Inter-Process Communication

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

The OS kernel does a great job of *isolating* processes from each other

If not, programming would be *much harder*!

- all data accessible (read/write) to world
- memory integrity not guaranteed
- control flow unpredictable

But processes are more useful when they can *exchange data* & *interact dynamically*

The original data exchange unit: the *file*

see: BBS, FTP, Napster, BitTorrent

But what about *dynamic* data exchange?

e.g., instant messaging, VOIP, MMOGs

Demo:

shell “pipes”

The kernel *enforces* isolation

… so to perform *inter-process communication* (IPC), must ask kernel for help/assistance

Another role for the kernel: errand boy

Select IPC mechanisms:

1. signals
2. (regular) files
3. shared memory
4. unnamed & named pipes
5. file locks & semaphores
6. sockets

§Common Issues

1. link/endpoint creation
   - naming
   - lookup / registry

2. data transmission

- unidirectional/bidirectional
- single/multi-sender/recipient
- speed/capacity
- message packetizing
- routing

3. data synchronization

- behavior with multiple senders and/or receivers
- control: implicit / explicit / none

4. access control

- mechanism
- granularity

§ Files

in general, regular files are a really lousy mechanism for *dynamic* IPC

- ultra-slow backing store (disk)
- coordinating file positions is tricky

```
int main() {
    pid_t pid1, pid2;
    int fd = open("shared.txt", O_CREAT|O_TRUNC|O_RDWR, 0644);
    if ((pid1 = fork()) == 0) {
        dup2(fd, 1);
        execl("/bin/echo", "/bin/echo", "hello", NULL);
    }
    if ((pid2 = fork()) == 0) {
        dup2(fd, 0);
        execl("/usr/bin/wc", "/usr/bin/wc", "-c", NULL);
    }
    waitpid(pid2, NULL, 0);
}
```

Output?

… it depends …

we won't be considering regular files as a mechanism for (dynamic) IPC

§Shared Memory

simple idea: allow processes to share data stored in memory

i.e., sidestep memory protection

`shm...` APIs:

- file descriptor based
- memory mapped

FD-based API:

```
int shm_open(const char *name, int oflag, mode_t mode);
```

- returns FD for shared memory
- may be mapped to temp file (of `name`)
- persists until explicitly removed!

```
int shm_unlink(const char *name);
```

- explicitly remove shared memory

```
#define SHM_NAME "/myshm"  /* arbitrary shm identifier */


/* writing process */
int shmfd = shm_open(SHM_NAME, O_RDWR|O_CREAT, 0644);
write(shmfd, ...);
```

```
/* reading process */
int shmfd = shm_open(SHM_NAME, O_RDONLY, 0);
char buf[N];
read(shmfd, buf, N);
```

memory-mapped API:

```
int   shmget(key_t key, size_t size, int shmflg);
```

- returns ID for shm of `size`

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- returns (local) pointer to shm given ID

```
int   shmdt(const void *shmaddr);
```

- detach from shm (but still persists)

```
int   shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- manage existing shm object

```c
#define SHM_KEY  0xABCD
#define SHM_SIZE 1024

int shmid = shmget(SHM_KEY,          /* unique system-wide shm key        */
                   SHM_SIZE,         /* size of shm (in bytes)            */
                   IPC_CREAT|0600);  /* IPC_CREAT not needed if already exists */

char *shm = shmat(shmid, NULL, 0);   /* map shm into my address space     */


strcpy(shm, "hello world!");         /* access shm (via pointer)          */

shmdt(shm);                          /* detach from shm (i.e., unmap)     */

shmctl(shmid, IPC_RMID, NULL);       /* remove shm from system            */
```

if shm isn't removed, it persists … and processes have limited IPC resources!

```
int shmid   = shmget(0xABCD, 1024, IPC_CREAT|0600);
...
shmdt(shm_arr);
/* but no shmctl(shmid, IPC_RMID, NULL); */
```

```
$ ipcs -mpb
IPC status from <running system> as of Thu Oct 18 18:13:07 CDT 2012
T ID      KEY        MODE        OWNER GROUP SEGSZ CPID  LPID
m 2162688 0x0000abcd --rw------- lee   staff 1024  44861 44861

$ ipcrm -m 2162688      # deletes the shared memory object
```

shm is the *fastest* form of IPC;

only overhead = process switch
(unavoidable anyway)

```
#define SHM_KEY  0xABCD
#define SHM_SIZE 5
```

```
int shmid, i;
char *shm_arr;

if ((pid = fork()) != 0) {
    shmid   = shmget(SHM_KEY, SHM_SIZE, IPC_CREAT|0600);
    shm_arr = shmat(shmid, NULL, 0);
    memset(shm_arr, 0, SHM_SIZE);
    for (i=0; i<SHM_SIZE; i++) {
        shm_arr[i] = i;
    }
    shmdt(shm_arr);
    shmctl(shmid, IPC_RMID, NULL);
} else {
    shmid = shmget(SHM_KEY, SHM_SIZE, 0600);
    shm_arr = shmat(shmid, NULL, 0);
    for (i=0; i<SHM_SIZE; i++) {
        printf("%d ", shm_arr[i]);
    }
    shmdt(shm_arr);
}
```

```
0 1 2 3 4
```

```
0 0 0 0 0
```

```
0 1 0 0 0
```

```
0 1 2 0 0
```

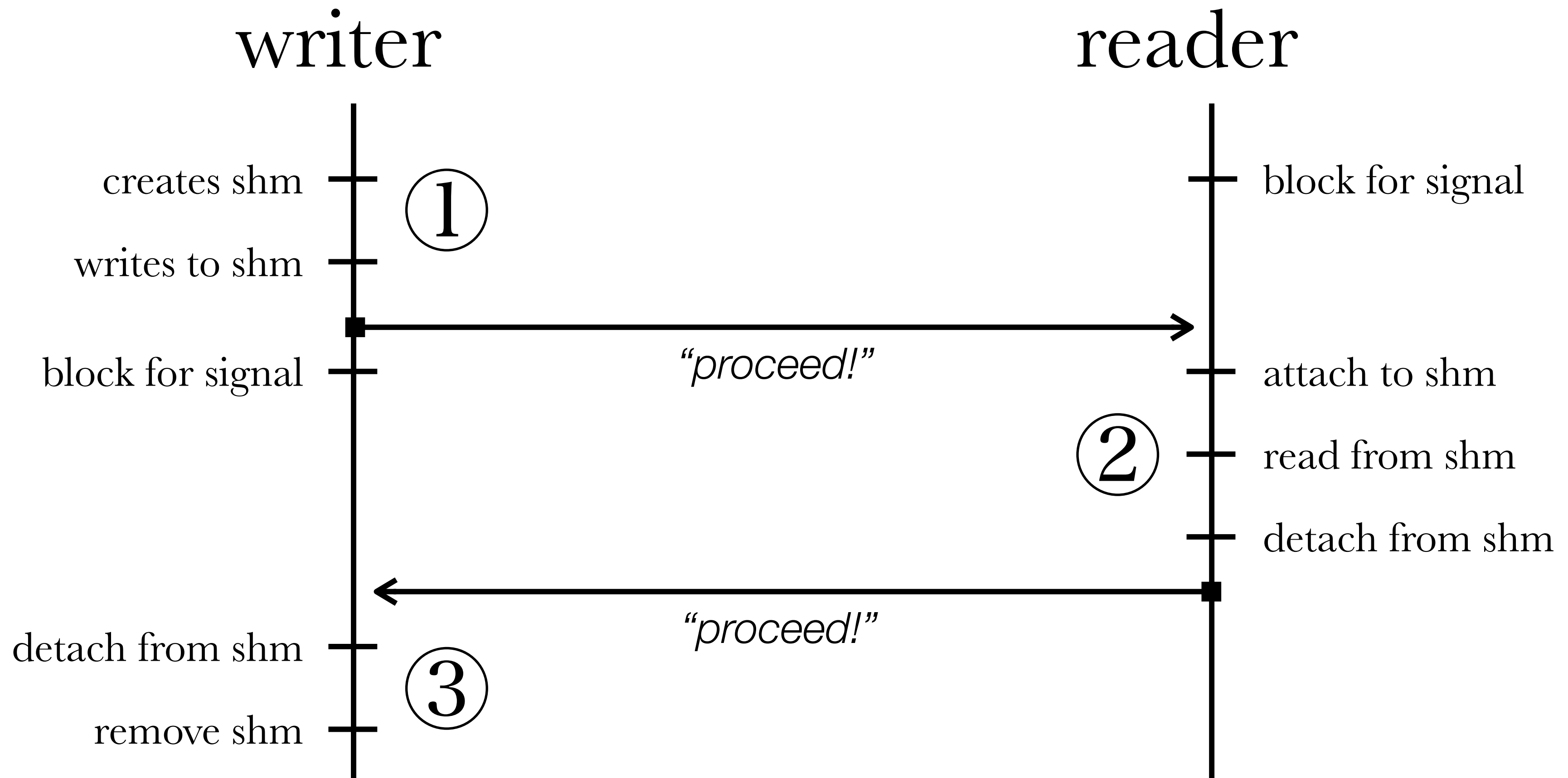
…

```
(error – no output)
```

to fix, we need processes using shared memory to communicate

… using another IPC mechanism!

one approach: signals

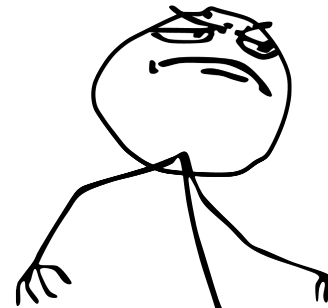


```c
int sig_recvd = 0;
void sighandler (int sig)
{
    if (sig == SIGUSR1)
        sig_recvd = 1;
}

int main (int argc, char *argv[])
{
    signal(SIGUSR1, sighandler);
```

```c
/* parent/writer process */
if ((pid = fork()) != 0) {
    shmid   = shmget(SHM_KEY, ..., IPC_CREAT|...);
    shm_arr = shmat(shmid, ...);

    for (i=0; i<SHM_SIZE; i++) {          ①
        shm_arr[i] = i;
    }

    kill(pid, SIGUSR1); /* signal child */

    while (!sig_recvd) /* block for child signal */
        sleep(1);

    shmdt(shm_arr);                        ③
    shmctl(shmid, IPC_RMID, NULL);
}
```

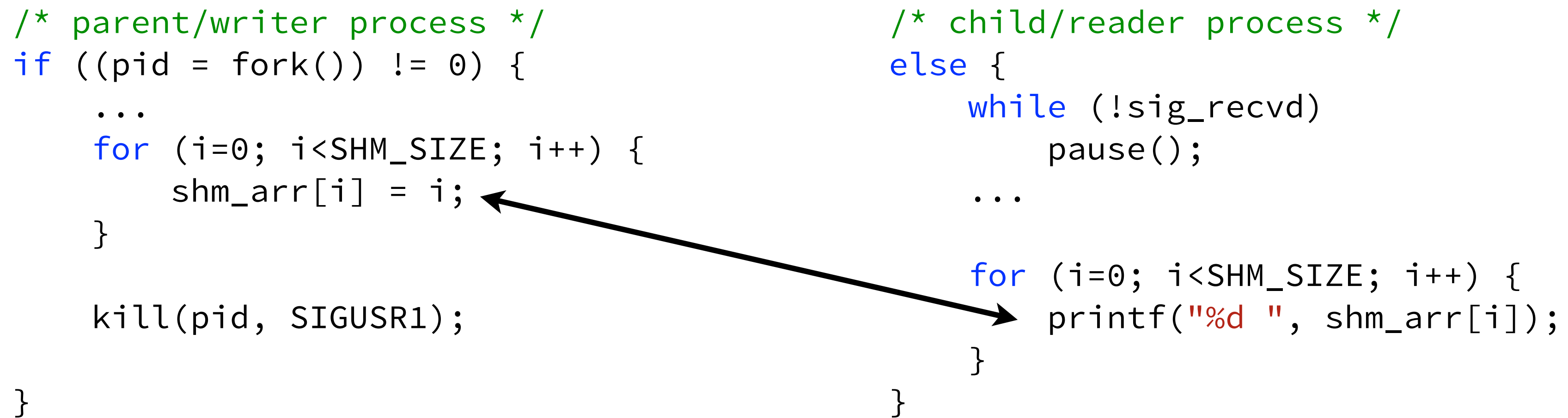
```c
/* child/reader process */
else {
    while (!sig_recvd) /* block for parent signal */
        sleep(1);

    shmid = shmget(SHM_KEY, ...);

    shm_arr = shmat(shmid, ...);           ②

    for (i=0; i<SHM_SIZE; i++) {
        printf("%d ", shm_arr[i]);
    }

    shmdt(shm_arr);
    kill(getppid(), SIGUSR1); /* signal parent */
}
```

```
0 1 2 3 4
```



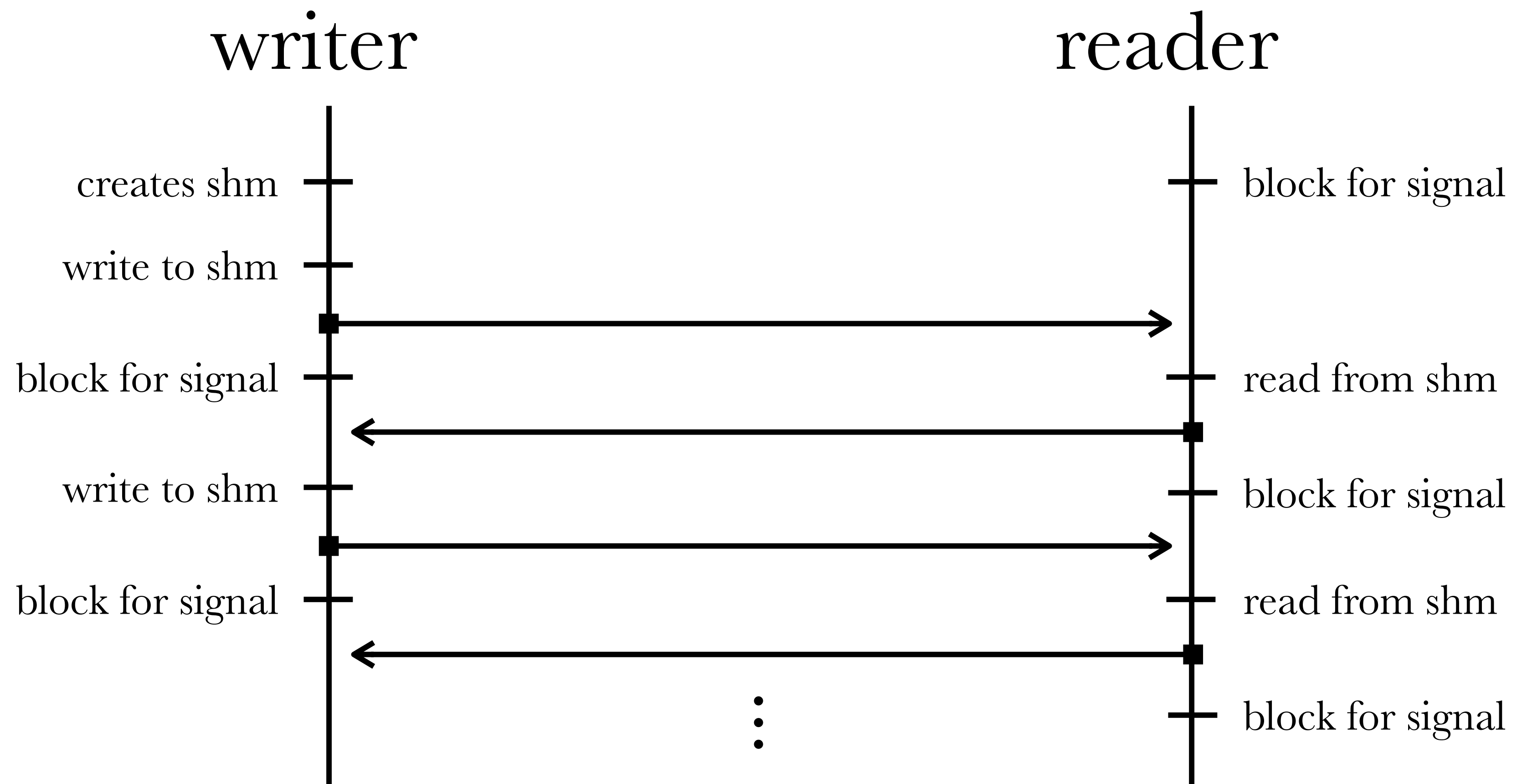
IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
Computer Science

but wait ...



we've eliminated concurrency!
(w.r.t. shm access)

how about:



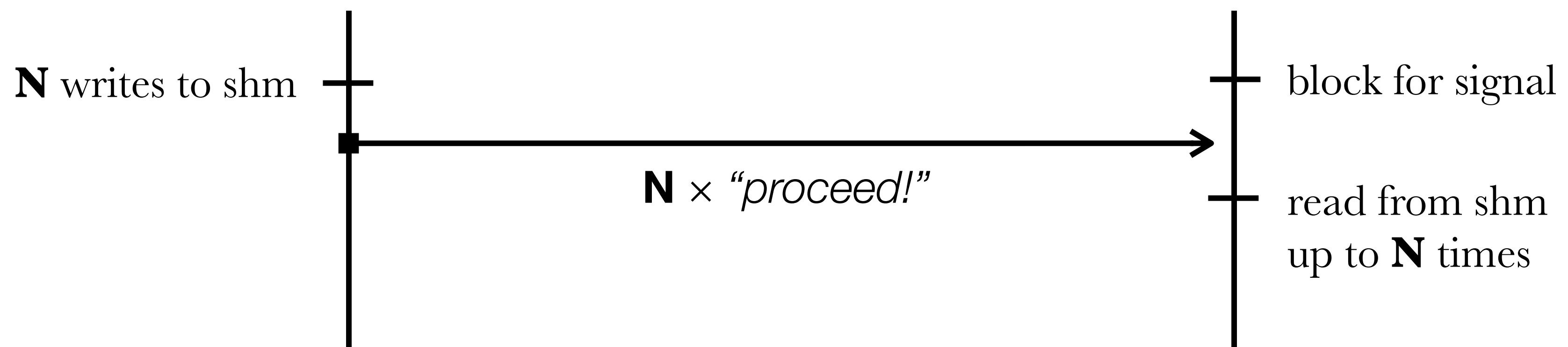
```
/* parent/writer process */
if ((pid = fork()) != 0) {
    ...

    for (i=0; i<SHM_SIZE; i++) {
        shm_arr[i] = i;
    }

    ...
}
```

```
/* child/reader process */
else {
    ...

    for (i=0; i<SHM_SIZE; i++) {
        printf("%d ", shm_arr[i]);
    }

    ...
}
```

still not really good enough …

writer should be permitted to write up to `SHM_SIZE` before stopping

how about:



recall: signals aren't queued! :-(

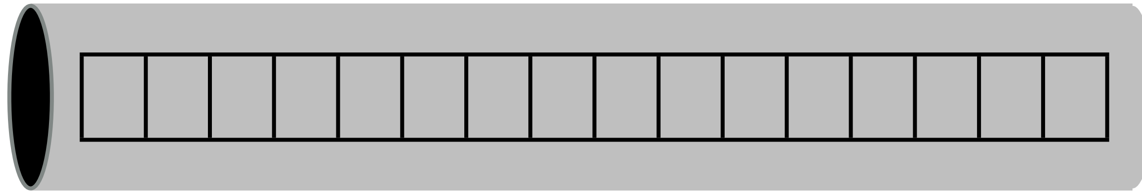
also, with all this sync overhead,
shm isn't looking so hot anymore

§Unnamed Pipes

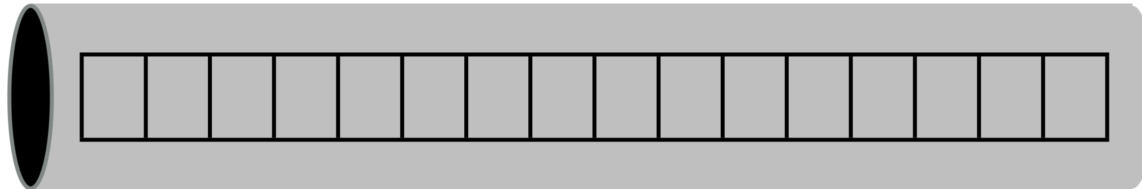
```c
int pipe(int fds[2]);
```

`fds[0]` is the “reading end”

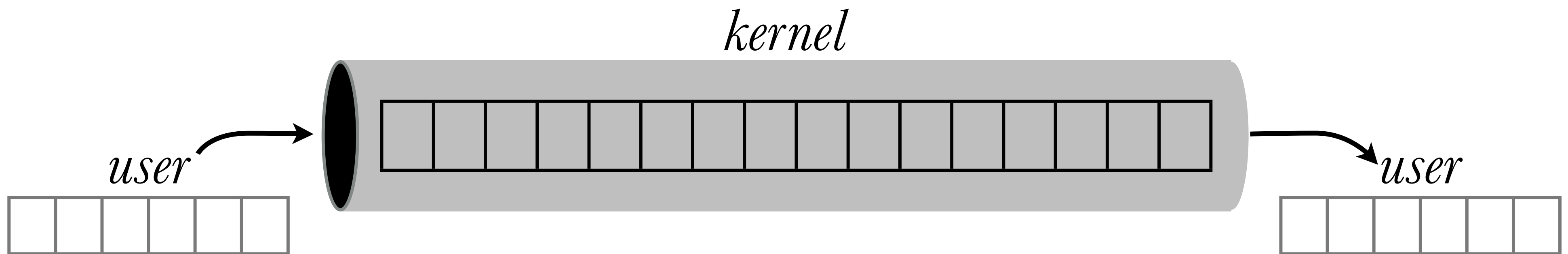
`fds[1]` is the “writing end”



- buffer of finite size = PIPE_BUF
  - defined in <limits.h>
- on fourier = 4096



- `read` blocks for min of 1 byte
- `write` blocks until complete
- writes ≤ `PIPE_BUF` are **atomic**
  - can't be interrupted by other writes



- speed can’t compare to shm!
- requires copy from user to kernel buffer, then back to a user buffer

```c
int i, j, fds[2];

pipe(fds); /* create pipe */

if (fork() != 0) {
    /* parent writes */
    for (i=0; i<10; i++) {
        write(fds[1], &i, sizeof(int));
    }
} else {
    /* child reads */
    for (i=0; i<10; i++) {
        read(fds[0], &j, sizeof(int));
        printf("%d ", j);
    }
}
```

```
0 1 2 3 4 5 6 7 8 9
```

```c
int i, n, fds[2];
char buf[80];
char *strings[] = {"the", "quick", "brown", "fox", "jumps",
                   "over", "the", "lazy", "dog"};

pipe(fds);
for (i=0; i<9; i++) { /* 9 child processes! */
    if (fork() == 0) {
        write(fds[1], strings[i], strlen(strings[i]));
        exit(0);
    }
}

while ((n = read(fds[0], buf, sizeof(buf))) > 0) {
    write(1, buf, n);
    printf("\n");
}
```

```
the
quick
foxoverbrown
jumpslazythe
dog
```

kernel takes care of buffering
& synchronization! (yippee!)

back to shell pipes:

```
$ echo hello | wc
       1       1       6
```

```
int fds[2];
pid_t pid1, pid2;
pipe(fds);
if ((pid1 = fork()) == 0) {
    dup2(fds[1], 1);
    execlp("echo", "echo", "hello", NULL);
}
if ((pid2 = fork()) == 0) {
    dup2(fds[0], 0);
    execlp("wc", "wc", NULL);
}
waitpid(pid2, NULL, 0);
```

```
(hangs)
```

key: read on pipe will *always block* for ≥ 1 byte until writing ends are all closed

```
int fds[2];
pid_t pid1, pid2;
pipe(fds);
if ((pid1 = fork()) == 0) {
    dup2(fds[1], 1);
    execlp("echo", ...);
}
if ((pid2 = fork()) == 0) {
    dup2(fds[0], 0);
    execlp("wc", ...);   ← never sees EOF!
}
waitpid(pid2, NULL, 0);
```

```
if ((pid1 = fork()) == 0) {
    dup2(fds[1], 1);
    close(fds[1]);
    execlp("echo", "echo", "hello", NULL);
}
close(fds[1]);
if ((pid2 = fork()) == 0) {
    dup2(fds[0], 0);
    execlp("wc", "wc", NULL);
}
```

```
      1       1       6
```

so … why “**unnamed**” pipes?

```
int fds[2];

if (fork() == 0) {
    /* proc 1 */
    pipe(fds);
    write(fds[1], …);
}

if (fork() == 0) {
    /* proc 2 */
    read(?, …);
}
```

- no way for proc 1 and proc 2 to talk over pipe!
- identified solely by FDs – process local

§Named Pipes (FIFOs)

```
int mkfifo (const char* path,
                  mode_t perms)
```

- creates a *FIFO special file* at `path` in file system
- `open`(s) then `read` & `write`
- exhibits pipe semantics!

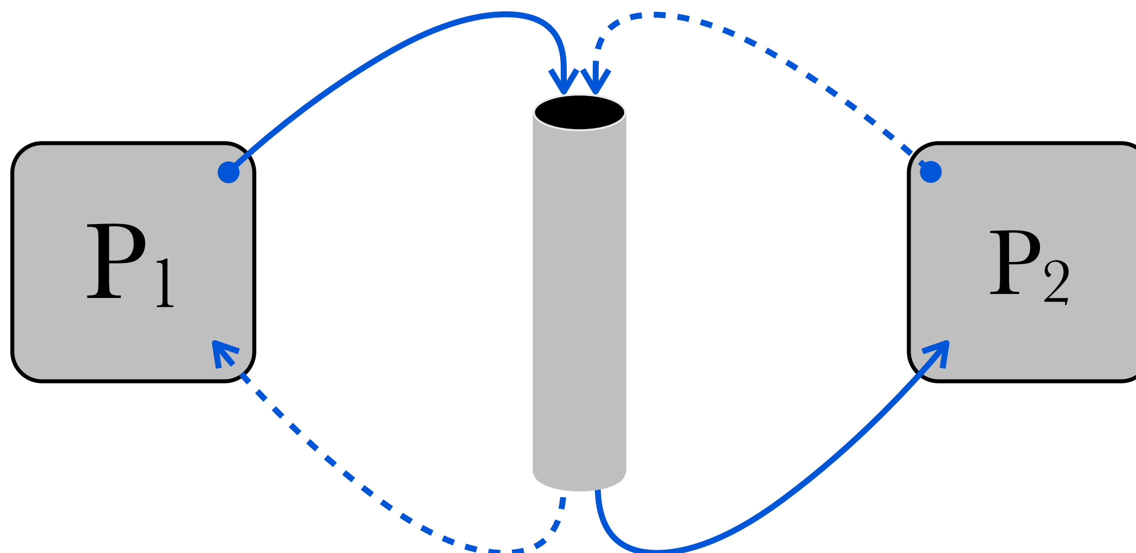
Science

Computer Science

```
$ mkfifo /tmp/my.fifo
$ ls -l /tmp/my.fifo
 prw-------  1 lee  wheel  0 24 Apr 04:13 /tmp/my.fifo

$ cat /tmp/my.fifo &
 [1]+ cat /tmp/my.fifo &

$ echo hello > /tmp/my.fifo
 hello
 [1]+  Done    cat /tmp/my.fifo

$ cat /etc/passwd > /tmp/my.fifo &
 [1]+ cat /etc/passwd >/tmp/my.fifo &

$ cat /tmp/my.fifo
 root:*:0:0:System Administrator:/var/root:/bin/tcsh
 lee:*:501:20:Michael Lee,,,:/Users/lee:/bin/bash
 ...
 [1]+  Done    cat /etc/passwd >/tmp/my.fifo

$ rm /tmp/my.fifo
```

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

important limitation:
while bi-directional communication is possible; it is ***half-duplex***



let's talk a bit more about
synchronization

why?

so **concurrent** systems can be made **predictable**

how?

so far:

- `wait` (limited)
- `kill` & `signal` (lousy)
- `pipe` (implicit)

some UNIX IPC mechanisms are *purpose-built* for synchronization

§File Locks

motivation:

- process virtual worlds don't extend to the file system
- concurrently modifying files can have ugly consequences
- but files are the most widely used form of IPC!

a process can acquire a **lock** on a file, preventing other processes from using it

```
int fcntl(int fd, int cmd, struct flock);
```

```
cmd = { F_GETLK, F_SETLK, F_SETLKW}
```

test, *set,* *set (wait)*

```
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;      /* Num bytes to lock (0 for all) */
    pid_t l_pid;      /* PID of process holding lock   */
    ...
};
```

important: locks are *not* preserved across forks! (i.e., a child doesn't inherit its parent's locks)

```c
int fd, n;
struct flock fl;
char buf[80];

fd = open("shared.txt", O_RDWR|O_CREAT, 0600);

fl.l_type = F_WRLCK;
fl.l_whence = 0;
fl.l_len = 0;
fl.l_pid = getpid();
fcntl(fd, F_SETLK, &fl); /* set write (exclusive) lock */

if (fork() == 0) {
    fl.l_type = F_RDLCK;
    fcntl(fd, F_SETLKW, &fl); /* wait for & set read lock */
    lseek(fd, 0, SEEK_SET);   /* rewind pos */
    n = read(fd, buf, sizeof(buf));
    write(1, buf, n);
} else {
    sleep(1);                 /* make child wait */
    write(fd, "hello there!", 13);
    fl.l_type = F_UNLCK;
    fcntl(fd, F_SETLK, &fl);  /* release lock */
}
```

```
hello there!
```

problem: most file systems only support **advisory** locking

i.e., locks are not enforced!

in Linux, mandatory locking is *possible*, but requires filesystem to support it

The implementation of mandatory locking in all known versions of Linux is **subject to race conditions** which render it **unreliable**: a write(2) call that overlaps with a lock may modify data after the mandatory lock is acquired; a read(2) call that overlaps with a lock may detect changes to data that were made only after a write lock was acquired. Similar races exist between mandatory locks and mmap(2). It is therefore **inadvisable to rely on mandatory locking**.

also, file locks are not designed for *general-purpose synchronization*

e.g., what if we want to:

- allow only 1 of N processes to access an *arbitrary* resource?
- allow M of N processes to access a resource?
- control the order in which processes run?

§Semaphores

semaphore = synchronization primitive

- object with associated counter
- usually init to count ≥ 0

```
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

- creates semaphore initialized to `value`

```
sem_t *sem_open(const char *name, int oflag);
```

- retrieves existing semaphore

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

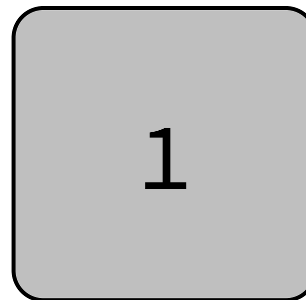
- **decrements** value; **blocks** if new value < 0
- returns 0 on success
- returns -1 if interrupted without decrementing

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

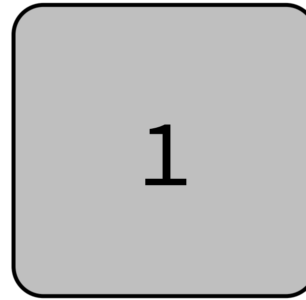
- **increments** value; **unblocks 1** process (if any)
- returns 0 on success

```
sem_t *sem = sem_open("/fred", O_CREAT, 0600, 1);
```

“/fred”



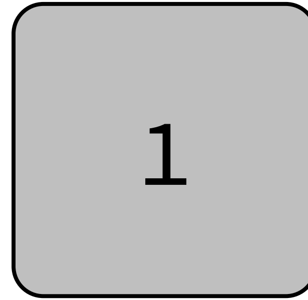
“/fred”



P_1

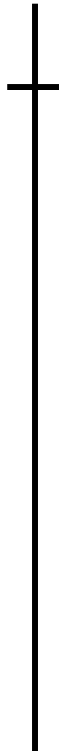
P_2

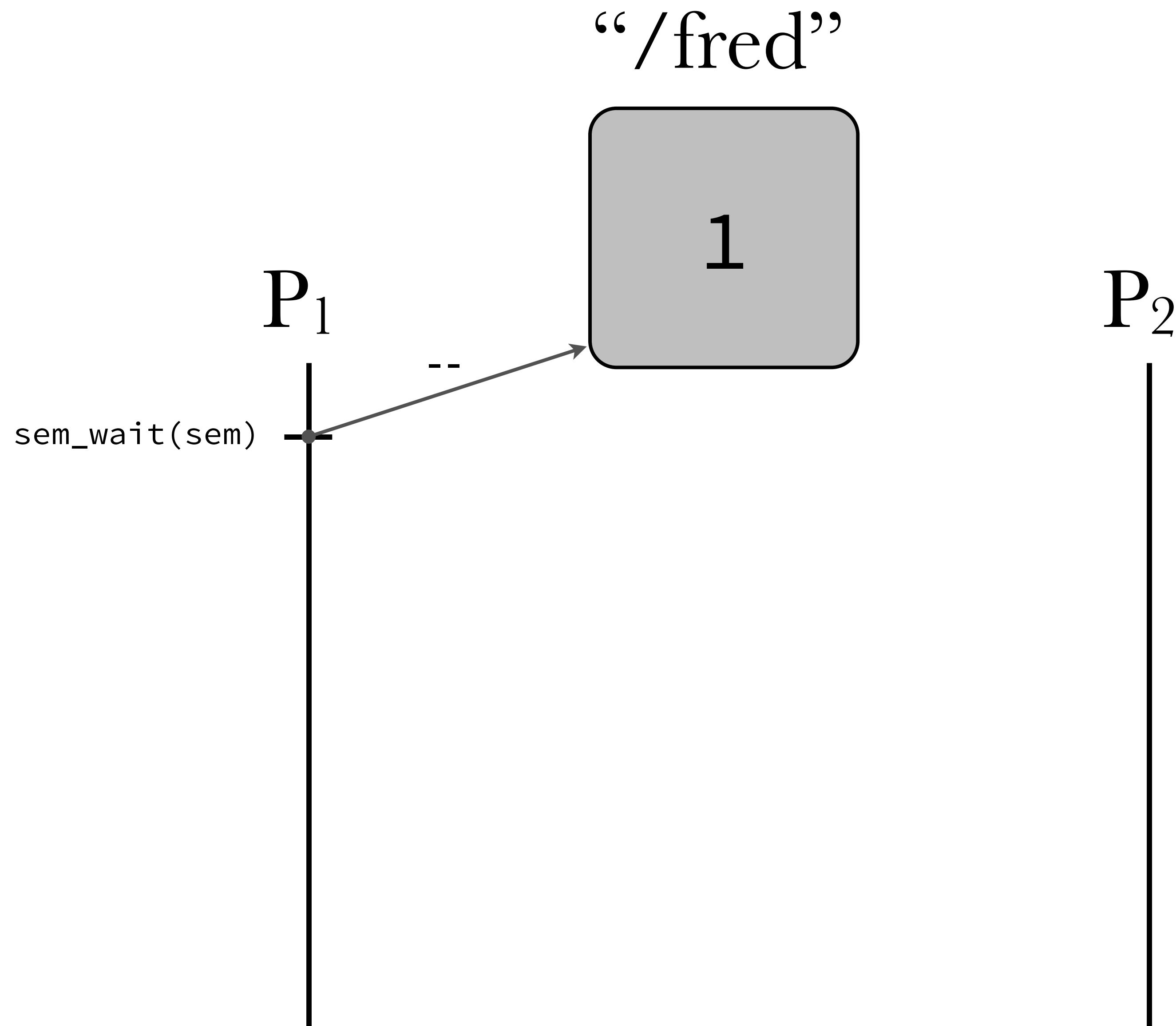
“/fred”

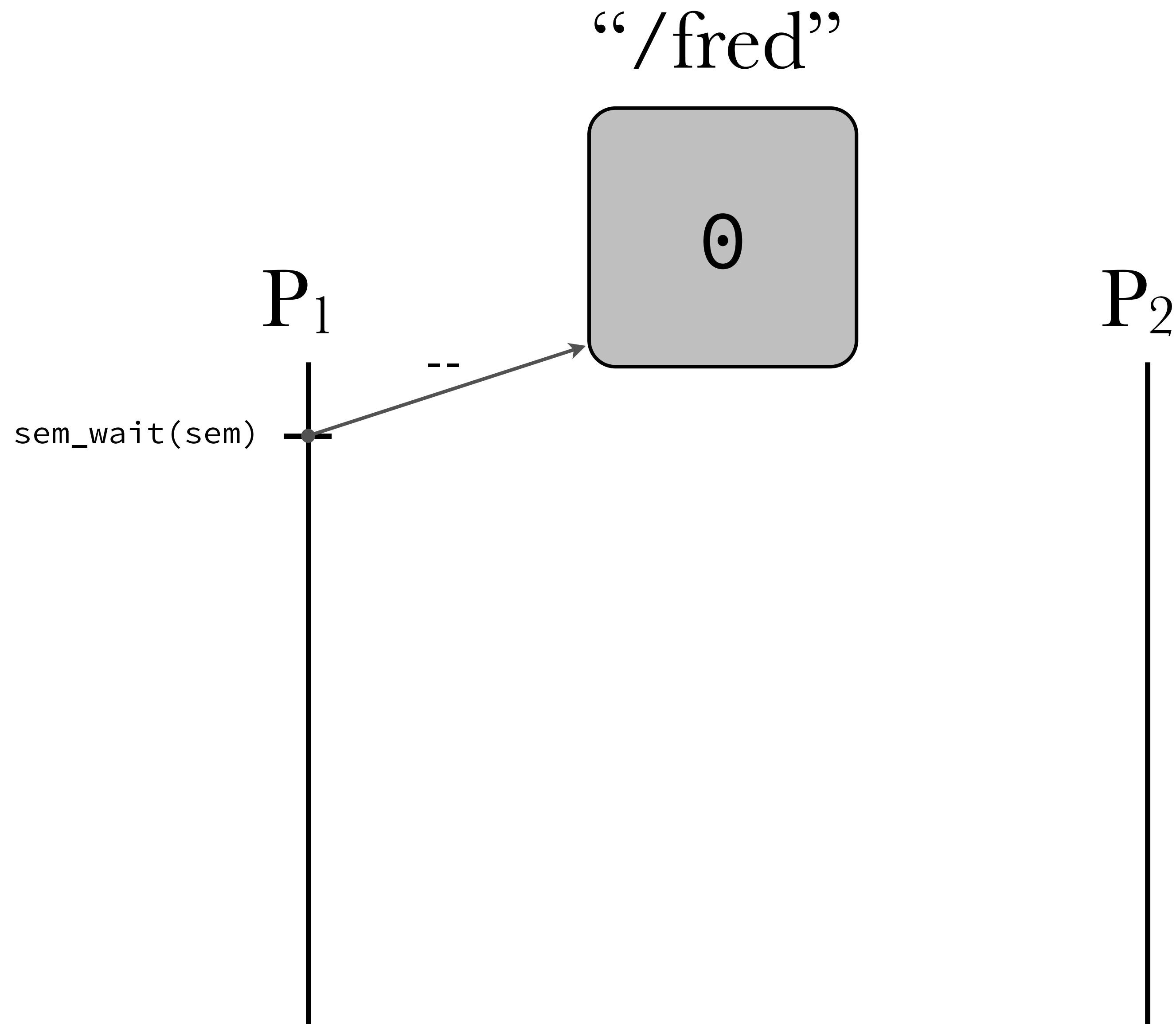


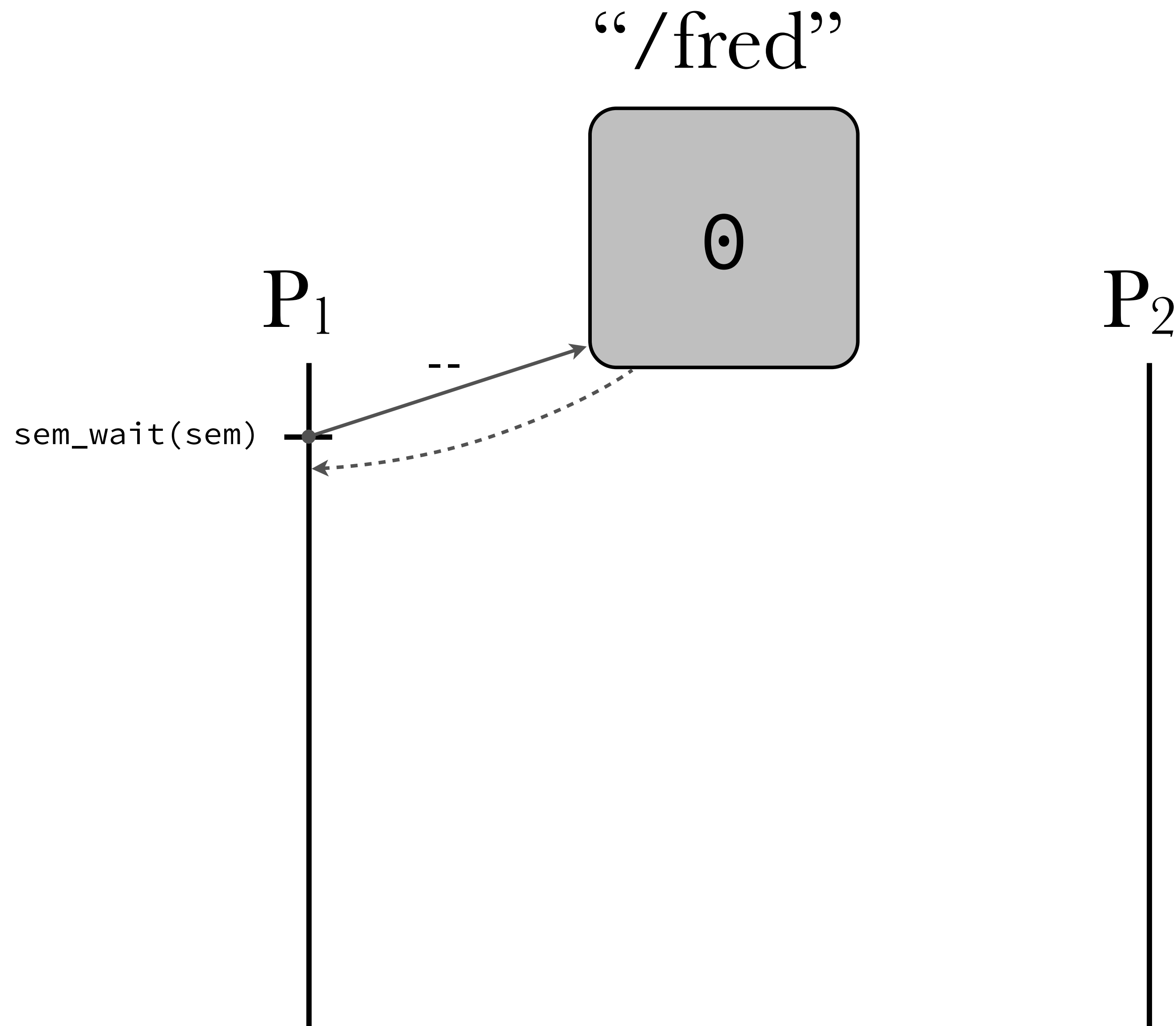
P_1

P_2

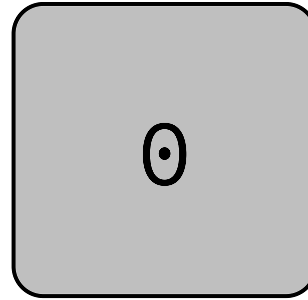



"/fred"
1
P1
P2
--
sem_wait(sem)


"/fred"
0
P1
P2
--
sem_wait(sem)


"/fred"
0
P_1
P_2
--
sem_wait(sem)

“/fred”

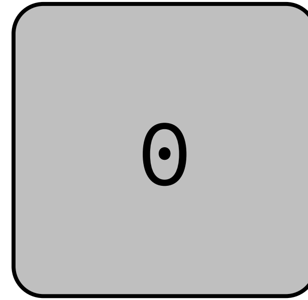


P_1

P_2

`sem_wait(sem)`

⋮

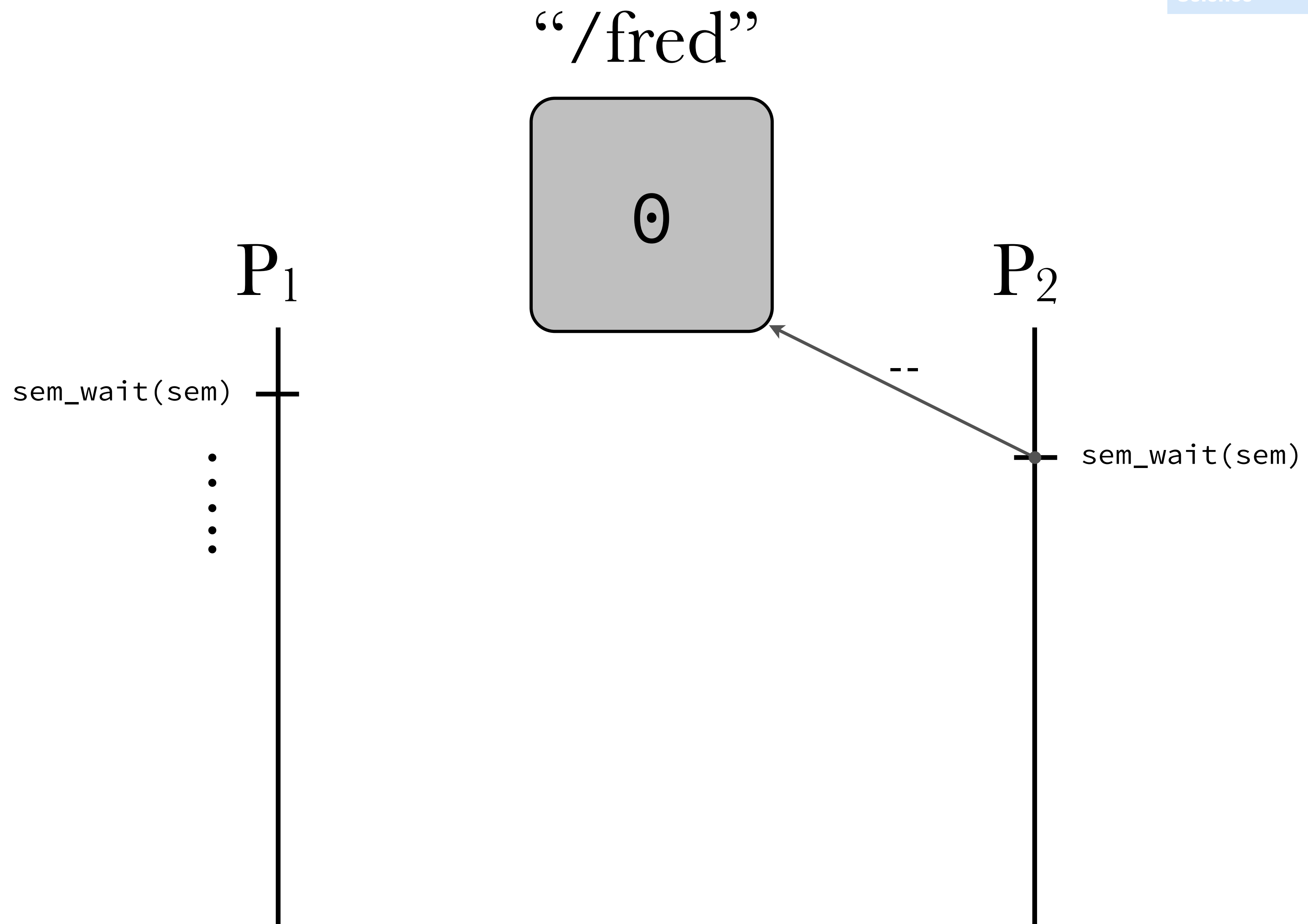


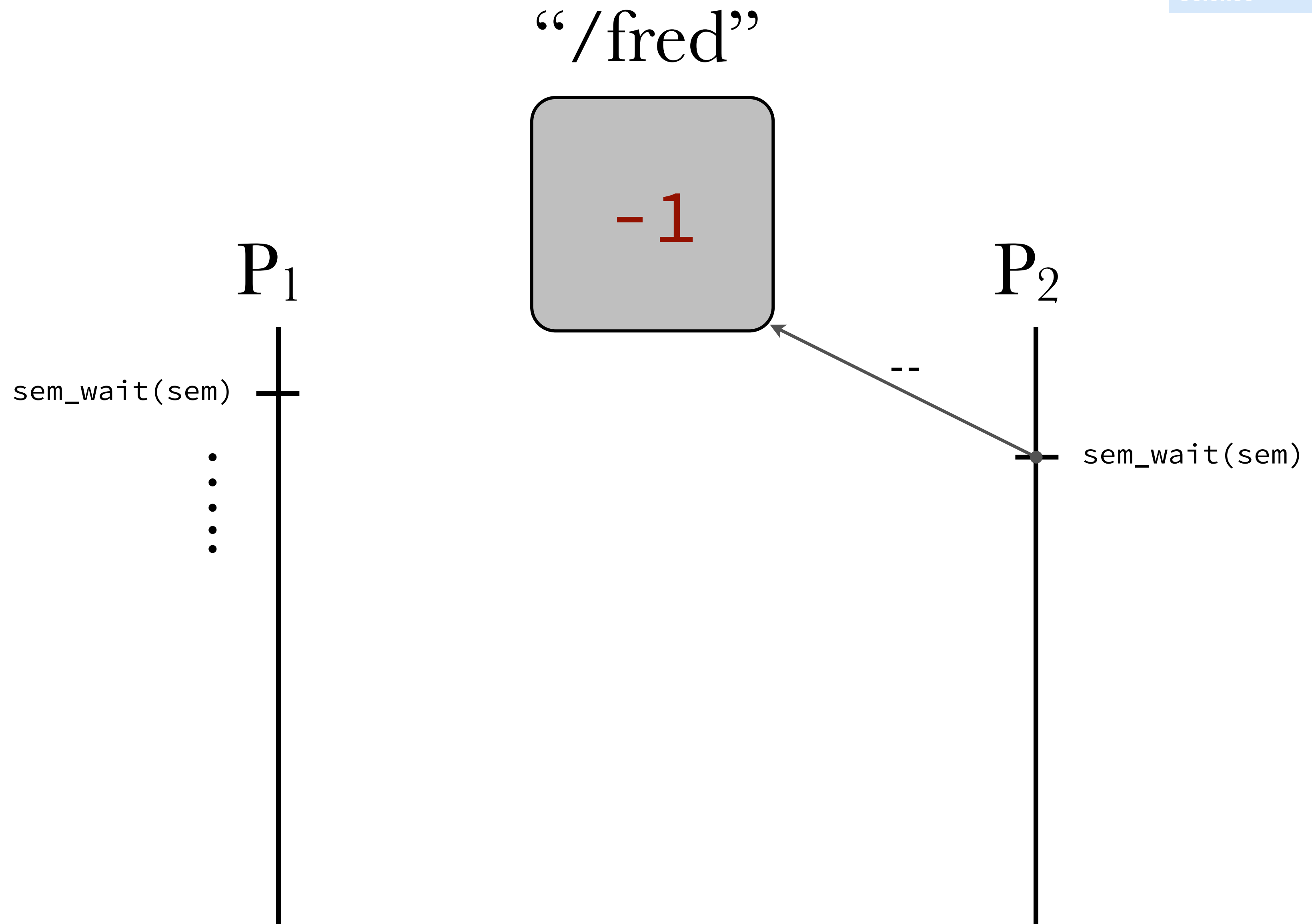
P_1

P_2

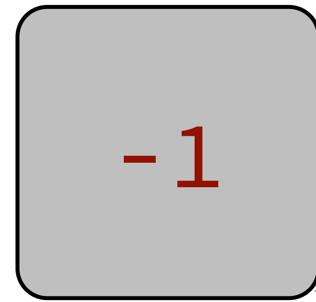
```
sem_wait(sem)
```

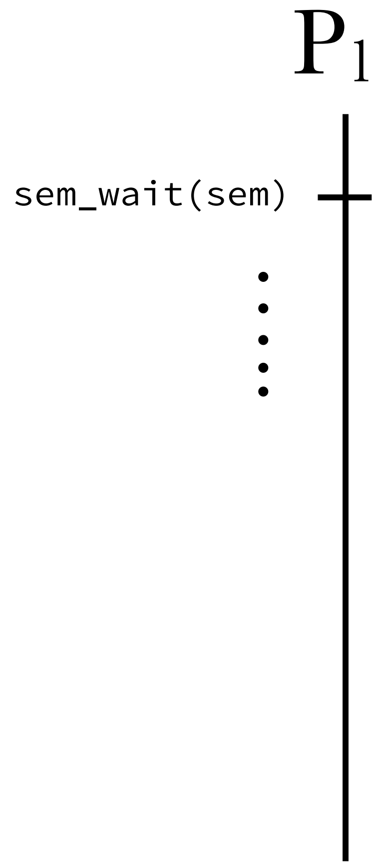
```
sem_wait(sem)
```

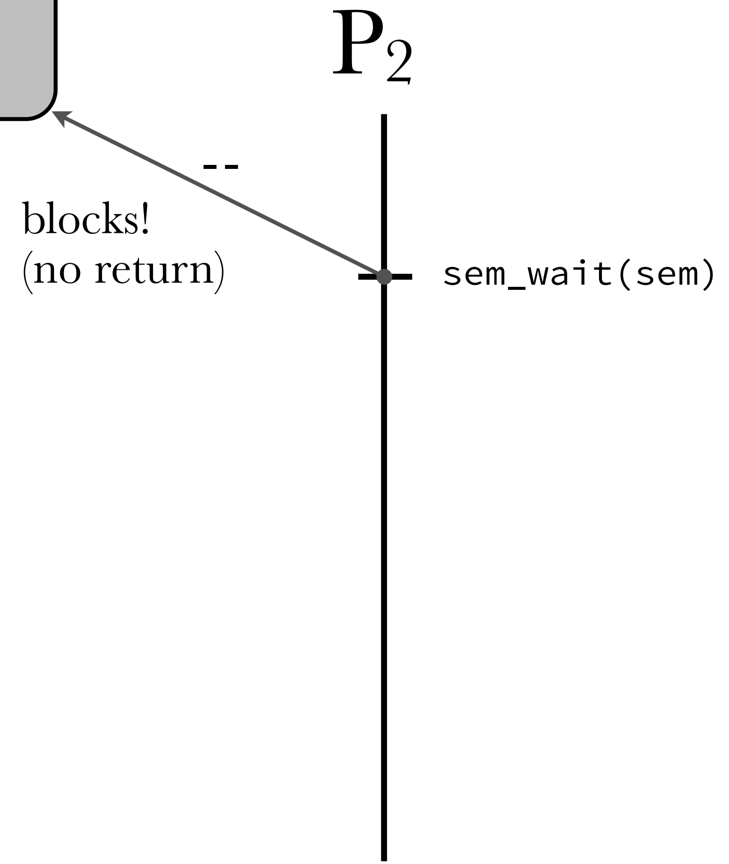

“/fred”
0
P_1
P_2
sem_wait(sem)
--
sem_wait(sem)

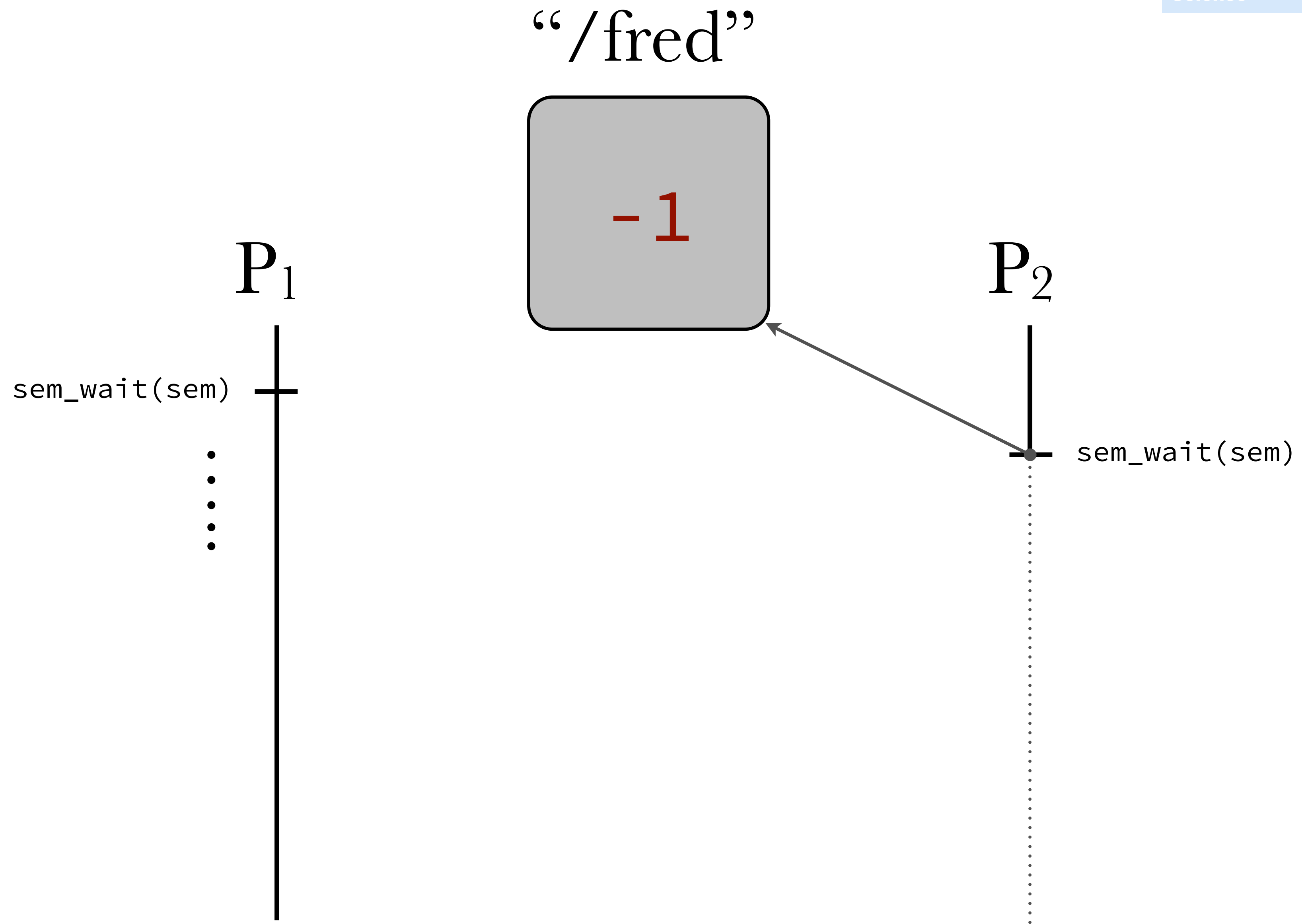


Science
Computer Science
IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

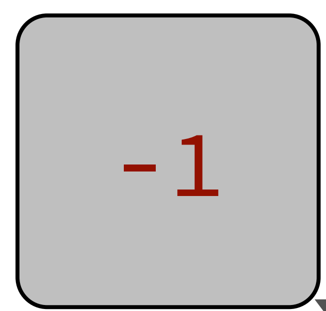

"/fred"
-1


P1
sem_wait(sem)


P2
--
blocks!
(no return)
sem_wait(sem)


"/fred"
-1
P_1
P_2
sem_wait(sem)
sem_wait(sem)

“/fred”

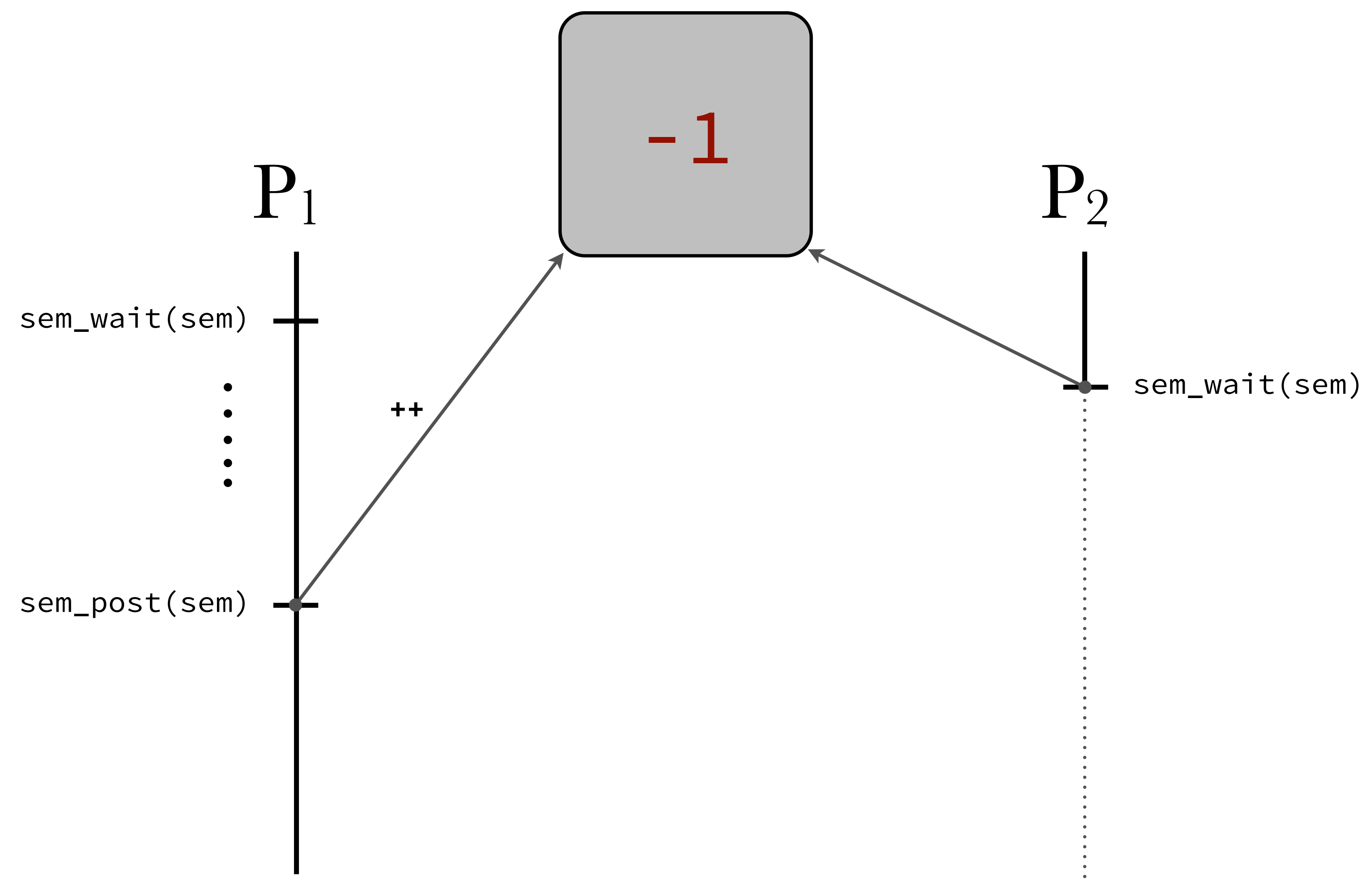


P_1

P_2

```
sem_wait(sem)
sem_post(sem)
```

```
sem_wait(sem)
```


"/fred"
-1
P_1
P_2
sem_wait(sem)
++
sem_post(sem)
sem_wait(sem)

“/fred”

0

P1

P2

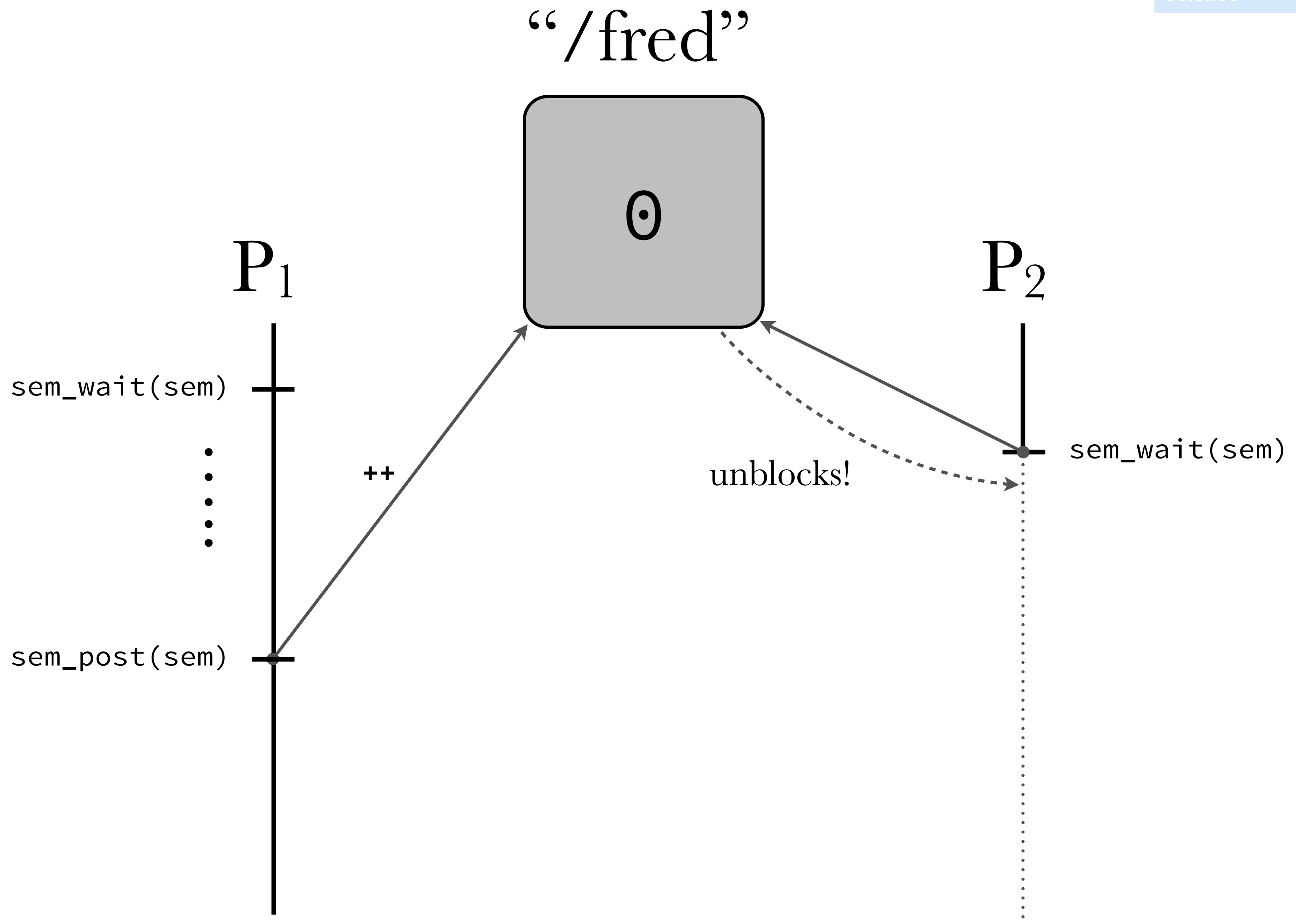
sem_wait(sem)

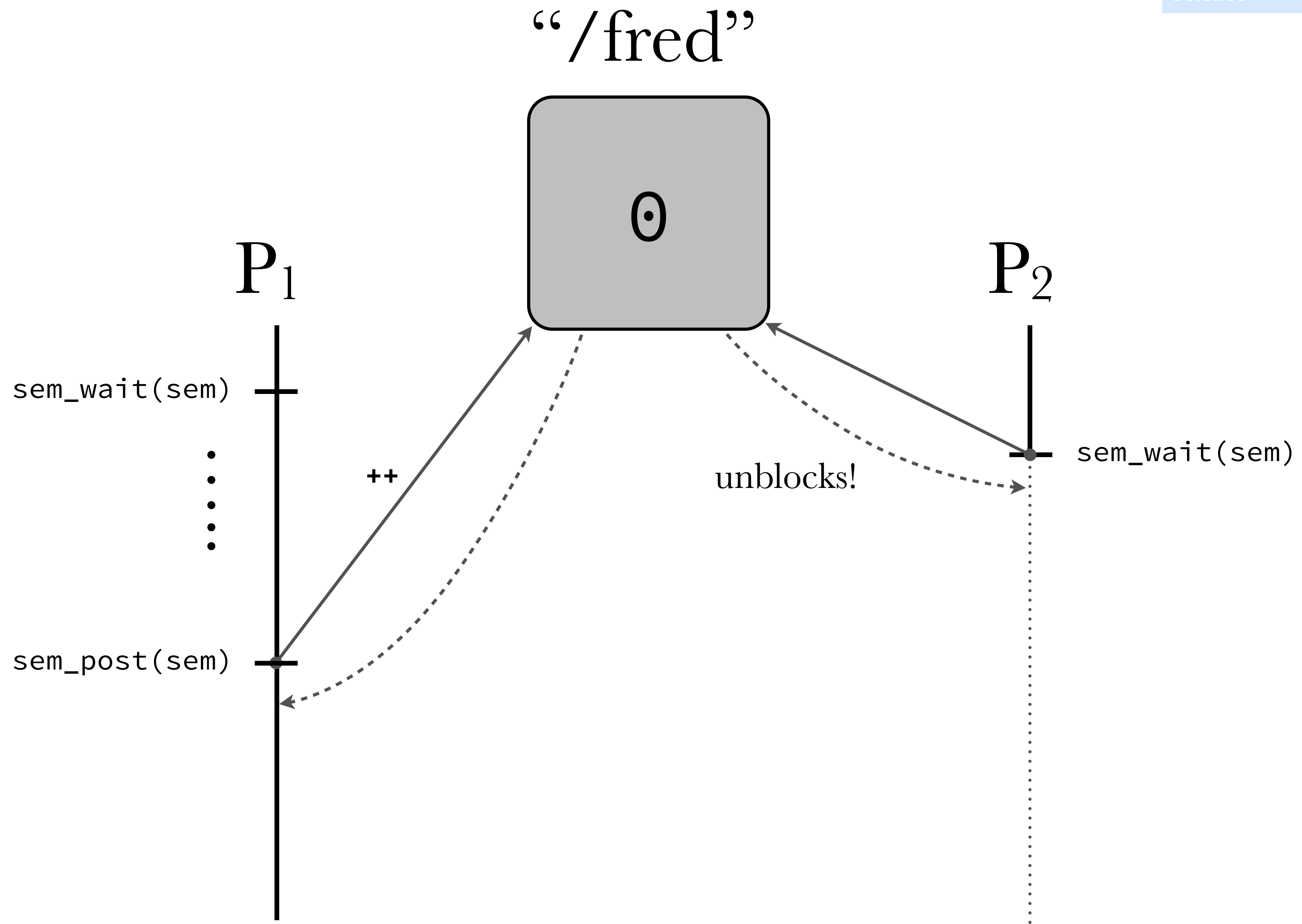
⋮

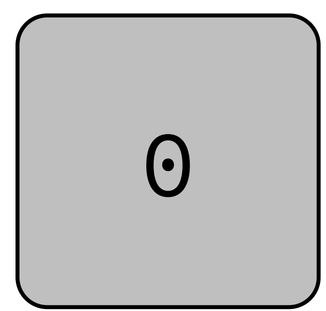
++

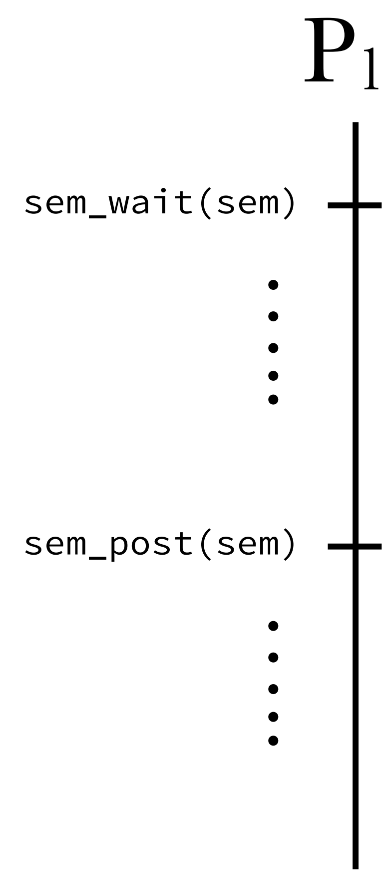
sem_post(sem)

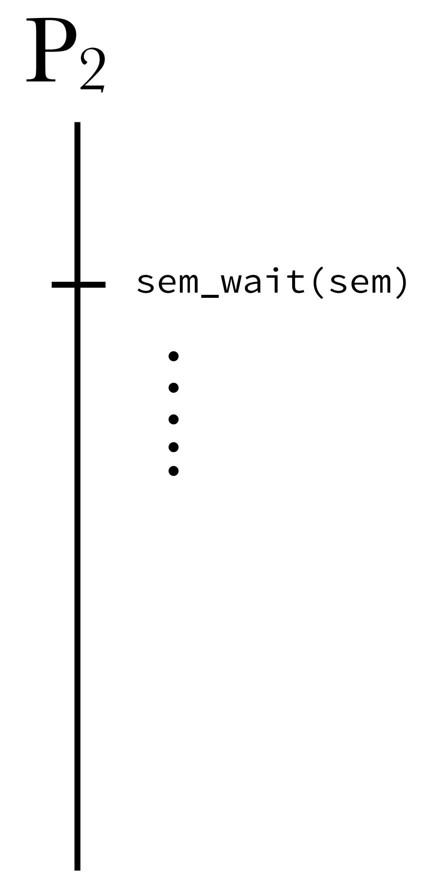
sem_wait(sem)

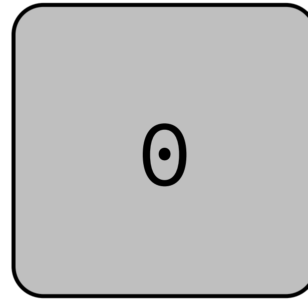

"/fred"
0
P1
P2
sem_wait(sem)
++
unblocks!
sem_wait(sem)
sem_post(sem)


“/fred”
0
P1
P2
sem_wait(sem)
sem_post(sem)
++
unblocks!
sem_wait(sem)


“/fred”
0


P1
sem_wait(sem)
sem_post(sem)


P2
sem_wait(sem)



P_1

`sem_wait(sem)`

⋮

`sem_post(sem)`

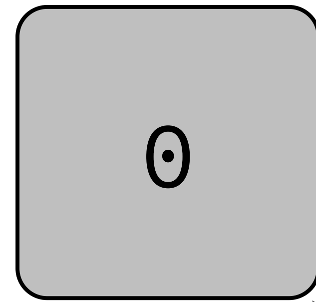
⋮

P_2

`sem_wait(sem)`

⋮

`sem_post(sem)`



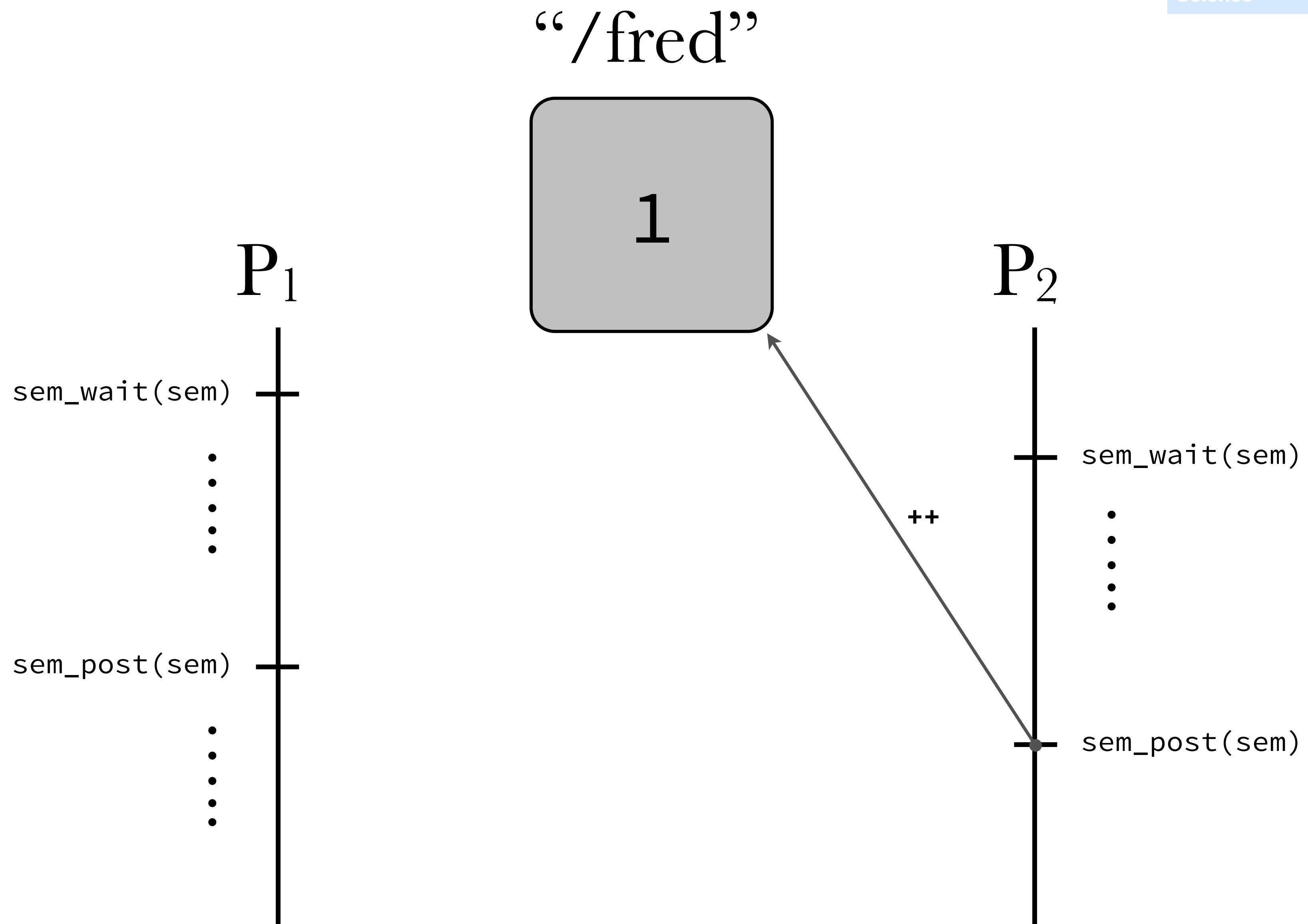
P_1

```
sem_wait(sem)
    ⋮
sem_post(sem)
    ⋮
```

P_2

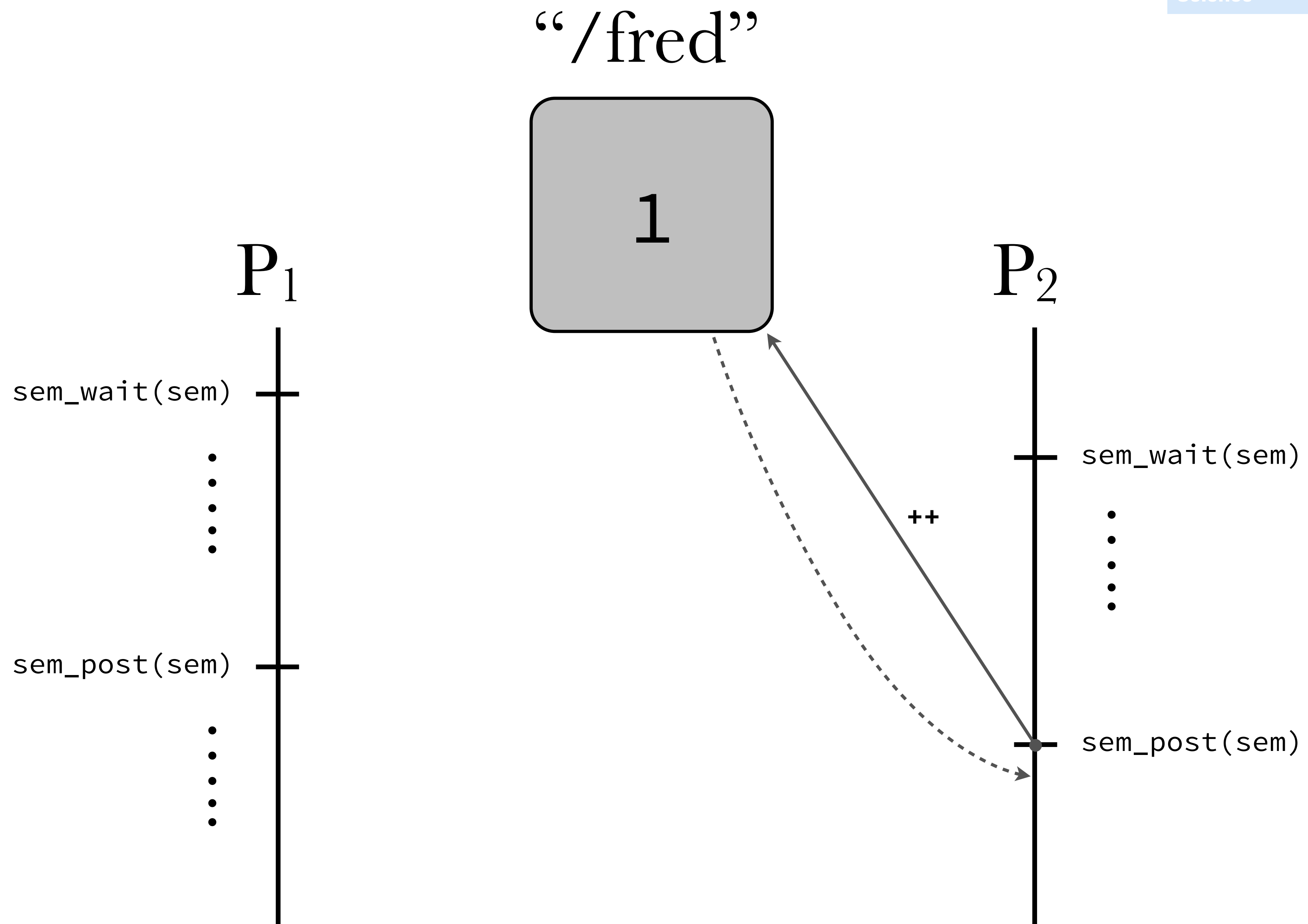
```
sem_wait(sem)
    ⋮
sem_post(sem)
```

++

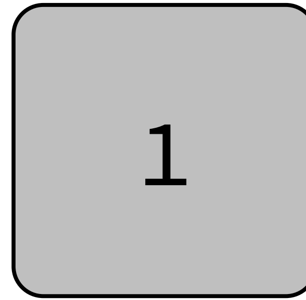


Science
Computer Science

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY


"/fred"
1
P1
P2
sem_wait(sem)
sem_post(sem)
++
sem_wait(sem)
sem_post(sem)

“/fred”



P_1

```
sem_wait(sem)
  ⋮
sem_post(sem)
  ⋮
```

P_2

```
sem_wait(sem)
  ⋮
sem_post(sem)
  ⋮
```

```c
/* unsynchronized file writers */
int i, j, fd;
fd = open("shared.txt", O_CREAT|O_WRONLY, 0600);
for (i=0; i<5; i++) {
    if (fork() == 0) {
        for (j='0'; j<='9'; j++) {
            write(fd, &j, 1);
            sleep(random() % 3);
        }
        exit(0);
    }
}
```

```
$ cat shared.txt
01000011223411234532356765475968764798789529869789
```

```
/* synchronized file writers */
int i, j, fd;
sem_t *mutex = sem_open("/mutex", O_CREAT, 0600, 1);
fd = open("shared.txt", O_CREAT|O_WRONLY, 0600);
for (i=0; i<5; i++) {
    if (fork() == 0) {
        while (sem_wait(mutex) < 0) ;
        for (j='0'; j<='9'; j++) {
            write(fd, &j, 1);
            sleep(random() % 3);
        }
        sem_post(mutex);
        exit(0);
    }
}
```

```
$ cat shared.txt
01234567890123456789012345678901234567890123456789
```

```
/* unsynchronized file writers */
for (i=0; i<5; i++) {
    if (fork() == 0) {
        /* write */
        exit(0);
    }
}
```

```
/* synchronized file writers */
sem_t *mutex = sem_open("/mutex", O_CREAT, 0600, 1);
for (i=0; i<5; i++) {
    if (fork() == 0) {
        while (sem_wait(mutex) < 0) ;
        /* write */
        sem_post(mutex);
        exit(0);
    }
}
```

```
...
for (j='0'; j<='9'; j++) {
    write(fd, &j, 1);
    sleep(random() % 3); /* avg sleep = 1 sec */

}
...
```

```
$ time ./fw-async
./fw-async  0.00s user 0.01s system 0% cpu 11.019 total

$ time ./fw-sync
./fw-sync   0.00s user 0.01s system 0% cpu 50.050 total
```

```c
int i;

if (fork() == 0) {
    /* P1 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        printf("P1: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
if (fork() == 0) {
    /* P2 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        printf("P2: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
```

```
P1: Iter 0
P2: Iter 0
P2: Iter 1
P1: Iter 1
P1: Iter 2
P2: Iter 2
P1: Iter 3
P1: Iter 4
P2: Iter 3
P2: Iter 4
P1: Iter 5
P2: Iter 5
P1: Iter 6
P1: Iter 7
P1: Iter 8
P1: Iter 9
P2: Iter 6
P2: Iter 7
P2: Iter 8
P2: Iter 9
```

```
int i;
sem_t *p1_arrived, *p2_arrived;
p1_arrived = sem_open("/sem1", O_CREAT, 0600, 0);
p2_arrived = sem_open("/sem2", O_CREAT, 0600, 0);

if (fork() == 0) {
    /* P1 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        sem_post(p1_arrived);
        sem_wait(p2_arrived);
        printf("P1: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
if (fork() == 0) {
    /* P2 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        sem_post(p2_arrived);
        sem_wait(p1_arrived);
        printf("P2: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
```

```
P1: Iter 0
P2: Iter 0
P1: Iter 1
P2: Iter 1
P2: Iter 2
P1: Iter 2
P1: Iter 3
P2: Iter 3
P2: Iter 4
P1: Iter 4
P1: Iter 5
P2: Iter 5
P1: Iter 6
P2: Iter 6
P1: Iter 7
P2: Iter 7
P1: Iter 8
P2: Iter 8
P1: Iter 9
P2: Iter 9
```

```
int i;
sem_t *p1_arrived, *p2_arrived;
p1_arrived = sem_open("/sem1", O_CREAT, 0600, 0);
p2_arrived = sem_open("/sem2", O_CREAT, 0600, 0);

if (fork() == 0) {
    /* P1 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        sem_wait(p2_arrived);
        sem_post(p1_arrived);
        printf("P1: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
if (fork() == 0) {
    /* P2 */
    for (i=0; i<10; i++) {
        sleep(random() % 3);
        sem_wait(p1_arrived);
        sem_post(p2_arrived);
        printf("P2: Iter %d\n", i);
        fflush(stdout);
    }
    exit(0);
}
```

just as with shared memory, semaphores *persist* when process exits … must *unlink*

```
sem_t *mutex = sem_open("/mutex", O_CREAT, 0600, 1);
for (i=0; i<5; i++) {
    if (fork() == 0) {
        while (sem_wait(mutex) < 0) ;
        ...
        sem_post(mutex);
        exit(0);
    }
}
while (wait(NULL) >= 0);
sem_close(mutex);
sem_unlink("/mutex");
```

there is much, much more to synchronization & concurrency …

(coming in CS 450!)

§IPC Recap

Select IPC mechanisms:

1. signals
2. (regular) files
3. shared memory
4. unnamed & named pipes
5. file locks & semaphores
6. sockets

one motive: *data communication*

- at one end: shm — fast but no synchronization
- at other end: pipes — slower but implicitly synchronized

another motive: *synchronization*

- signals: system events
- file locks (advisory!)
- semaphores: simple but surprisingly versatile!

so far, just **intra**-system IPC.

coming later, network sockets for **inter**-system IPC!