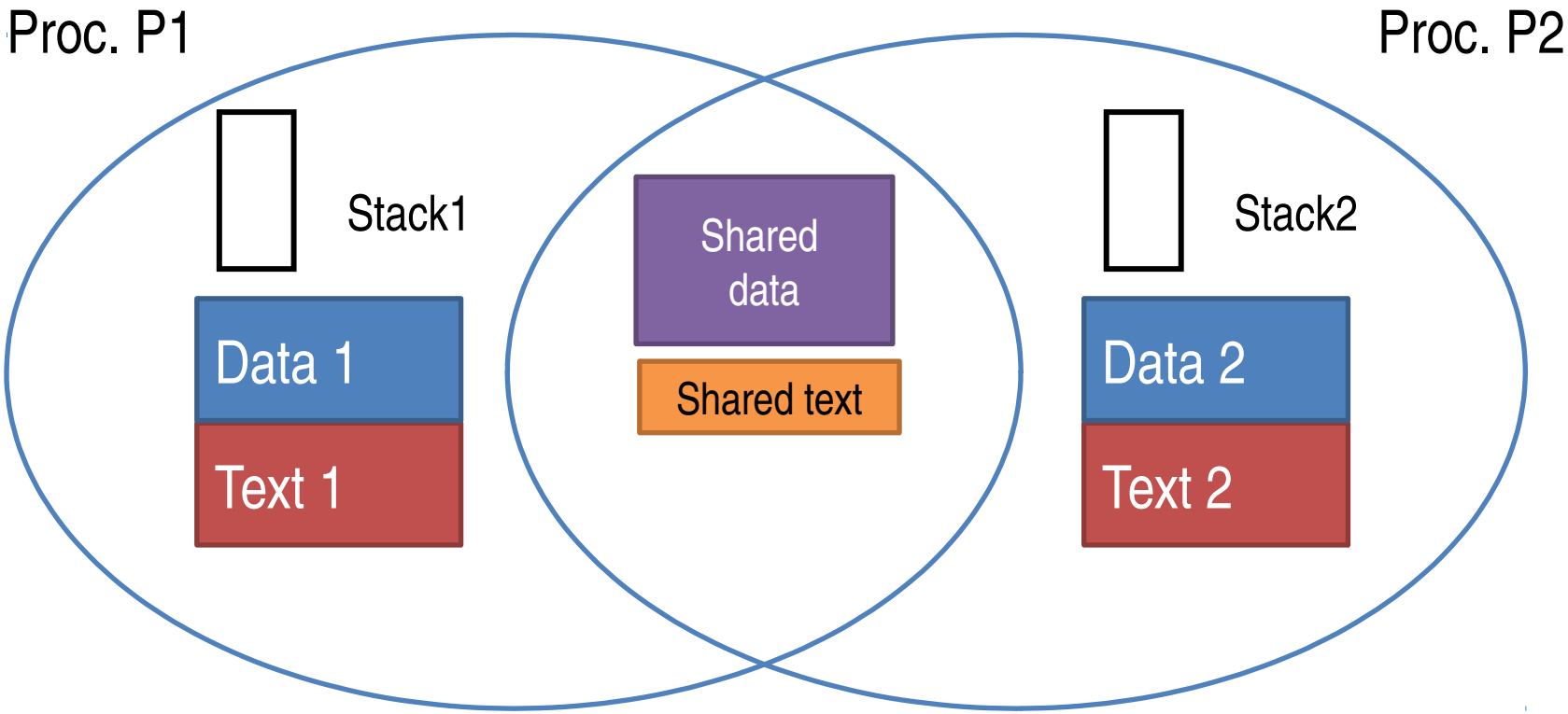
Cooperation models

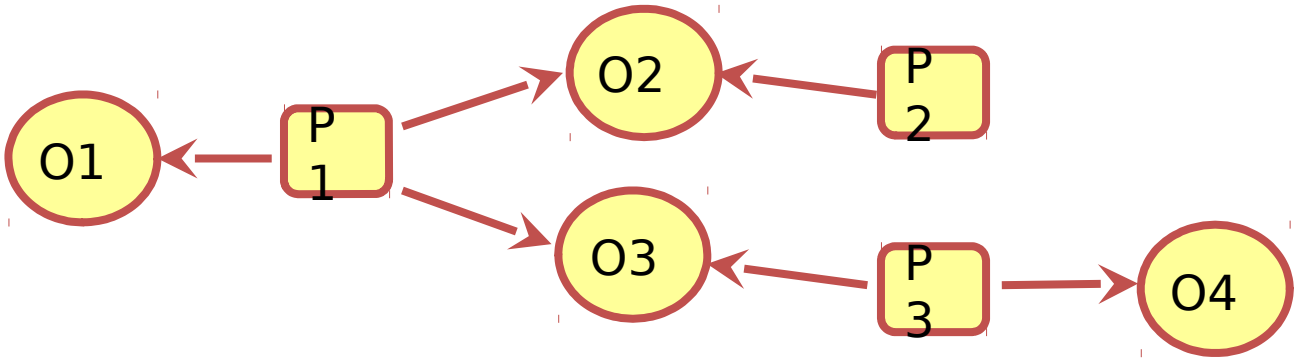
Global vs. local model

- Global environment
  - Processes can share data
  - Shared memory
- Local environment
  - processes/ do not share data
  - No shared memory

Global environment



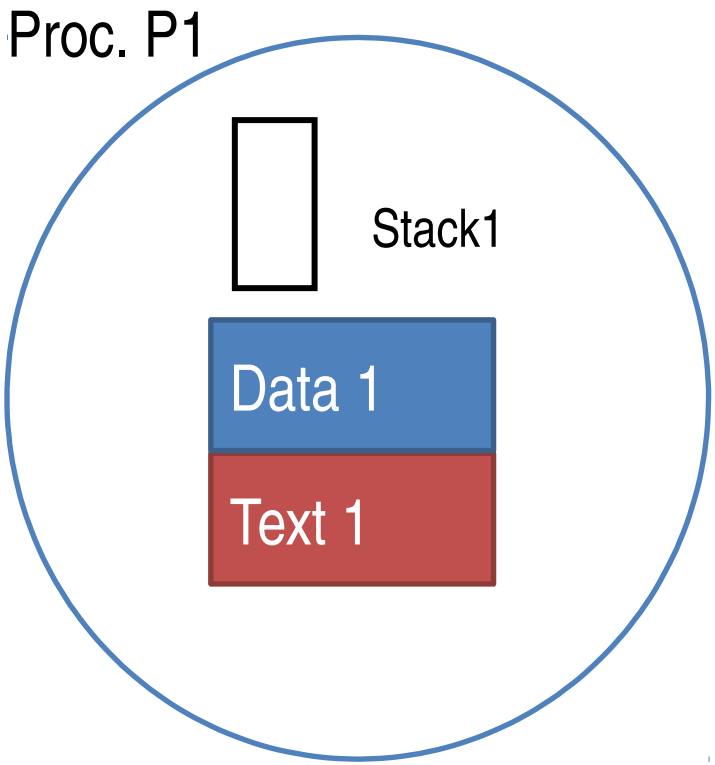
Global environment

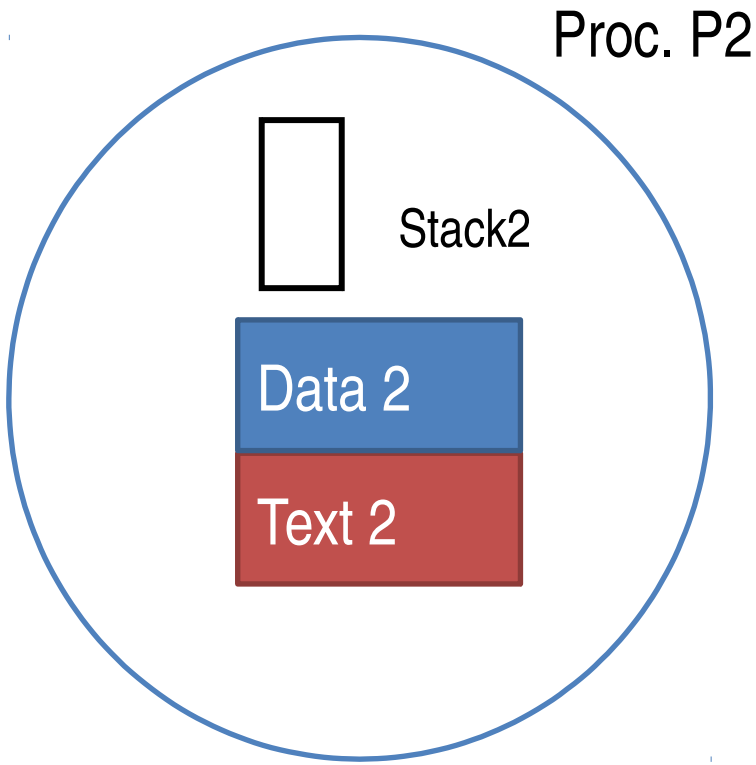


O1, O4 private objects
O2, O3 shared objects

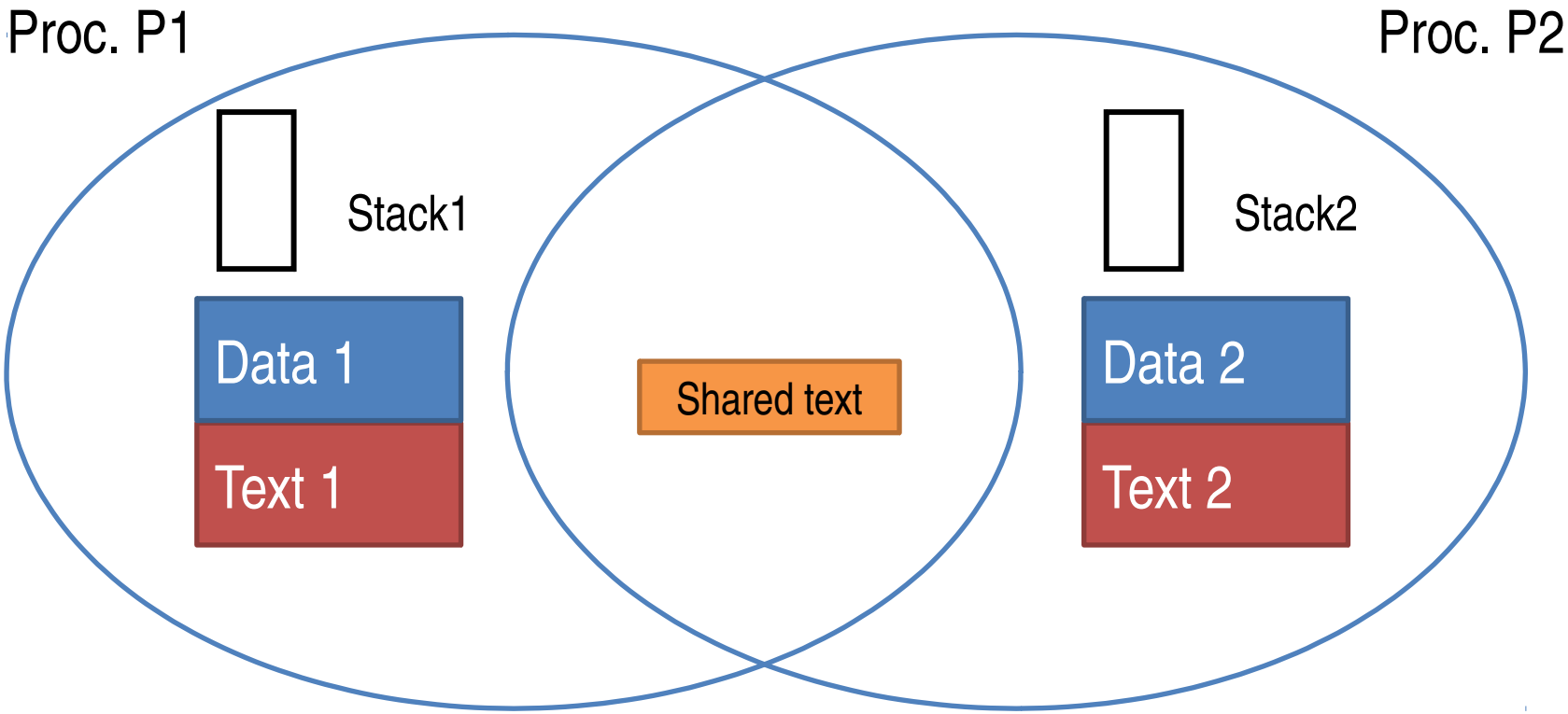
- competition, cooperation

Local environment

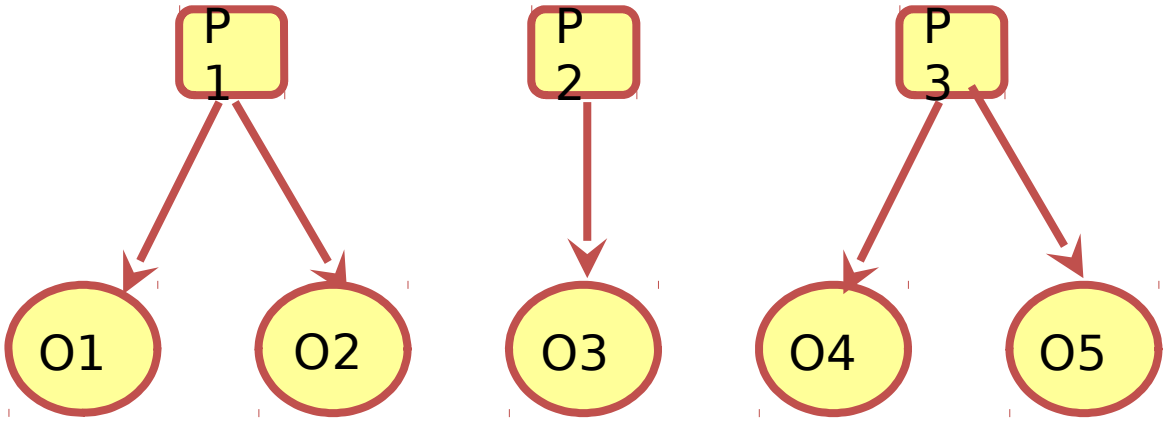




Local environment

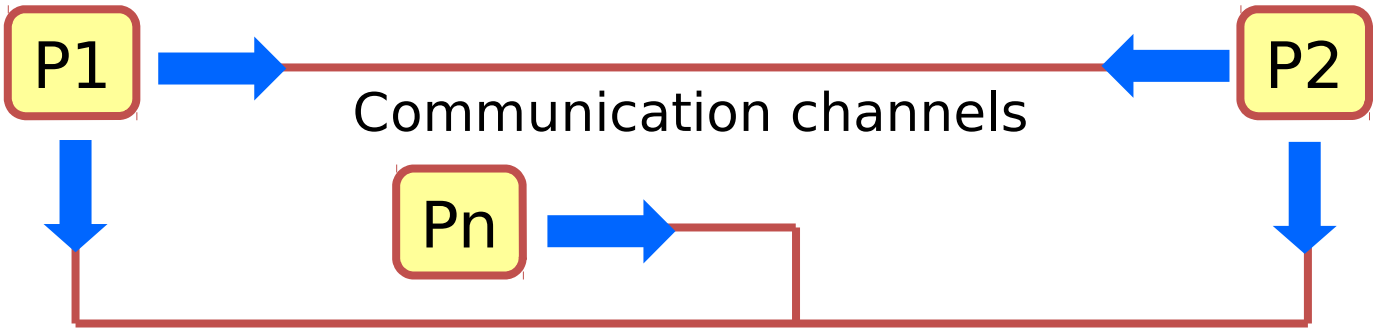


Local environment



O1-O5 are private objects
Competition through server processes
Cooperation through communication

Local environment



Cooperation (communication, synchronization) by means of message passing

Synchronization

Synchronization Motivation

Thread 1

```
p = someFn();
isInitialized = true;
```

Thread 2

```
while (! isInitialized ) ;
q = aFn(p);

if q != aFn(someFn())
   panic
```

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Definitions

Race condition: output of a concurrent program depends on the order of operations between threads

Mutual exclusion: only one thread does a particular thing at a time

- **Critical section:** piece of code that only one thread can execute at once

Lock: prevent someone from doing something

- Lock before entering critical section, before accessing shared data
- unlock when leaving, after done accessing shared data
- wait if locked (all synch involves waiting!)

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if !note
    if !milk {
        leave note
        buy milk
        remove note
    }
```

Too Much Milk, Try #2

Thread A	Thread B
leave note A	leave note B
if (!note B) {	if (!noteA){
if (!milk)	if (!milk)
buy milk	buy milk
}	}
remove note A	remove note B

Too Much Milk, Try #3

Thread A

```
leave note A
while (note B) // X
   do nothing;
if (!milk)
   buy milk;
remove note A
```

Thread B

```
leave note B
if (!noteA){   // Y
  if (!milk)
    buy milk
  }
remove note B
```

Can guarantee at X and Y that either:

(i) Safe for me to buy

(ii) Other will buy, ok to quit

Lessons

- Solution is complicated
  - “obvious” code often has bugs
- Modern compilers/architectures reorder instructions
  - Making reasoning even more difficult
- Generalizing to many threads/processors
  - Peterson’s algorithm: even more complex

Locks

- lock_acquire
  - wait until lock is free, then take it
- lock_release
  - release lock, waking up anyone waiting for it

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Too Much Milk, #4

Locks allow concurrent code to be much simpler:

```
lock_acquire()
if (!milk) buy milk
lock_release()
```

- How do we implement locks? (Later)
  - Hardware support for read/modify/write instructions

Lock Example: Malloc/Free

```
char *malloc (n) {
  lock_acquire(lock);
  p = allocate memory
  lock_release(lock);
  return p;
}
```

```
void free(char *p) {
   lock_acquire(lock);
   put p back on free list
   lock_release(lock);
}
```

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Will this code work?

```
if (p == NULL) {
   lock_acquire(lock);
   if (p == NULL) {
      p = newP();
   }
   release_lock(lock);
}
use p->field1
```

```
newP() {
    p =
malloc(sizeof(p));
    p->field1 = ...
    p->field2 = ...
    return p;
}
```

Lock example: Bounded Buffer

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < last) {
    item = buf[front % size]
    front++;
  }
  lock.release();
  return item;
}
```

```
tryput(item) {
  lock.acquire();
  if ((last – front) < size) {
    buf[last % size] = item;
    last++;
  }
  lock.release();
}
```

Initially: front = last = 0; lock = FREE; size is buffer capacity

Condition Variables

- Called only when holding a lock

- Wait: atomically release lock and relinquish processor until signalled
- Signal: wake up a waiter, if any
- Broadcast: wake up all waiters, if any

Example: Bounded Buffer

```
get() {
 lock.acquire();
 while (front == last)
     empty.wait(lock);
 item = buf[front % size]
 front++;
 full.signal(lock);
 lock.release();
 return item;
}
```

```
put(item) {
  lock.acquire();
  while ((last – front) == size)
    full.wait(lock);
  buf[last % size] = item;
  last++;
  empty.signal(lock);
  lock.release();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are condition variables

Pre/Post Conditions

- What is state of the bounded buffer at lock acquire?
  - front <= last
  - front + buffer size >= last
  - (also true on return from wait)
- Also true at lock release!
- Allows for proof of correctness

Condition Variables

- ALWAYS hold lock when calling wait, signal, broadcast
  - Condition variable is sync FOR shared state
  - ALWAYS hold lock when accessing shared state
- Condition variable is memoryless
  - If signal when no one is waiting, no op
  - If wait before signal, waiter wakes up
- Wait atomically releases lock
  - What if wait, then release?
  - What if release, then wait?

Condition Variables, cont'd

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait())
    condition.Wait(lock);
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Java Manual

When waiting upon a Condition, a “spurious wakeup” is permitted to occur, in general, as a concession to the underlying platform semantics. This has little practical impact on most application programs as a Condition should always be waited upon in a loop, testing the state predicate that is being waited for.

Structured Synchronization

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) condition.Wait(lock);
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

Mesa vs. Hoare semantics

- Mesa (in textbook, Hansen)
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

FIFO Bounded Buffer (Hoare semantics)

```
get() {
 lock.acquire();
 if (front == last)
     empty.wait(lock);
 item = buf[front % size];
 front++;
 full.signal(lock);
 lock.release();
 return item;
}
```

```
put(item) {
  lock.acquire();
  if ((last – front) == size)
    full.wait(lock);
  buf[last % size] = item;
  last++;
  empty.signal(lock);
// CAREFUL: someone else ran
  lock.release();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are condition variables

FIFO Bounded Buffer (Mesa semantics)

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

FIFO Bounded Buffer
(Mesa semantics, put() is similar)

```
get() {
 lock.acquire();
 if (front == last) {
     self = new Condition;
     nextGet.Append(self);
     while (front == last)
        self.wait(lock);
     nextGet.Remove(self);
     delete self;
 }
 item = buf[front % size]
 front++;
 if (!nextPut.empty())
    nextPut.first()->signal(lock);
 lock.release();
 return item;
}
```

Initially: front = last = 0; size is buffer capacity
nextGet, nextPut are queues of Condition Variables

Implementing Synchronization

Concurrent Applications

Semaphores Locks Condition Variables

Interrupt Disable Atomic Read/Modify/Write Instructions

Multiple Processors Hardware Interrupts

Implementing Synchronization

Take 1: using memory load/store

- See too much milk solution/Peterson's algorithm

Take 2:

lock.acquire() { disable interrupts }

lock.release() { enable interrupts }

Lock Implementation, Uniprocessor

```
LockAcquire(){
  disableInterrupts ();
  if(value == BUSY){
     waiting.add(current TCB);
     suspend();
  } else {
     value = BUSY;
  }
  enableInterrupts ();
}
```

```
LockRelease() {
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
readyList.Append(thread);
  } else {
value = FREE;
  }
  enableInterrupts ();
}
```

Multiprocessor

- Read-modify-write instructions
  - Atomically read a value from memory, operate on it, and then write it back to memory
  - Intervening instructions prevented in hardware
- Examples
  - Test and set
  - Intel: xchgb, lock prefix
  - Compare and swap
- Does it matter which type of RMW instruction we use?
  - Not for implementing locks and condition variables!

Spinlocks

Lock where the processor waits in a loop for the lock to become free

- Assumes lock will be held for a short time
- Used to protect ready list to implement locks

```
SpinlockAcquire() {
  while (testAndSet(&lockValue) == BUSY)
    ;
}
SpinlockRelease() {
  lockValue = FREE;
}
```

Lock Implementation, Multiprocessor

```
LockAcquire(){
  spinLock.Acquire();
  disableInterrupts ();
  if(value == BUSY){
    waiting.add(current TCB);
     suspend();
  } else {
     value = BUSY;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

```
LockRelease() {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
readyList.Append(thread);
  } else {
value = FREE;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

Lock Implementation, Linux

- Fast path
  - If lock is FREE, and no one is waiting, test&set
- Slow path
  - If lock is BUSY or someone is waiting, see previous slide

- User-level locks
  - Fast path: acquire lock using test&set
  - Slow path: system call to kernel, to use kernel lock

Semaphores

- Semaphore has a non-negative integer value
  - P() atomically waits for value to become > 0, then decrements
  - V() atomically increments value (waking up waiter if needed)
- Semaphores are like integers except:
  - Only operations are P and V
  - Operations are atomic
  - If value is 1, two P's will result in value 0 and one waiter
- Semaphores are useful for
  - Unlocked wait: interrupt handler, fork/join

Semaphore Bounded Buffer

```
get() {
  empty.P();
  mutex.P();
  item = buf[front % size]
  front++;
  mutex.V();
  full.V();
  return item;
}
```

```
put(item) {
  full.P();
  mutex.P();
  buf[last % size] = item;
  last++;
  mutex.V();
  empty.V();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores

Implementing Condition Variables using Semaphores (Take 1)

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  sem.V();
}
```

Implementing Condition Variables using Semaphores (Take 2)

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if semaphore is not empty
     sem.V();
}
```

Implementing Condition Variables using Semaphores (Take 3)

```
wait(lock) {
  sem = new Semaphore;
  queue.Append(sem);   // queue of waiting threads
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if !queue.Empty()
     sem = queue.Remove();
     sem.V();      // wake up waiter
}
```

Synchronization Summary

- Use consistent structure
- Always use locks and condition variables
- Always acquire lock at beginning of procedure, release at end
- Always hold lock when using a condition variable
- Always wait in while loop
- Never spin in sleep()

Caso di studio: libreria p-thread in Unix

(materiale aggiuntivo)

Lo standard POSIX per i thread

- UNIX, nella versione nativa, non prevede multithreading e non dispone di chiamate di sistema per i threads
  - in particolare: *fork* genera *processi* con spazi di indirizzamento disgiunti

  ==> i thread sono realizzabili a livello utente
- LINUX è un sistema con multithreading e dispone di chiamate di sistema per i threads
  - in particolare chiamata *clone:*
  - genera un thread figlio che condivide (in parte) lo spazio di indirizzamento del padre
  - se lo spazio condiviso è vuoto, genera un processo
  - ogni processo ha un thread iniziale che può generare altri thread
  - l'unità di schedulazione è il thread
  - Programmi che utilizzano le chiamate per i thread, e in particolare la *clone,* non sono portabili su UNIX

Lo standard POSIX per i thread

- Per garantire la portabilità, POSIX definisce un insieme di funzioni standard per i thread
  - realizzabili con librerie *(livello delle librerie standard)*
  - differenti realizzazioni per UNIX e LINUX

Libreria p-thread: sincronizzazione

- *Mutex*: meccanismo per la mutua esclusione nello standard POSIX 1003.1c:
  - dati del tipo `pthread_mutex_t`
  - sono semafori binari (valore 0 --> *occupato*, valore 1 --> *libero*);
  - operazioni fondamentali:
  - inizializzazione: `pthread_mutex_init`
  - locking: `pthread_mutex_lock`
  - unlocking: `pthread_mutex_unlock`

Libreria p-thread: inizializzazione di un *mutex*

- L'inizializzazione di un `mutex` si può realizzare con:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
   pthread_mutexattr_t *attr)
```

dove:

`mutex:` individua il mutex da inizializzare

`attr:` punta a una struttura che contiene gli attributi del mutex; se NULL, il mutex viene inizializzato a *libero.*

attribuisce un valore iniziale all'intero associato al semaforo (default: *libero*):

Libreria p-thread: operazioni *lock/unlock*

```
int pthread_mutex_lock(pthread_mutex_t *mux);

int pthread_mutex_unlock(pthread_mutex_t *mux);
```

- lock: se il mutex *mux* è occupato, il thread chiamante si sospende; altrimenti occupa il mutex.
- unlock: se vi sono thread in attesa del mutex *mux*, ne risveglia uno; altrimenti libera il mutex.

Libreria p-thread: : variabili *condition*

- *condition*: meccanismo per la sincronizzazione nello standard POSIX 1003.1c:

- dato del tipo `pthread_cond_t`
  - analogamente al semaforo, ha una coda sulla quale si sospendono i thread
- a differenza dei semafori, non ha un valore che condiziona la sospensione;
  - sono associate operazioni per la sospensione e la riattivazione *incondizionata* dei thread

==> *ma: la sospensione può essere condizionata dal valore di un'espressione valutata esplicitamente dal thread.*

Libreria p-thread: inizializzazione di una *condition*

- L'inizializzazione di una *condition* si realizza con:

```
Int pthread_cond_init(pthread_cond_t *cond,
    pthread_cond_attr_t *cond_attr)
```

dove:

`cond` : individua la condizione da inizializzare

`attr` : punta a una struttura che contiene gli *attributi* della condizione; se NULL, viene inizializzata a default.

Libreria p-thread: uso delle *condition*

- Per l'utilizzo delle variabili *condition* sono disponibili le operazioni:

`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mux)`

– Sospende incondizionatamente il thread sulla *condition* cond

==> Si può realizzare la sospensione condizionata, condizionando *l'esecuzione* alla preventiva valutazione di una *condizione di accesso.*

`pthread_cond_signal(pthread_cond_t *cond)`

– Se esistono thread sospesi sulla *condition*, riattiva il primo; altrimenti non ha effetto

Libreria p-thread: operazione *wait*

- Il meccanismo `pthread_cond_wait` è combinato con la preventiva valutazione di una *condizione di accesso* per realizzare la sospensione condizionata
- La condizione di accesso dipende generalmente da variabili comuni e pertanto deve essere eseguita in mutua esclusione

==>> necessario associare un *mutex* alla variabile *condition*

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mux)
```

dove:

- `cond`: è la variabile *condition*
- `mux`: è il *mutex* associato

Effetto:

- il thread chiamante si sospende sulla coda `cond`, *rilasciando automaticamente il mutex `mux`*
- alla successiva riattivazione, il thread *riacquisisce automaticamente `mux`*.

Libreria p-thread: operazione *signal*

- Riattivazione di un thread che si era sospeso sulla *condition* cond associata al *mutex* `mux`:

```
int pthread_cond_signal(pthread_cond_t *cond)
```

Effetto:

a) se esistono thread sospesi nella coda associata a `pthread_cond_signal`, viene selezionato il **`primo`**;

- viene identificato il mutex associato **`mux`**;

- viene riattivato il thread **`primo`**, che riacquisisce automaticamente la mutua esclusione

b) se non esistono thread sospesi nella coda associata a **`cond`**, l'operazione `pthread_cond_signal` non produce nessun effetto

Esempio: uso combinato di condition e mutex

Problema: risorsa utilizzabile da più thread, fino a un massimo di *max (1)*

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max)
    pthread_cond_wait(&max_utenti,&mux);
utilizzatori++;
pthread_mutex_unlock(&mux);
```

```
<uso della risorsa>
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: normalmente eseguita in mutua esclusione */
pthread_mutex_lock(&mux);
utilizzatori--;
pthread_cond_signal(&max_utenti);
pthread_mutex_unlock(&mux);
```

Note sull'implementazione dei meccanismi

1) Nella `signal`, associazione di `mux` a `max_utenti` indotta dalla `wait`
2) Nella fase di uscita la `unlock` (se eseguita dopo la `signal`) non rilascia `mutex` (vedere slide seguente)

Esempio: uso combinato di condition e mutex

Note sull'implementazione dei meccanismi

```
(max_utenti: condition, mux: mutex)
/*Fase di Entrata: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
while(utilizzatori==max)
    pthread_cond_wait(&max_utenti,&mux); pthread_mutex_lock(&mux);
utilizzatori++;
pthread_mutex_unlock(&mux);
```

```
<uso della risorsa>
/* permesso a non più di max utenti senza mutua esclusione */
```

```
/*Fase di Uscita: necessaria la mutua esclusione*/
pthread_mutex_lock(&mux);
utilizzatori--;
pthread_cond_signal(&max_utenti);
pthread_mutex_unlock(&mux);
```

La riacquisizione automatica di `mutex` *da parte del thread riattivato da* `pthread_cond_signal` *realizzata con l'esecuzione implicita dell'operazione* `pthread_mutex_lock(&mux);`

`while` *sostituibile con* `if` *se questa operazione e la riattivazione sono indivisibili*

Ma: `while` *necessario per evitare effetti indesiderabili dei segnali rientranti (--> laboratorio)*

Interprocess communication in the local environment model
- the case of Unix -

Communication mechanisms in Unix

- Unix (processes):
  - Asynchronous events: *signals*
  - Message passing: *pipe*

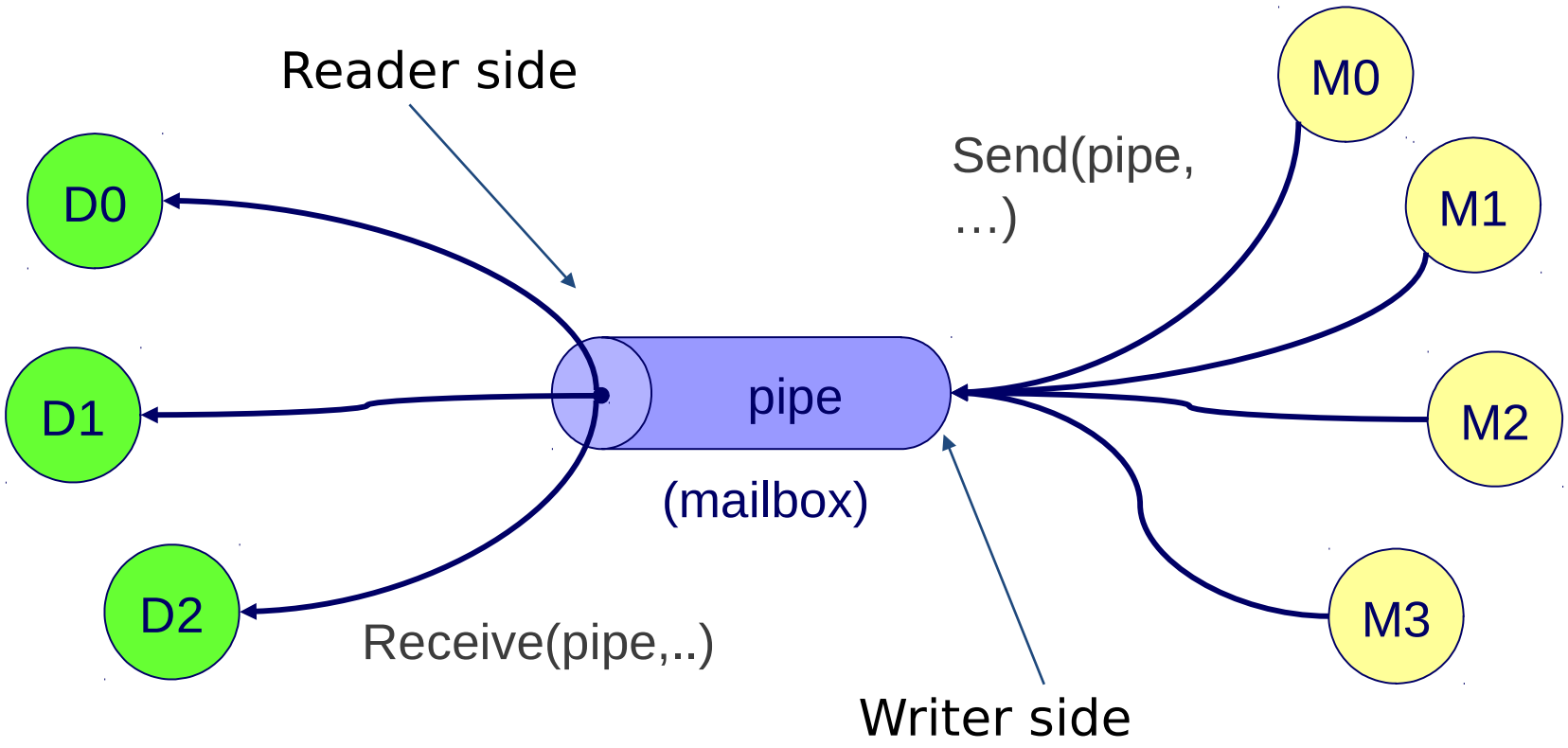
- Unix (Pthreads):
  - Condition variables

Pipes

- A pipe is a communication channels:
  - Many to many
  - Unidirectional
  - FIFO policy
  - Variable length messages
  - Limited capacity
  - The buffer pipe may get full
  - Implements a producer/consumer model

Pipes

- Indirect communication



pipe system call

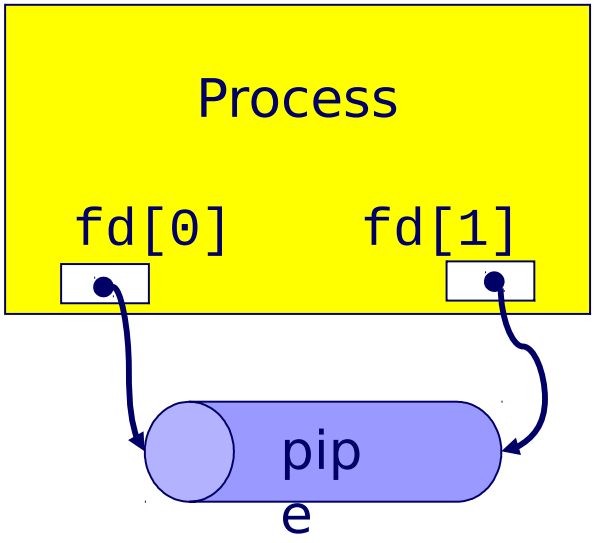
- to create a pipe:

```
int pipe(int fd[2]);
```

  - `fd` is the pointer to a vector of two file descriptors, which are initialized as follows (only in case of success):
    - `fd[0]` represents the read side of the pipe
    - `fd[1]` represents the write side of the pipe
- The system call pipe returns:
  - A negative integer, in case of failure
  - 0, in case of success

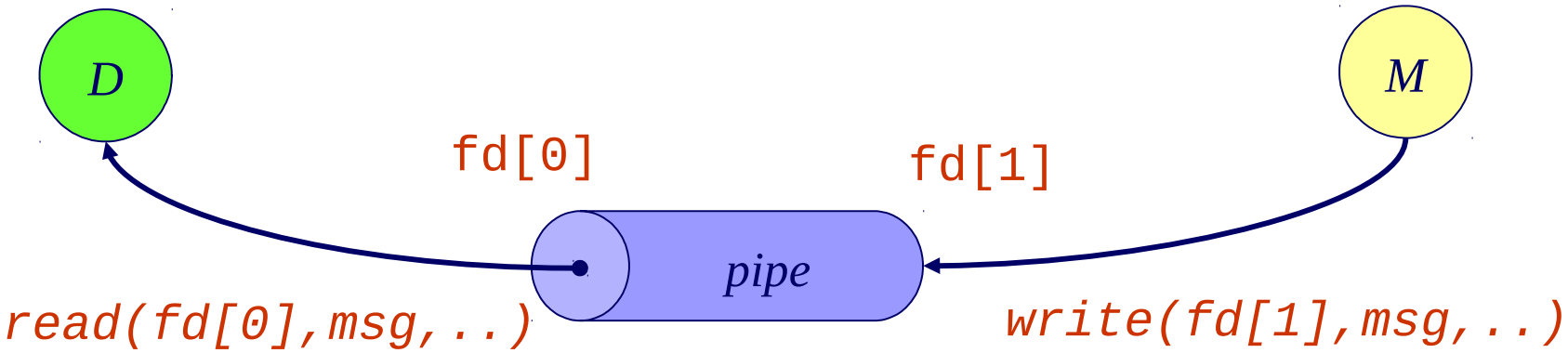
pipe system call

- If `pipe(fd)` succeeds:
  - `fd[0]`: *receive* side of the del pipe
  - `fd[1]`: *send* side of the pipe



Accessing a pipe

- Two system calls, `read`, `write`:



  - `read:` implements a *receive*
  - `write:` implements a *send*

Which processes can communicate through a pipe?

- For sender and receiver the reference to the channel is a *file descriptor*
- The pipe is inherited by the children (and the descendants)
- Hence only the processes belonging to the same hierarchy can share the same pipe:
  - A process and its children
  - “brother” processes that inherit the pipe from the father
  - “grandpa” and “grandchild”
  - ...

Process synch. through a pipe

- A pipe has limited capacity:
  - If the pipe is empty: the reader waits
  - If the pipe is full: the writer waits
- Hence `read` e `write` from/to a pipe are blocking !

Closing a pipe

- Each process can close one side of a pipe by means of a `close`.
- close applies only to the process that invokes it
- A side of a pipe is definitively closed when all the process that share the pipe close that side.

pipe: an example

- Implementation of the shell command
  `sort<f | head`

  - Shell invokes fork and creates F1
  - F1 invokes exec and replaces its code with that of `sort`;
  - F1 creates a pipe
  - F1 invokes a fork to create F1.1, which inherits the pipe from F1
  - F1.1 invokes exec and replaces its code with that of `head`. In any case F1.1 keeps the access to the pipe
  - F1 closes the read side of the pipe, takes input from *f* and writes on the pipe
  - F1.1 closes the write side of the pipe and reads from the pipe

pipes: conclusions

- Pipes have two drawbacks:
  - Allow interactions only among processes in the same hierarchy
  - Pipes are not persistent: when the processes that use a pipe terminate the pipe is destroyed
- Different abstractions to enable communications among processes in different hierarchies:
  - *FIFO* (system V)
  - *socket*

Signals

- Signals are upcalls in Unix
  - Think about software interrupts
  - Define asynchronous events
- Signals can be sent to one or more processes
- Signals can be sent by:
  - a process by means of the system call *kill*
  - the kernel, as a consequence of different events:
  - Interrupts (for example of the timer)
  - Exceptions (violations of protection policies, errors of the process etc.)

System call `kill`

- The system call *kill* is used to send signals to other processes

```
int kill(int pid, int sig);
```

  - `sig` is an integer code that represents the signal
  - `pid` specifies the ID of the recipient of the signal:
    - pid> 0: the recipient is the process identified by pid
    - pid=0: ithe signal is sent to all processes in the group of the sender
    - pid <-1: the signal is sent to all processes with *groupId* equal to – pid
    - pid== -1: not specified in Posix (different behaviours are possible)

System call `signal`

- Used to assign an handler to a signal:

```
void signal(int sig, void *func);
```

  - `sig` is the integer code of the signal to be associated with an handler
  - `func` is a pointer to the handler function or defines a behaviour for the signal:
    - `func` points to the signal handler
    - `func = SIG_IGN` (means ingnore signal)
    - `func = SIG_DFL` (associate the default action to the signal)
  - The system call sssigns to `func` :
    - The pointer to the previous handler
    - SIG_ERR(-1), in case of error

Signal handler:

- Takes an integer that represents the signal code
- Does not returns any value

```
void handler(int sign)
{   ....
    ....
    return;
}
```

System call pause

- Used to suspend a process waiting for a signal
- The process is activated again at the reception of a signal

Signals: data structures

- *signal handler array* :
  - In the process PCB (specifically ,in the user structure)
  - Defines what to do with the signal
  - Ignore
  - Default action
  - Execute handler (in this case it contains the pointer to the handler)
- *pending signal bitmap (signal mask)*:
  - In the process PCB (specifically in the process structure)
  - One bit for each kind of signal:
  - If bit is set (=1) the corresponding signal is pending
- *Signal handler array* and *pending signal bitmap:*
  - are inherited from the father by the fork;
  - The exec system calls resets these structures

Signals: behavior

- When a signal is sent to a process, the kernel sets the corresponding bit in the pending signal bitmap
  - Several signals of the same kind can be seen as just one
- If the process is being executed:
  - the signal is recognized immediately
- If the process is ready or blocked:
  - The signal is recognized as soon as the process is executed again
- A process can wait any signal with the system call *pause*

Signals: behavior

- When a pending signal is recognized by a process:
  - The kernel executes the action defined in the signal handler array
  - either invokes the signal handler of the process
  - or ignores the signal
  - or executes the default action (that typically kills the process)
  - If the kernel invokes the handler:
  - This corresponds to making an upcall (see slide set: 02- kernel)
  - At the end of the handler the process continue the execution from the point where it was interrupted