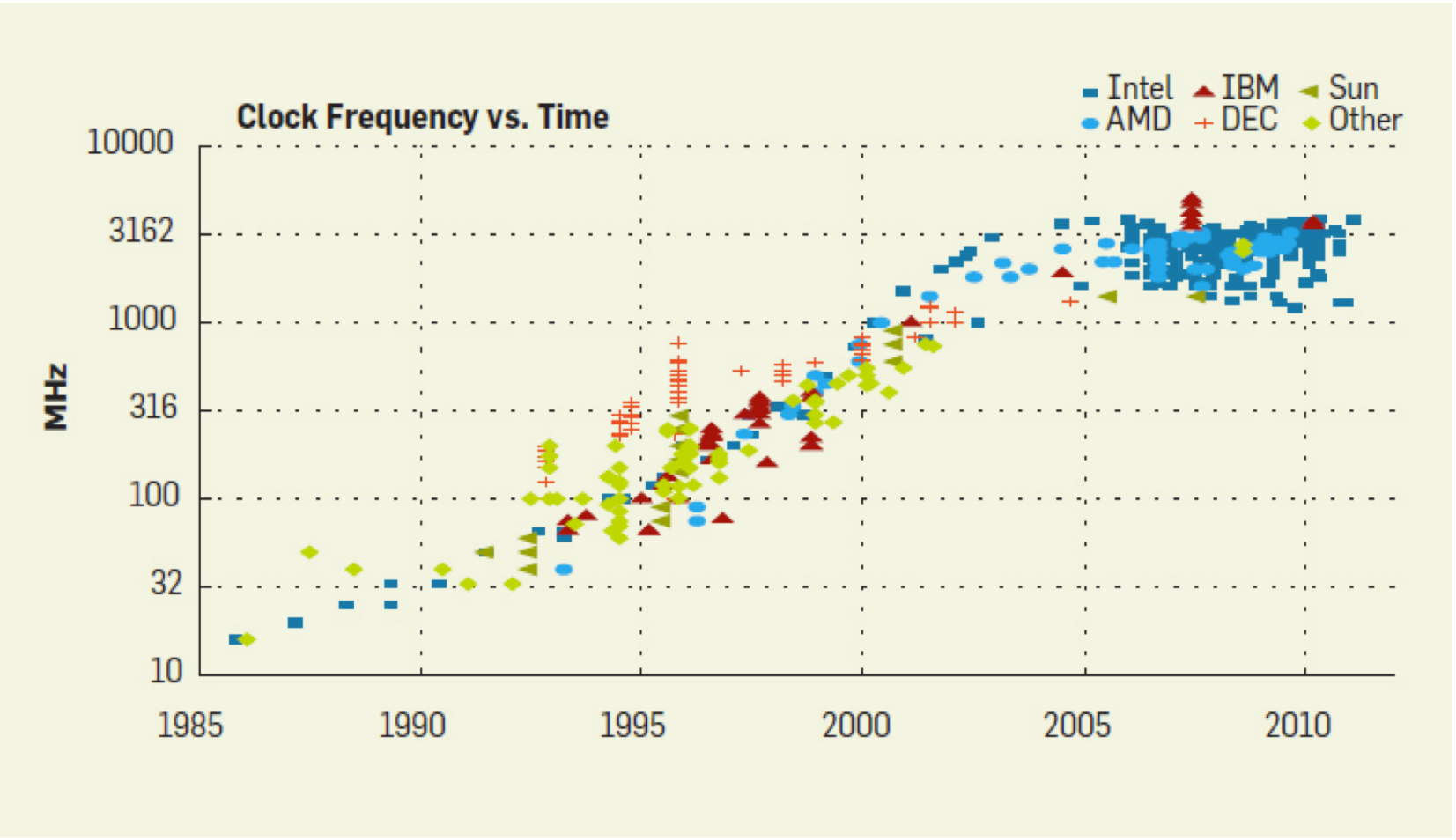
Operating Systems

4. Concurrency

Pablo Prieto Torralbo

Department of Computer Engineering and Electronics

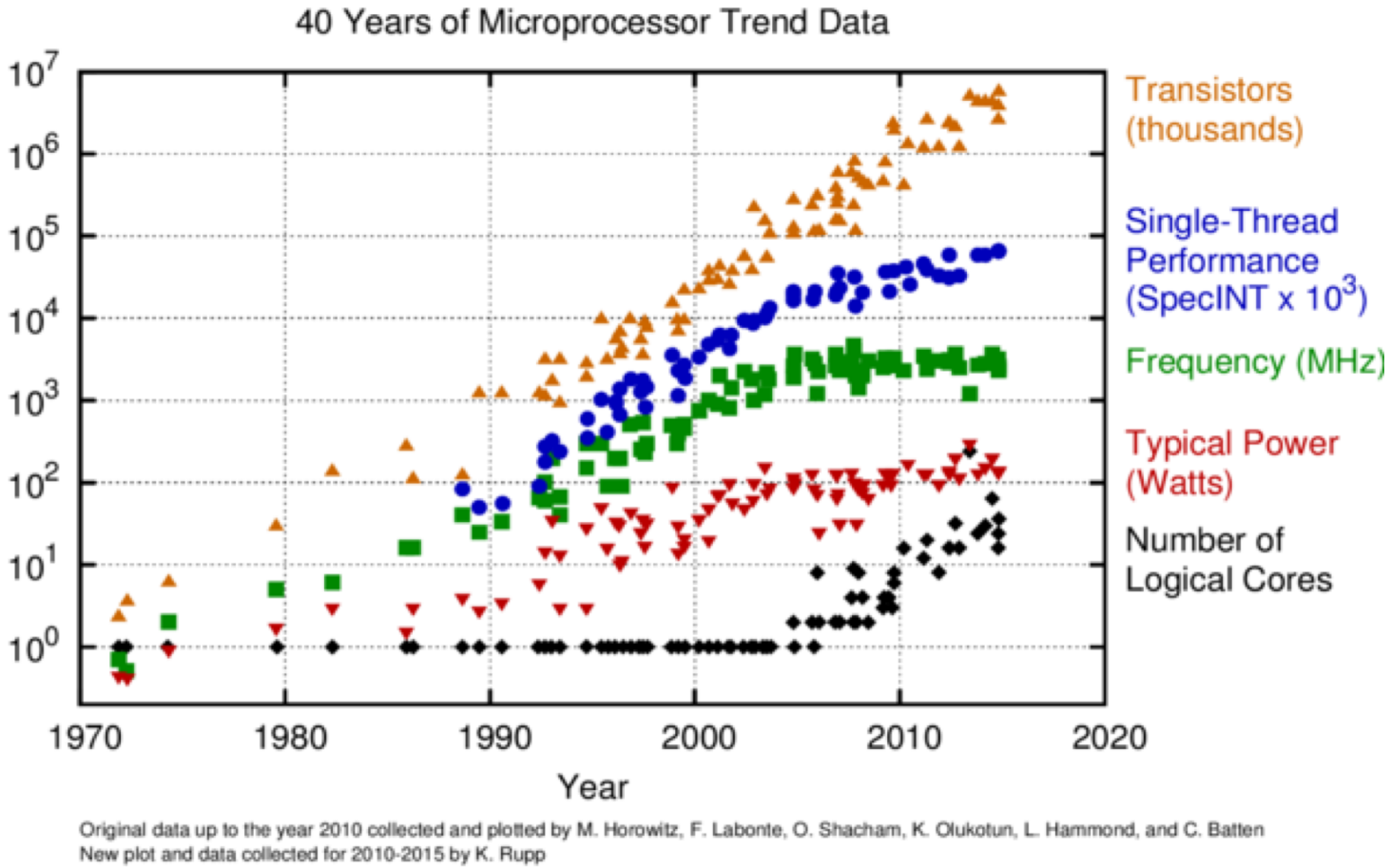
Motivation



CPU DB: Recording Microprocessor History

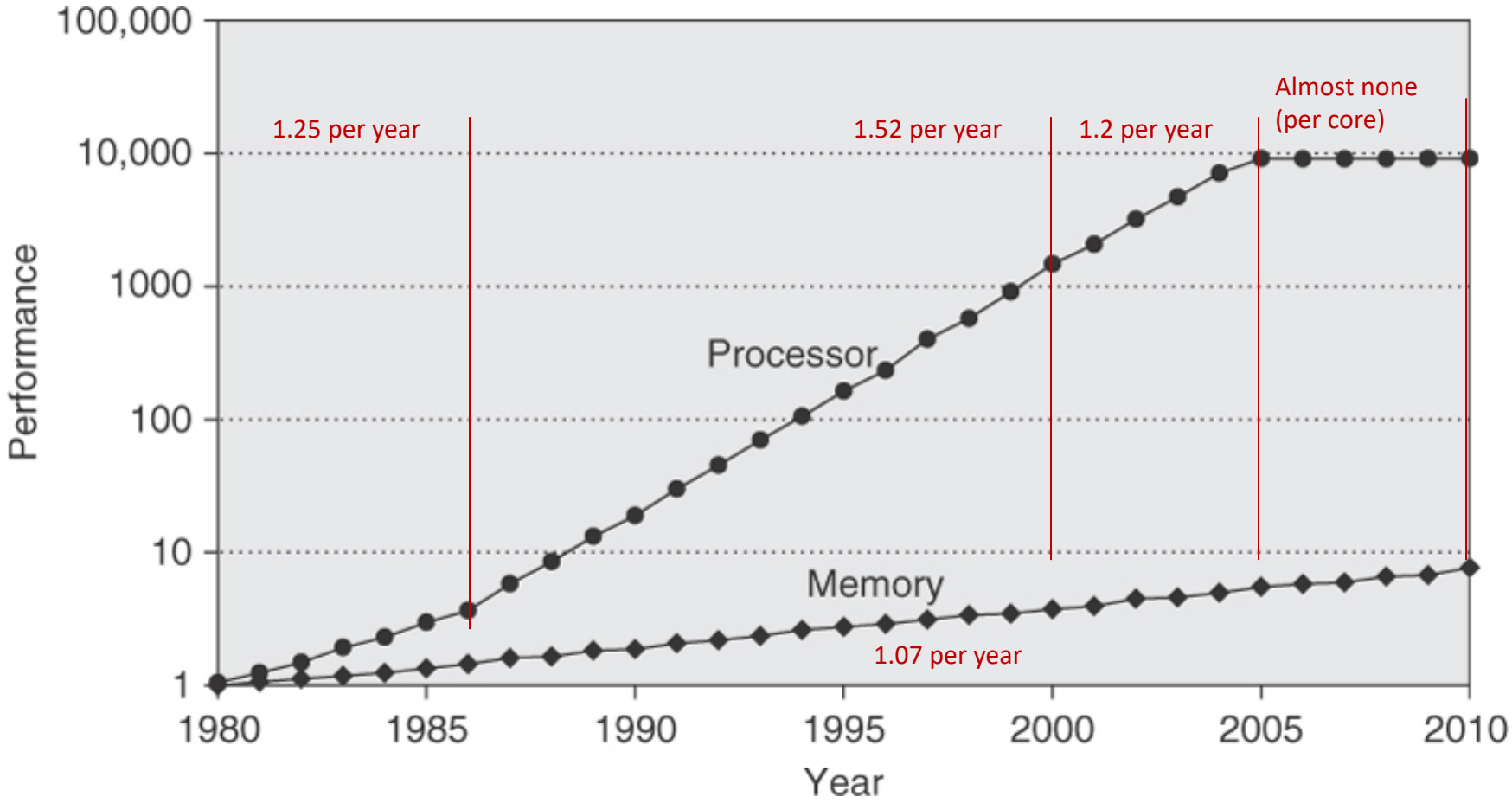
Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, Mark Horowitz Communications of the ACM, Vol. 55 No. 4, Pages 55-63

Motivation



By Karl Rupp
https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/
Original Data up to year 2010 by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten.

Motivation



Motivation

- The future:
  - ~same speed
  - more cores

- Goal
  - Faster programs need write applications that fully utilize many CPUs → concurrent execution.
  - Need communication.

4. Concurrency
4.1 Communication

Process Communication

- Advantages of Process Cooperation:
  - Information Sharing
  - Computation Speed-up (multiple-CPUs)
  - Modularity (dividing functions into separate threads)
  - Convenience (editing, listening to music, compiling… in parallel)

- Communication between processes has two components: information and synchronization.

- Basic techniques:
  - Shared memory (easy programming and synchronization problems).
  - Message passing.

Message Passing

- Based on two primitives:
  - Send(destination, message)
  - Receive(source, message)
- Synchronization
  - Blocking send, Blocking receive
    - Wait until both are ready
  - Non-blocking send, Blocking receive
    - Sender doesn't wait, but receiver does.
    - Useful in sending multiple messages
    - Use replies to verify message receipt
  - Non-blocking send and receive
    - If no messages, returns immediately
  - Can allow test for waiting messages
- Addressing:
  - Direct: Provide ID of destination
  - Indirect: Send to a mailbox
    - Can vary mailbox/process matching

Shared-Memory

- Another way of communication: sharing memory.

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms (synchronization) to ensure orderly execution of cooperating processes.

- Main options for shared memory:
  - Multiple processes.
  - Multi-threads.

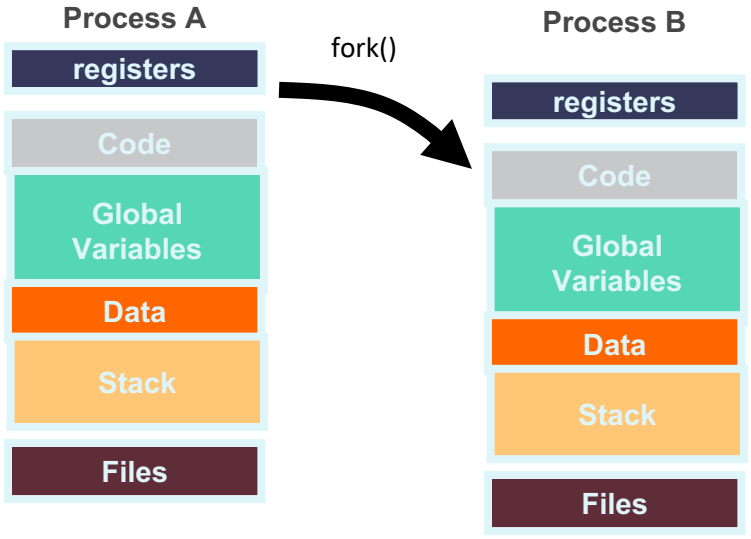
Shared-Memory

- Option 1: Build applications from many communicating processes.
  - like Chrome (one process per tab)
  - communicate via pipe() or similar (shared memory regions on each process address space)
  - Pros/cons?
    - ✓ don't need new abstractions
    - ✗ cumbersome programming
    - ✗ copying overheads
    - ✗ expensive context switching

- Option 2: New abstraction: the thread.
  - Threads are just like processes, but they share the address space (e.g., using same PT).

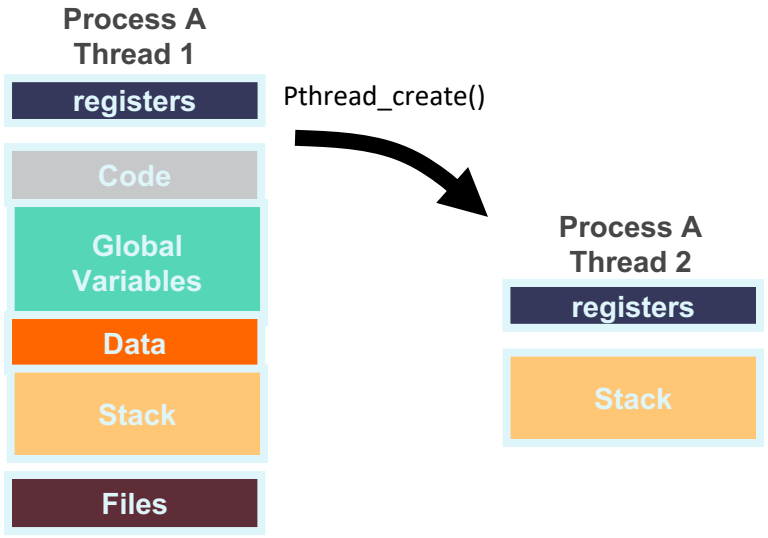
Threads

- Instead of a single point of execution within a program, multi-thread.
  - Each thread is a like a separate process, but sharing the same address space, sharing files...→ lightweight process.
  - Similar state to a process (PC, private set of registers…) → switching threads within a processor means a context switch (not changing the address space but the stack).
    - Thread Control Blocks (TCBs) for each thread of a process.
    - Context switches are less expensive than between processes.



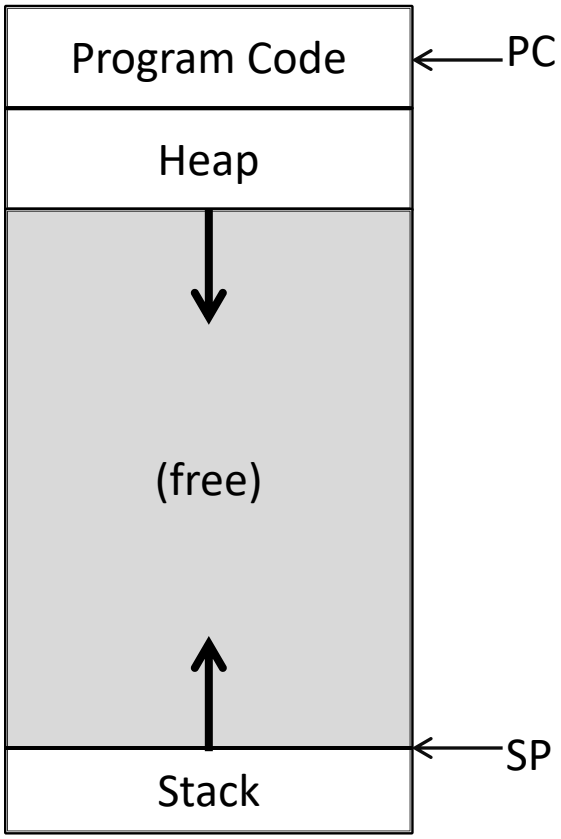
Threads

- Instead of a single point of execution within a program, multi-thread.
  - Each thread is a like a separate process, but sharing the same address space, sharing files...→ lightweight process.
  - Similar state to a process (PC, private set of registers…) → switching threads within a processor means a context switch (not changing the address space but the stack).
    - Thread Control Blocks (TCBs) for each thread of a process.
    - Context switches are less expensive than between processes.

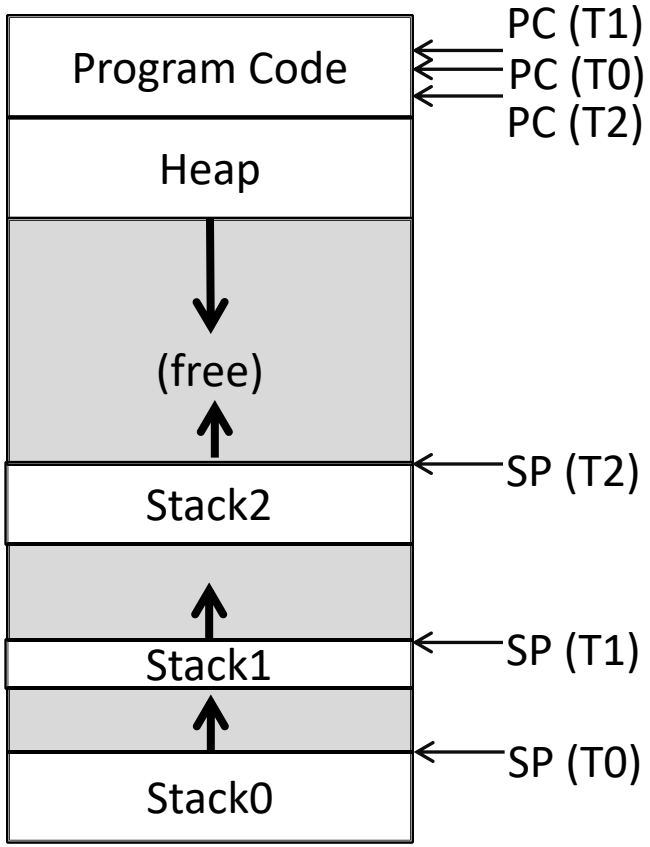


Threads

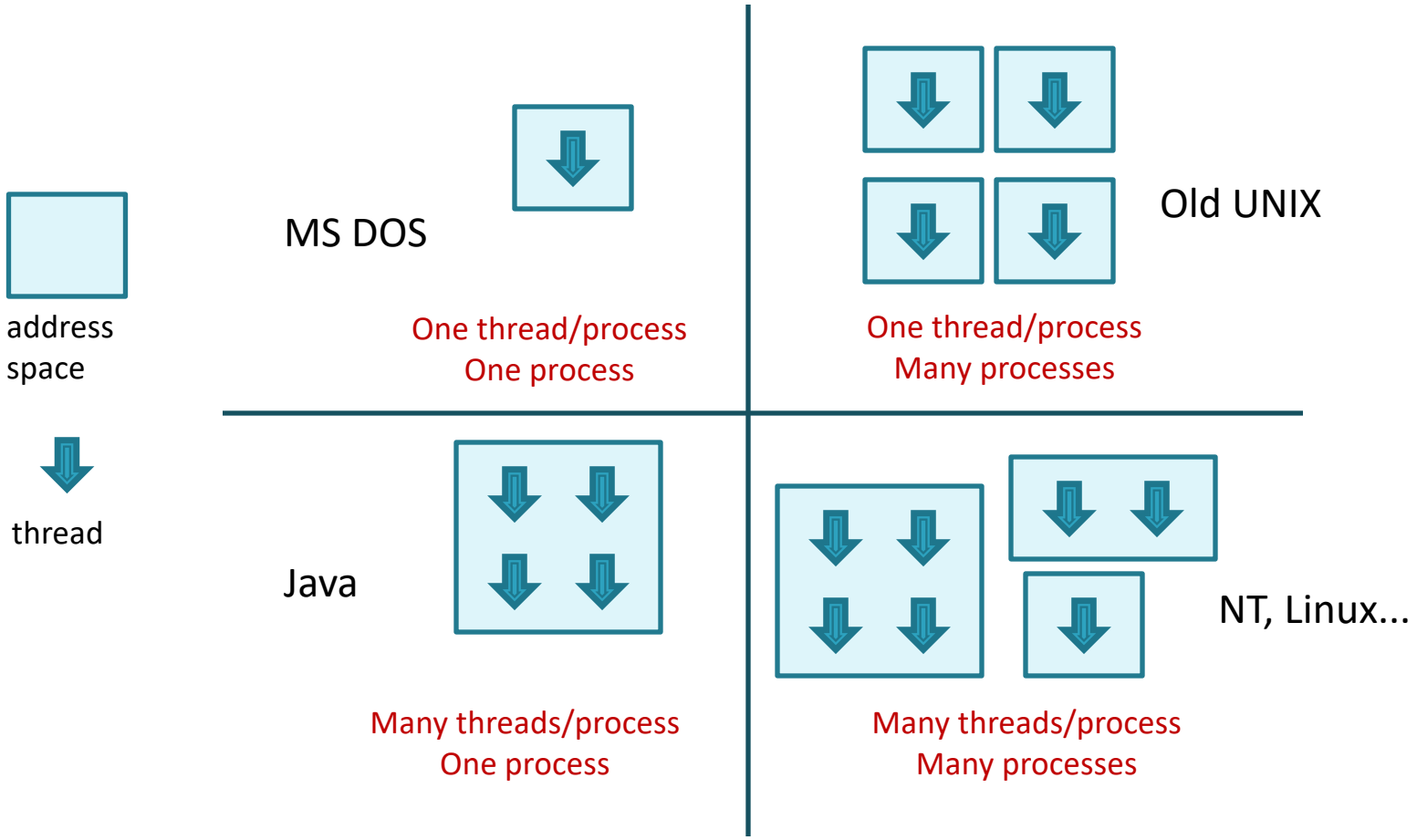
(Old) Process Address Space



Process Address Space with Threads



Design Space



Shared-Memory Example

- The producer-consumer problem (Bounded-Buffer):

```
Shared data:
            type item;
            var buffer array[BUFFER_SIZE] of item;
            int in, out, counter;
            counter=0;
            in=0; out=0;
Producer process:
            do {
               …
               nextp=produce_item();
               …
               while(counter==BUFFER_SIZE) ;
               buffer[in]=nextp;
               in=(in+1)%BUFFER_SIZE;
               counter++;
            } while(true);
Consumer process:
            do {
               while(counter==0) ;
               nextc=buffer[out];
               out:=(out+1)%BUFF_SIZE;
               counter--;
               …
               consume_item(nextc);
               …
            }while(true);
```

Spinlock

Spinlock

WARNING: Shared!
(Critical Section)

Critical-Section Problem

- N processes competing to use some shared data.
  - Each process has a code segment (critical section) in which the shared data is accessed.
  - Concurrency leads to non-deterministic bugs called race conditions.
  - Passing tests once mean little

Example:

```
mov 0x123, %eax
add $0x1, %eax
mov %eax, 0x123
```

```
Thread 1
mov %eax, 0x123
add %0x1, %eax

mov %eax, 0x123
```

```
Thread 2


mov %eax, 0x123

add %0x1, %eax
mov %eax, 0x123
```

Race condition depends on CPU schedule

Expect the worst!

- Many kernel-mode processes can be active in the OS at a time:
  - Subject to race conditions.

Critical-Section Solution

- Requirements:
  - Mutual exclusion: ensure that when one process is executing in its critical section, no other process is allowed to execute its critical section.
  - Progress: if no process is executing in its critical section and there are processes waiting to do so, the selection cannot be postponed indefinitely.
  - Bounded waiting: A limit must exist on the number of times other processes are granted access to the critical section after a process request to enter and before the request is granted.

- Assume that each process executes at non-zero speed.
  - No assumption concerning relative speeds.

Initial Attempts

- Algorithm 1
  - Shared variables:

```
int turn;
turn = 0;
```

  - Process P_i

```
while(continue)
{
    while(turn != i) {; //do-nothing}
    ----  Critical Section ----
    turn = j;
    ---- Non-critical Section ----
}
```

- Satisfies mutual exclusion, but not progress (if turn=0 and P_1 is ready to enter its critical section, P_1 cannot do so, even if P_0 is in the Non-Critical Section).

Initial Attempts

- Algorithm 2
  - Shared variables:

```
bool flag[NUM_PROC];
for(i=0; i<NUM_PROC; i++)
{  flag[i]=false; }
```

`flag[i]==true` means P_i is ready to enter its critical section

  - Process P_i

```
while(continue)
{
    flag[i]=true;
    while(flag[j]) {; //do-nothing}
    ----  Critical Section ----
    flag[i]=false;
    ---- Non-critical Section ----
}
```

- Satisfies mutual exclusion, but not progress (P_0 sets `flag[0]=true` and P_1 sets `flag[1]=true`, waiting for each other forever).

Initial Attempts

- Algorithm 3 (Dekker and Peterson)
  - Shared variables (combined algorithm 1 and 2):

```
int turn=0;
bool flag[2];
flag[0]=false;
flag[1]=false;
```

  - Process P_i

```
while(continue)
{
    flag[i]=true;
    turn = j;
    while(flag[j] && turn==j) { ; //do-nothing}
    ----  Critical Section ----
    flag[i]=false;
    ---- Non-critical Section ----
}
```

- Satisfies all three requirements and solves the critical section for two processes.
  - This algorithm works (without fence instructions) when sequential consistency is assumed.

Heart of the Problem

`counter=counter+1` is not atomic

```
mov 0x123, %eax
add $0x1, %eax
mov %eax, 0x123
```

- Uncontrolled Scheduling
  - One thread can enter that region of code to increment counter by one and be interrupted after loading the value to eax but before storing it back.
  - The contents of its register get saved in the TCB (eax included).
  - The new thread loads counter, but gets the unmodified value.
  - When the first thread comes back, it works with the previously loaded value (the one stored in eax).

- Controlling Interrupts
  - Disable Interrupts for critical sections.
  - Easily Abused.
  - Only works on systems with a single CPU.
  - Lost Interrupts.

Heart of the Problem

`counter=counter+1` is not atomic

```
mov 0x123, %eax
add $0x1, %eax
mov %eax, 0x123
```

- Uncontrolled Scheduling
  - One thread can enter that region of code to increment counter by one and be interrupted after loading the value to eax but before storing it back.
  - The contents of its register get saved in the TCB (eax included).
  - The new thread loads counter, but gets the unmodified value.
  - When the first thread comes back, it works with the previously loaded value (the one stored in eax).

- One way to solve the problem: Atomicity → Hardware.
  - To have more powerful instructions that, in one single step, do whatever we need (so an interrupt cannot occur in the middle).
  - Instead of having atomic instructions for everything we need to do in critical sections (costly). Build general set of synchronization primitives.

One More Problem: Waiting for Another One

- Access to shared variables is not the only concurrency problem.
- Another problem: when one process must wait for another process to complete before it continues.
- Sleeping/Waking interaction (similar to the interaction when a process performs disk I/O and is put to sleep).
- Condition Variables.

4. Concurrency
4.2 Thread API (POSIX)

Thread Creation

- `#include <pthread.h>`
  - Link with –lpthread (or –pthread)

```
pthread_create( pthread_t *          thread,
                pthread_attr_t *     attr,
                void *               (*start_routine)(void*),
                void *               arg);
```

- Function returns:
  - 0 → everything is ok.
  - ≠0 → there was an error.
- Attribute definition (`pthread_attr_t`)
  - Stack size, priority…
  - Use `pthread_attr_init(&attribute);`
  - Or NULL → default.
- `start_routine` is a `void*` returning function with only one input (`arg`)
  - A `struct` can be used (global).

Thread join & detach

- Each thread can be:
  - "Joinable": depends on its owner (default)
  - "Detached": independent from its owner.

  - Specified in the attributes:
    - `int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);`

    - `detachstate` can be:
      - `PTHREAD_CREATE_DETACHED`
      - `PTHREAD_CREATE_JOINABLE` (**`default`**)

Thread join & detach

- Wait for a thread:

```
int pthread_join(pthread_t thread, void **thread_return);
```

  - This function suspends the calling thread.
  - When the thread finishes, the OS saves the thread ID and the state at `thread_return`.

- Or detach it:

```
int pthread_detach(pthread_t thread);
```

  - Does not suspend the calling thread.
  - Cannot be undone.
  - When the thread exits, resources are liberated (thread cannot be joined).

Thread exiting

- To end a thread we should use:

```
void pthread_exit(void *thread_return);
```

  - Ends the execution of the calling thread.

  - Or

```
return (void*)thread_return;
```

  - Same behavior as above.

  - Or

```
int pthread_cancel(pthread_t tid);
```

  - Ends the execution of the thread with ID: `tid`.

- WARNING: Beware when using `exit()`.

Example

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

typedef struct __myarg_t {
int a;
int b;
} myarg_t;

typedef struct __myret_t {
int x;
int y;
} myret_t;

void *mythread(void *arg) {
   myarg_t *m = (myarg_t *) arg;
   printf("%d %d\n", m->a, m->b);
   myret_t *r = malloc(sizeof(myret_t));
   r->x = 1;
   r->y = 2;
   return (void *) r;
}

int
main(int argc, char *argv[]) {
   int rc;
   pthread_t p;
   myret_t *m;

   myarg_t args;
   args.a = 10;
   args.b = 20;
   pthread_create(&p, NULL, mythread,
   &args);
   pthread_join(p, (void **) &m);
   printf("returned %d %d\n", m->x, m->y);
   return 0;
}
```

Example

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r;
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

ALLOCATED ON STACK: BAD!

Example

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    pthread_create(&p, NULL, mythread, (void *) 100);
    pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

Why threads? → use procedure call

Critical Section

- Locks:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- If another thread holds the lock when calling `pthread_mutex_lock`, the calling thread will not return from the call until the mutex is released.

- All locks must be properly initialized:
  - Using:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

  - Or

```
int rc = pthread_mutex_init(&lock, NULL);
```

    - Returns 0 if everything is ok.
    - Passing `NULL` implies using default attributes.

- There are more interesting routines regarding pthread_mutex:

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

Condition Variables

- Condition variables are used when signaling is needed between threads → one thread is waiting for another to do something before it can continue.

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- To use a condition variable, one has to have a lock that is associated with this condition. When calling one of the routines, this lock should be held.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Condition Variables

Thread 1:

```
pthread_mutex_lock(&lock);
while (ready == 0)
     pthread_cond_wait(&cond, &lock);
pthread_mutex_unlock(&lock)
```

Atomically releases the mutex and waits for the signal.

When the signal arrives, the thread competes for the mutex again
→ Do a pthread_mutex_lock();

Thread 2:

```
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

Do not require the mutex, but it is recommended for proper behavior. Needs to be unlocked right after.

4. Concurrency
4.3 Implementation

Remember...
Critical-Section Problem

- N processes competing to use some shared data.
- Each process has a code segment (critical section) in which the shared data is accessed.
- Many kernel-mode processes can be active in the OS at a time:
  - Subject to race conditions.
  - Preemptive kernel vs. non-Preemptive kernel.

- Solution Requirements:
  - Mutual exclusion: ensure that when one process is executing in its critical section, no other process is allowed to execute its critical section.
  - Progress: if no process is executing in its critical section and there are processes waiting to do so, the selection cannot be postponed indefinitely.
  - Bounded waiting: A limit must exist on the number of times other processes are granted access to the critical section after a process request to enter and before the request is granted.
- Assume that each process executes at non-zero speed.
  - No assumption concerning relative speeds.

Remember...
Heart of the Problem

`counter=counter+1` is not atomic

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0c8049a1c
```

- Uncontrolled Scheduling
  - One thread can enter that region of code to increment counter by one and be interrupted after loading the value to eax but before storing it back.
  - The contents of its register get saved in the TCB (eax included).
  - The new thread loads counter, but gets the unmodified value.
  - When the first thread comes back, it works with the previously loaded value (the one stored in eax).

- Controlling Interrupts
  - Disable Interrupts for critical sections.
  - Easily Abused.
  - Only works on systems with a single CPU.
  - Lost Interrupts.

Remember... Heart of the Problem

`counter=counter+1` is not atomic

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0c8049a1c
```

- Uncontrolled Scheduling
  - One thread can enter that region of code to increment counter by one and be interrupted after loading the value to eax but before storing it back.
  - The contents of its register get saved in the TCB (eax included).
  - The new thread loads counter, but gets the unmodified value.
  - When the first thread comes back, it works with the previously loaded value (the one stored in eax).

- One way to solve the problem: Atomicity → Hardware.
  - To have more powerful instructions that, in one single step, did whatever we need (so an interrupt cannot occur in the middle).
  - Instead of having atomic instructions for everything we need to do in critical sections (costly). Build general set of synchronization primitives.

Locks

- Mutual exclusion:
  - Shared variable lock

```
-- Non critical section --
lock(lock)
-- Critical section –
unlock(lock)
-- Non critical section --
```

Locks: Test-and-Set

- Test and modify the content of a word atomically in a single hardware instruction (cannot schedule in the middle).
  - Initially check for false value, and set to true.
  - Shared variable boolean (lock)

```
void initialize(lock_t lock)
{
    lock=false;
}

void unlock(lock_t lock)
{
    lock=false;
}
```

```
void lock(lock_t lock)
{
   while(test-and-set(lock))
    {; //do-nothing } //
}
```

→ Spinlock

Locks: Swap

- Swap: atomic instruction that exchanges two values.
  - i.e. shared `lock=false,` local `key=true.`

```
void initialize(lock_t lock)
{
    lock=false;
}

void unlock(lock_t lock)
{
    lock=false;
}
```

```
void lock(lock_t lock)
{
   lock_t key=true;
   while(swap(key,lock))
    {; //do-nothing } //
}
```

→ Spinlock

- Compare-and-swap
  - only swap if condition is met.

Locks: LL-SC

- Load Linked (LL) and Store Conditional (SC)
  - LL loads and sets a linked bit value (LLbit).
  - SC performs a store if and only if the LLbit is set. If it is not set, the store fails.
  - The link is broken if:
    - Update to the cache line containing the lock-variable.
    - Invalidate to the cache line containing the lock-variable.
    - Upon returning from an exception.

Load-Linked & Store-Conditional

```
void lock(lock_t *lock)
{
  while(1)
    {
       while(loadLinked(lock->flag)==1)
       {; //do-nothing }
       if(storeConditional(lock->flag, 1)==1)
       { return; }
       //otherwise try it all over again.
    }
  }
}

--------------------

void unlock()
{
  lock->flag = 0;
}
```

Critical Section

- Spinlocks
  - Avoid Context switching
  - Poor performance: with few CPUs, it uses the CPU, thus not allowing the holding thread to make progress → Scheduler is ignorant!
  - Unfair: A process may spin forever leading to starvation.

Ok with many CPUs

- Alternative: blocking locks/Queue Locks
  - Suspend the thread on a wait queue until lock is released.
  - Do not waste CPU cycles.
  - More context switches.

Better if CPU limitations

- Hybrid approach: spin a while, then queue self.
  - Two-phase locks

Parallel Programming

- Coarse-Grained Locking
  - ✓ Easy to program.
  - ✗ Scales poorly.
- Fine-Grained Locking
  - ✓ Scales well, good performance.
  - ✗ Hard to get it right.

SO Mechanism: Semaphores

- More sophisticated than mutex lock. Reduce the busy wait.
- A semaphore (S) is an integer variable that can be accessed by two atomic operations: `wait()` and `signal()`.

```
wait(S) {
  while(S<=0)
     /* spinlock? */
  S--;
}
signal(S) {
  S++;
}
```

More details later

- Modifications must be executed indivisibly. If one process modifies a semaphore, no other process can simultaneously modify the same semaphore.
  - Mono-processor: Inhibit interrupts.
  - Multi-processor: Hardware support or software solution (critical section now the `wait()` and `signal()` code).

Example: Critical Section

- Shared variables:

```
semaphore_t synch;
synch = 1;
```

- Process P_i:

```
while(condition)
{
    wait(synch);
        --- critical section ---
    signal(synch);
        --- remainder section ---
}
```

Semaphore Implementation

- Define semaphore as a struct:

```
typedeff struct {
    int value;
    struct process *list;
} semaphore;
```

- Assume two operations:
  - `block()` (suspend the process that invokes it)
  - `wakeup(P)` resume the execution of a blocked process P.

- Semaphores have history:
  - The counter (when it reaches 0, the semaphore is closed).
    - wait() decrements counter, signal() increments it.
  - Blocked processes are queued on a list.

Semaphore Implementation

- Semaphore operations: (initially `S->value=1`)

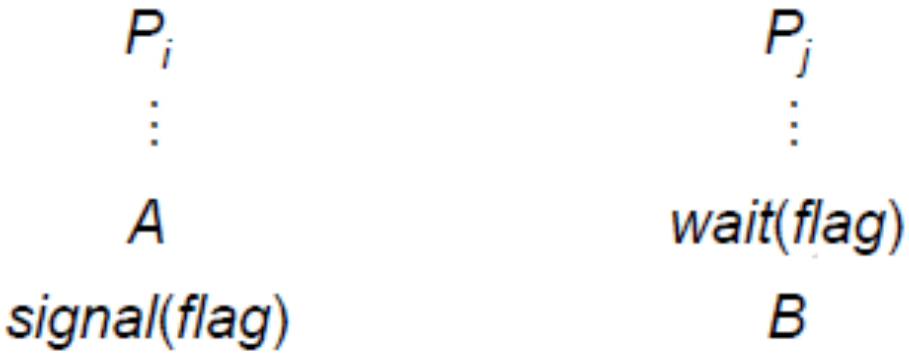
```
wait(semaphore *S) {
     S->value--;
     if(S->value < 0) {
             add this process to S->list;
             block;
     }
}
```

Number of waiting threads.

```
signal(semaphore *S) {
     S->value++;
     if(S->value <= 0) {
             remove a process P from S->list;
             wakeup(P);
}
```

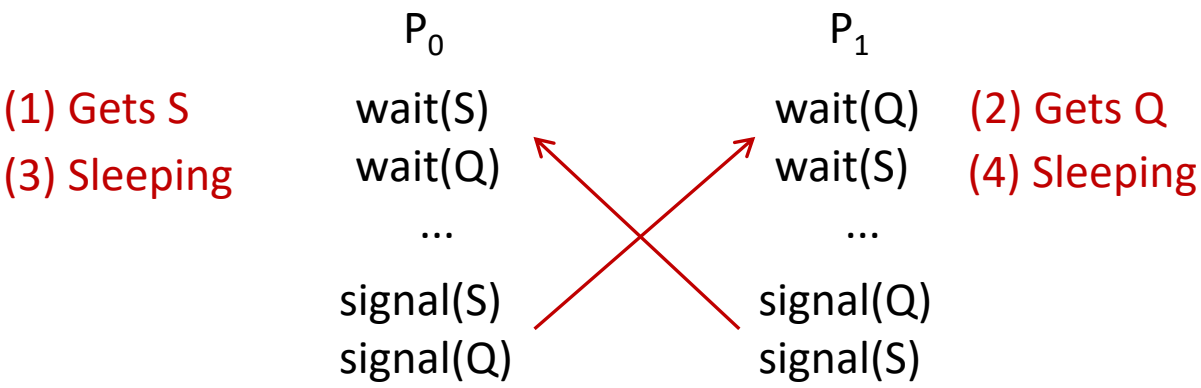
Semaphore as a Synch. Tool

- Execute B in P_j only after A in P_i.
- Semaphore flag initialized to 0.



Deadlock and Starvation

- Deadlock: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1, and suppose the sequence:



- Starvation: indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
  - May occur if we removed processes from the list in LIFO (last-in, first-out).

Priority Inversion

- Three processes: L, M and H.
  - Priority L<M<H.
  - Both L and H needs to read or modify kernel protected resource R (with a lock).
  - Preemptive scheduler.
- If L is already accessing R, H is waiting for R, and then M enters the ready state.
  - M becomes runnable, preempting L.
  - M has now affected how long process H has to wait
  - This problem is known as Priority Inversion.
- E.g. Mars Pathfinder realtime system.
- Solutions:
  - Just two priorities → Insufficient in most general-purpose operating Systems.
  - Priority-inheritance: All processes that are accessing a resource needed by a higher priority process, inherit the higher priority until they are finished with the resource.

Classic Synchronization Problems

- Used for testing nearly every new proposed synchronization scheme.
- Bounded-Buffer Problem
  - There is a buffer in memory with finite size.
  - Simplest case: single producer, single consumer.
  - Used to illustrate the power of synchronization primitives.
- Readers-Writers Problem
  - A data object is to be shared among several concurrent processes.
  - Some read, some write.
- Dinning-Philosophers Problem
  - A classical synchronization problem for a large class of concurrency-control problems.

Bounded-Buffer Problem

- Also known as the producer-consumer problem.
  - Make sure that the producer doesn't try to add data into the buffer if it's full and that the consumer doesn't try to remove data from an empty buffer.

```
semaphore full = BUFFER_SIZE; // remaining space
semaphore empty = 0; // items produced
semaphore mutex = 1; // lock to access shared resource
```

```
producer()
{
    while (true) {
        item = produceItem();
        wait(full);
        wait(mutex);
        putItemIntoBuffer(item);
        signal(mutex);
        signal(empty);
    }
}
```

```
consumer()
{
    while (true) {
        wait(empty);
        wait(mutex);
        item = getItemFromBuffer();
        signal(mutex);
        signal(full);
        consumeItem(item);
    }
}
```

Readers-Writers Problem

- If multiple readers are permitted, then there are further issues about when writer can access the object.
- We must wait for all readers to complete reading before issuing the lock to the writer.
  - We can let no new readers start
  - Or make the writer wait until there are no readers waiting.
- Both solutions may result in starvation.

```
semaphore mutex= 1; // readers waiting
semaphore rw_mutex = 1; // readers/writers mutex
semaphore read_count = 0; // "number" of readers
```

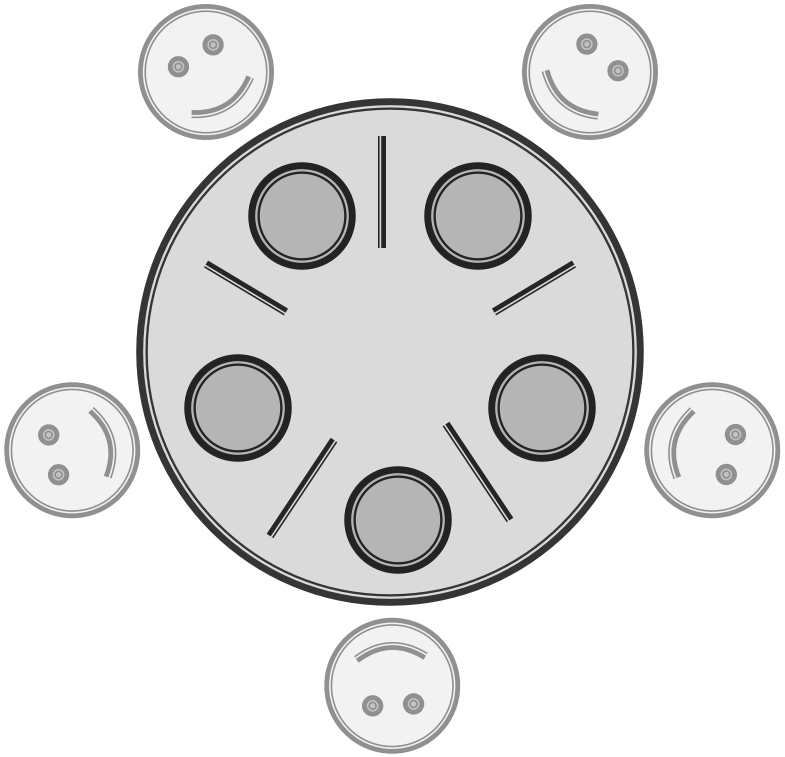
```
do {
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    signal(rw_mutex);
} while(true);
```

Next ones sleep here

If there is a writer, first reader sleeps here

```
do {
    wait(mutex);
    read_count++;
    if (read_count==1)
        wait(rw_mutex);
    signal(mutex);
    /* reading is performed */
    wait(mutex);
    read_count--;
    if(read_count==0)
        signal(rw_mutex);
    signal(mutex)
} while(true);
```

Dining-Philosophers Problem



- Shared data:

```
semaphore chopstick[5]; // initially 1 //
```

Dinning-Philosophers Problem

- Philosopher i :

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1)%5]
   ...
   /* eat for a while */
   ...
   signal(chopstick[i]);
   signal(chopstick[(i+1)%5]);
   ...
   /* think for a while */
   ...
} while(true);
```

- This is not a valid solution because of the possibility of deadlock if the five philosophers became hungry simultaneously.
  - Pick up the chopsticks only if both are available (this is a critical section).
  - Asymmetric solution: even philosophers pick the right chopstick first, odd philosophers pick up the left one first.

Semaphores Problem

- Semaphores are very susceptible to programming errors:
  - The reversal of a wait() and signal() will cause mutual exclusion to be violated.
  - The omission of either wait() or signal() will potentially cause deadlock.

- These problems are difficult to debug and potentially difficult to reproduce.

- Higher-Level Synchronization:
  - There are a number of higher level programming language constructs that exist to cope with this:
    - Condition Variables
    - Monitors
    - Critical Regions
  - These constructs lead not only to safer but also neater and more understandable code.

Condition Variables

- Synchronization structures associated with a mutex that can block a process until an event (condition) happens.
  - Allows ordering (e.g. A runs after B).
  - Associated with a mutex
    - Should be executed between a lock and unlock.
  - queue of sleeping threads
    - Threads add themselves to the queue with wait.
    - Threads wake up threads on the queue with signal.

Condition Variables

- Wait:
  - Assumes the lock is held when wait() is called.
  - Puts caller to sleep + releases the lock (atomically).
  - When awoken, reacquires lock before returning.
- Signal:
  - Releases a suspended process on the condition variable.
  - The waking up process competes again for the mutex.
    - Mesa semantics: there is no guarantee that when the woken up thread runs, the state will still be as desired.
- Broadcast:
  - Wakes up all waiting threads.
  - Guarantees that any threads that should be awoken, are.
  - Negative performance impact
    - Can wake up waiting threads that it shouldn't.
    - Many threads will compete, re-check condition and go back to sleep.
    - If possible, use multiple condition variables.

Condition Variables

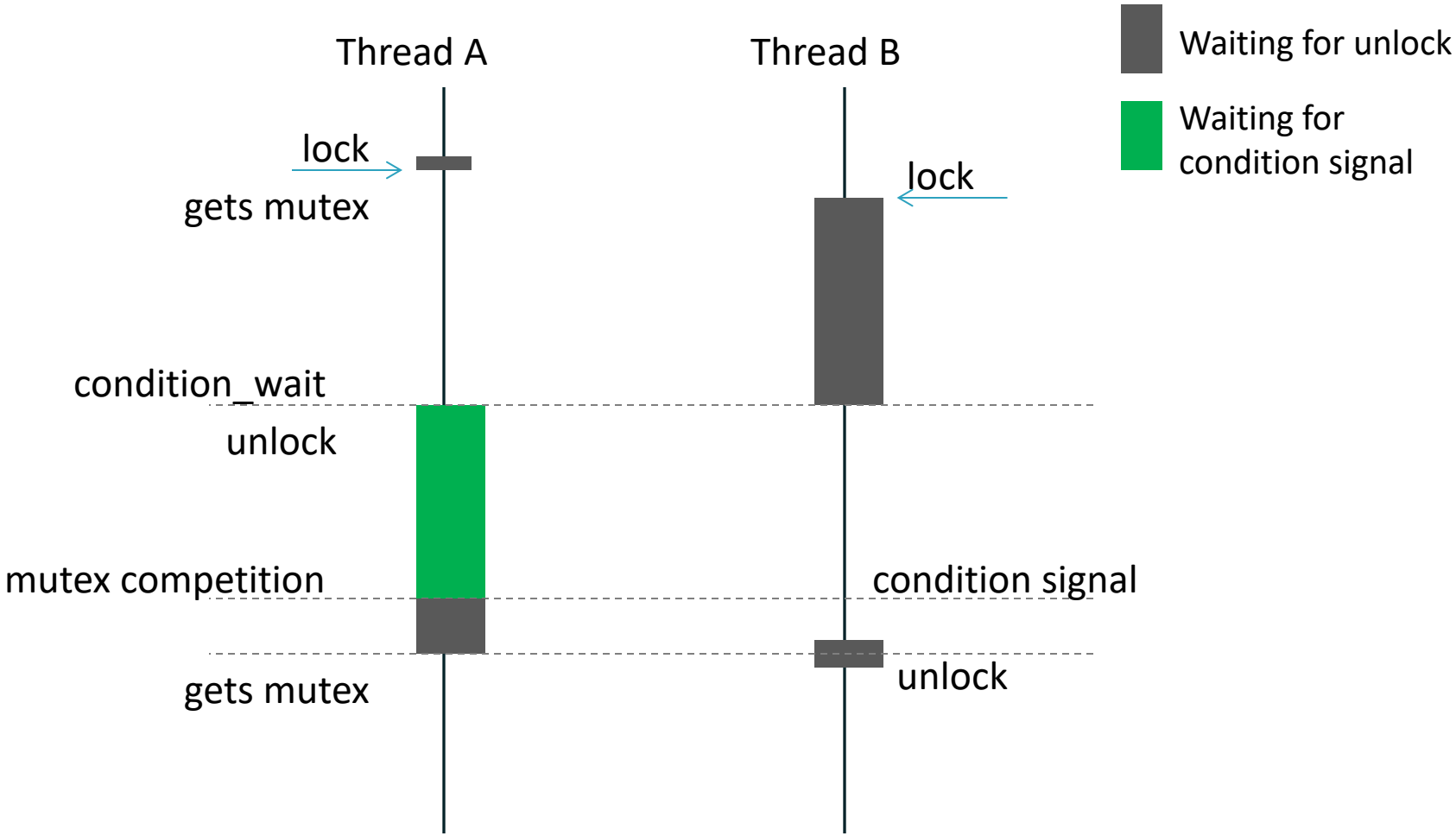
- Always do wait and signal while holding the lock!

Strictly required

Good practice, but in some libraries it is not required

Doing so will help prevent lost signals.

Condition Variables



Condition Variables usage

- Thread A:

```
lock(mutex); /* accessing resource */
/* check the shared resources */
while(occupied) {
  wait(condition, mutex);
}
/* use resource */
unlock(mutex);
```

- Thread B:

```
lock(mutex); /* accessing resource */
/* use resource */
occupied = false;
signal(condition, mutex); /* free resource */
unlock(mutex);
```

Producer-Consumer, again...

```
producer()
{
   int pos = 0;
   while (true) {
      produceItem();
      lock(mutex);
      /* access buffer */
      if(n_elements==BUFFER_SIZE)
         wait(condition, mutex);

      buffer[pos]=dato;
      pos = (pos+1)%BUFFER_SIZE;
      n_elementos++;
      if(n_elementos == 1)
         signal(condition);

      unlock(mutex);
   }
}
```

```
consumer()
{
   while (true) {
      lock(mutex);
      if(n_elementos == 0)
         wait(condition, mutex);

      dato=buffer[pos];
      pos=(pos+1)%BUFFER_SIZE;

      n_elementos--;
      if(n_elementos==BUFFER_SIZE-1)
         signal(condition);

      unlock(mutex);
      consumeItem(item);
   }
}
```

If there is more than one producer/consumer, there is a race.

Mesa semantics signal interpretation vs. Hoare semantic.

Producer-Consumer, again...

```
producer()
{
   int pos = 0;
   while (true) {
      produceItem();
      lock(mutex);
      /* access buffer */
      while(n_elements==BUFFER_SIZE)
         wait(condition, mutex);

      buffer[pos]=dato;
      pos = (pos+1)%BUFFER_SIZE;
      n_elementos++;
      if(n_elementos == 1)
         signal(condition);

      unlock(mutex);
   }
}
```

```
consumer()
{
   while (true) {
      lock(mutex);
      while(n_elementos == 0)
         wait(condition, mutex);

      dato=buffer[pos];
      pos=(pos+1)%BUFFER_SIZE;

      n_elementos--;
      if(n_elementos==BUFFER_SIZE-1)
         signal(condition);

      unlock(mutex);
      consumeItem(item);
   }
}
```

If there is more than one producer/consumer → deadlock.
producers just wake consumers and vice-versa.

Producer-Consumer, again...

```
producer()
{
   int pos = 0;
   while (true) {
      produceItem();
      lock(mutex);
      /* access buffer */
      while(n_elements==BUFFER_SIZE)
         wait(c_full, mutex);

      buffer[pos]=dato;
      pos = (pos+1)%BUFFER_SIZE;
      n_elementos++;
      if(n_elementos == 1)
         signal(c_vacio);

      unlock(mutex);
   }
}
```

```
consumer()
{
   while (true) {
      lock(mutex);
      while(n_elementos == 0)
         wait(c_vacio, mutex);

      dato=buffer[pos];
      pos=(pos+1)%BUFFER_SIZE;

      n_elementos--;
      if(n_elementos==BUFFER_SIZE-1)
         signal(c_full);

      unlock(mutex);
      consumeItem(item);
   }
}
```

Semaphores, Locks and CVs

- Semaphores are a powerful and flexible primitive.
  - Semaphores contain state.
  - Can be used for both, mutual exclusion and ordering.
- Semaphores are equivalent to Locks + Condition Variables.
  - It is possible to build semaphores with locks and condition variables.
  - It is possible to build locks and condition variables with semaphores.
- Sometimes semaphores flexibility is not needed.
  - Building Condition Variables with Semaphores is not easy.
  - Semaphores flexibility sometimes means more susceptibility to programming errors.
  - Locks or Condition Variables may be more convenient.

Summary: Concurrency Keys

- Critical Section: A piece of code that accesses a shared resource.
- Race Condition: Arises if multiple threads enter the critical section at roughly the same time, leading to an undesired/indeterminate outcome.
- Non-determinism: A program with one or more race conditions may vary the output from run to run. The outcome is indeterminate, something we usually expect from a computer system.
- Mutual exclusion: Guarantees that only a single thread enters a critical section, avoiding races and providing deterministic program outputs.
- Ordering: In concurrent programs sometimes one thread needs to wait for another to complete an action before it continues.