Synchronization Primitives

- so far we've seen two synchronization primitives
  - *locks* provide mutual exclusion
  - *condition variables* provide an alternative to busy waiting until a condition is satisfied
- *semaphores* can support both applications

Semaphores

```
#include <semaphore.h>
```

- a semaphore holds an integer count
  - represents how many resources are available
  - all operations are atomic

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

  - initialize semaphore *sem* with *value*
    - *pshared* indicates whether semaphore is shared between threads (0) or processes (non-zero)

```
int sem_destroy(sem_t *sem);
```

  - destroy (deallocate) semaphore

Semaphores

```
#include <semaphore.h>
```

- a semaphore holds an integer count
  - represents how many resources are available
  - all operations are atomic

this follows the pthreads library API documentation – the book's alternative (`wait` decrements, blocking when value < 0; `post` increments, unblocking a waiting thread) has the same public semantics

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict abs_timeout);
```

  - *wait* attempts to decrement the value
    - `sem_wait` – if the current value is 0, the calling thread blocks until the decrement is possible
    - `sem_trywait` – if the current value is 0, return an error (`EAGAIN`)
    - `sem_timedwait` – if the current value is 0, block until the value becomes > 0 or the timeout is reached

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

  - *post* increments the value
    - if this makes the value > 0, unblock one of the waiting threads

Locks Using Semaphores

- use a *binary semaphore*
  - initialize to 1
  - `sem_wait` to acquire the lock
  - `sem_post` to release the lock

pthreads documentation describes `wait` as locking the semaphore and `post` as unlocking

- initializing to 1 means the first attempt to acquire the lock succeeds
- since `post` only occurs after a `wait`, the semaphore's value is only ever 1 or 0

Locks Using Semaphores

semaphore is initialized to 1

value reflect book's internal semantics – pthreads version is that the value is 0 with threads waiting

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Ordering Using Semaphores

- one thread waiting for another thread to do something imposes an *ordering* constraint
  - the other thread must do its thing first, before the one waiting can continue
  - e.g. parent waiting for child to end
  - e.g. consumer waiting for producer to put something in the buffer
- use a semaphore
  - initialize to 0
  - `sem_wait` to wait for the thing to happen
  - `sem_post` when the thing happens

Parent Waiting For Child

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists, can run)*	Ready
0	call `sem_wait()`	Run		Ready
-1	`decr sem`	Run		Ready
-1	`(sem<0)→sleep`	Sleep		Ready
-1	*Switch→Child*	Sleep	`child runs`	Run
-1		Sleep	call `sem_post()`	Run
0		Sleep	`inc sem`	Run
0		Ready	`wake(Parent)`	Run
0		Ready	`sem_post()` returns	Run
0		Ready	*Interrupt→Parent*	Ready
0	`sem_wait()` returns	Run		Ready

Val	Parent	State	Child	State
0	`create(Child)`	Run	*(Child exists; can run)*	Ready
0	*Interrupt→Child*	Ready	`child runs`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`inc sem`	Run
1		Ready	`wake(nobody)`	Run
1		Ready	`sem_post()` returns	Run
1	`parent runs`	Run	*Interrupt→Parent*	Ready
1	call `sem_wait()`	Run		Ready
0	`decrement sem`	Run		Ready
0	`(sem≥0)→awake`	Run		Ready
0	`sem_wait()` returns	Run		Ready

two cases –
– parent starts waiting before child finishes
– child finishes before parent starts waiting

Initializing Semaphores

- initialize to 1 to use as a lock
- initialize to 0 to use for ordering
- in general –

> One simple way to think about it, thanks to Perry Kivolowitz, is to consider the number of resources you are willing to give away immediately after initialization.