
```

- Same as `sem_wait()`, except that if the decrement cannot be immediately performed, then call returns an error (`errno` set to `EAGAIN`) instead of blocking.

- Returns
  - 0 on success
  - -1 on failure, sets `errno` (= `EAGAIN`) if semaphore already locked

- Parameters
  - `sem`: pointer to target semaphore
Semaphore Operations

#include <semaphore.h>
int sem_destroy(sem_t *sem);

- Destroy a semaphore
- Returns
  - 0 on success
  - -1 on failure, sets errno
- Parameters
  - sem: pointer to target semaphore
- Notes
  - Can destroy a sem_t only once
  - Destroying a destroyed semaphore gives undefined results
  - Destroying a semaphore on which a thread is blocked gives undefined results
Back to the counter example: solution with unnamed semaphore

// Unnamed semaphores are not supported by MAC OS. Test it with Linux!
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/mman.h>
#define PAGE 4096

typedef struct COUNTER {
   sem_t mysem;
   int cnt;
} counter;

int main(int argc, char **argv) {

   counter *p;
   int i;

   if ((p = (counter *) mmap( NULL, PAGE,
      PROT_READ | PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == MAP_FAILED) {
      perror("mmap");
      exit(1);
   }
Back to the counter example: solution with unnamed semaphore

```c
/* initialize semaphore */
if (sem_init(&p->mysem, 1, 1) < 0) {
    perror("semaphore initialization");
    exit(1);
}

if (fork() == 0) { /* child process*/
    for (i = 0; i < 50000; i++) {
        while (sem_wait(&p->mysem) < 0)
            if (errno != EINTR)
                exit(1);
        // locked in mutual exclusion
        p->cnt++;
        if (sem_post(&p->mysem) < 0)
            exit(1);
        // lock is released
    }
    exit(0);
}

/* back to parent process */
for (i = 0; i < 50000; i++) {
    while (sem_wait(&p->mysem) < 0)
        if (errno != EINTR)
            exit(1);
    // locked in mutual exclusion
    p->cnt++;
    if (sem_post(&p->mysem) < 0)
        exit(1);
    // lock is released
    wait(NULL);
    printf("parent: final value is %d\n", p->cnt);
    exit(0);
}
```
Some errors are recoverable

```c
/* Parent process */
for (i = 0; i < 50000; i++) {
    while (sem_wait(&p->mysem) < 0)
        if (errno != EINTR)
            exit(1);
    // locked in mutual exclusion
    p->cnt++;
    if (sem_post(&p->mysem) < 0)
        exit(1);
    // lock is released
}
```

**Quiz:** why is `sem_wait` inside a while loop?

A signal can interrupt a thread/process blocked on `sem_wait`; however, the semaphore might still be busy and require further blocking.
Good Practices

```c
sem_t cnt_mutex

int main(void) {
...
/* initialize semaphore */
if( sem_init(&cnt_mutex, 1, 1) < 0) {
    perror("semaphore initialization");
    exit(1);
}
...

/* Clean up the semaphore that we're done with */
result = sem_destroy(&cnt_mutex);
if (result < 0) {
    perror("sem_destroy failed");
    exit(1);
}
}
```

Check for errors on each call

Clean up resources
Why bother checking for errors?

- Without error handling, your code might
  - Crash rather than exiting gracefully
  - Keep working for a while, crash later
  - Sometimes fail randomly, but usually work fine
    - Hard to reproduce: even harder to debug
  - Fail when it might have recovered from the error cleanly! (see EINTR)

- At a minimum, error handling converts a messy failure into a clean failure
  - Program terminates, but you know what caused it to terminate