Synchronization and communication

ARCOS Group

Distributed Systems

Bachelor In Informatics Engineering

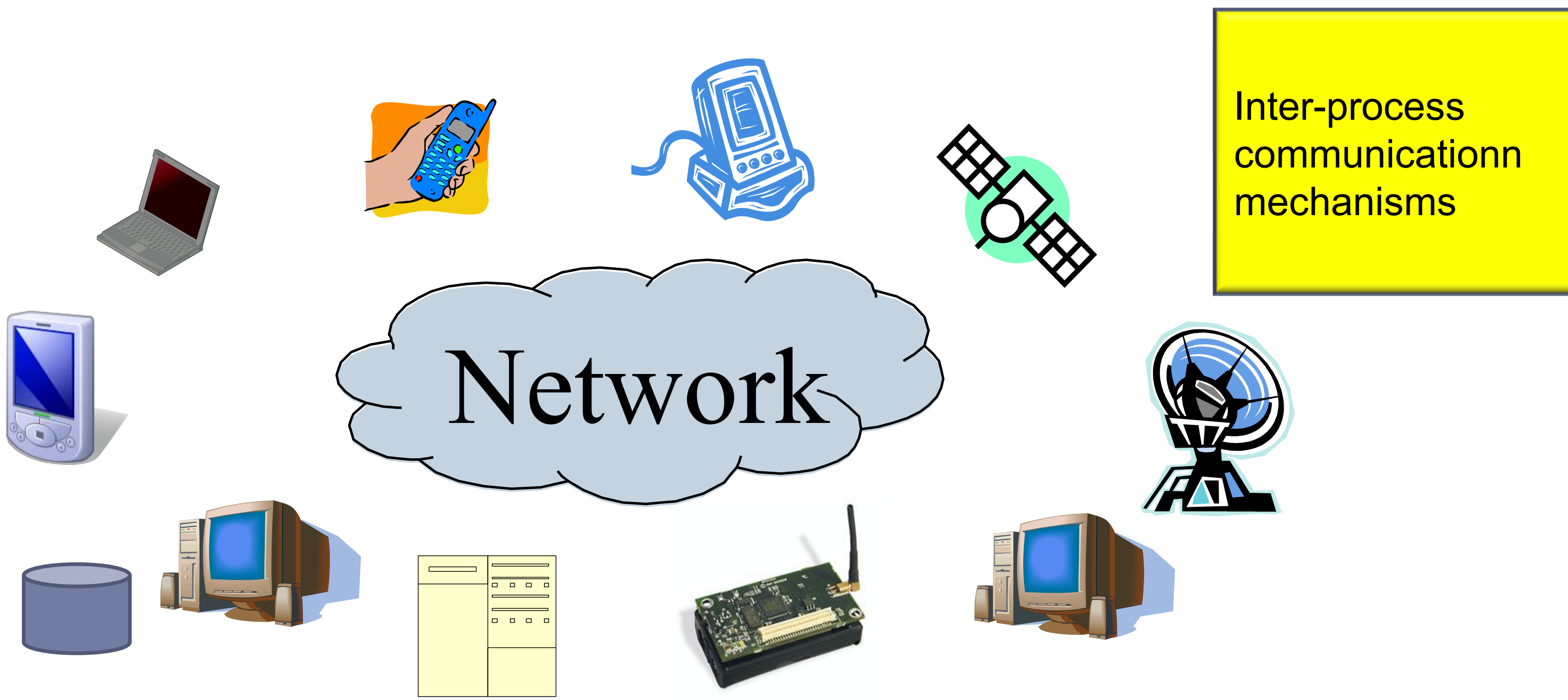
Universidad Carlos III de Madrid

Syllabus

- Processes and threads
- Concurrency
- Synchronization mechanisms
- Communication mechanisms
- POSIX

Distributed system

- Consists of physically distributed resources (hardware and software) which are connected via a network and are able to collaborate to accomplish a given task



Monothread vs. Multithread processes

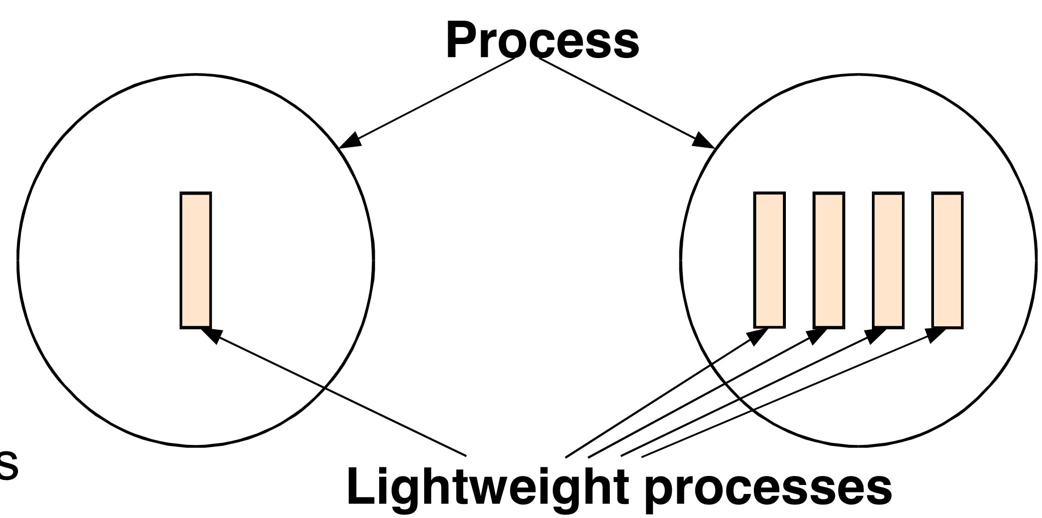
- A lightweight process / thread is an execution flow which shares the memory image (and other information) with other lightweight processes
  - Depending on the number of processes that can execute simultaneously an OS is unitask or multitask

- A process may consist of a single execution flow (aka monothread) or more (aka multithread)

Private and shared information

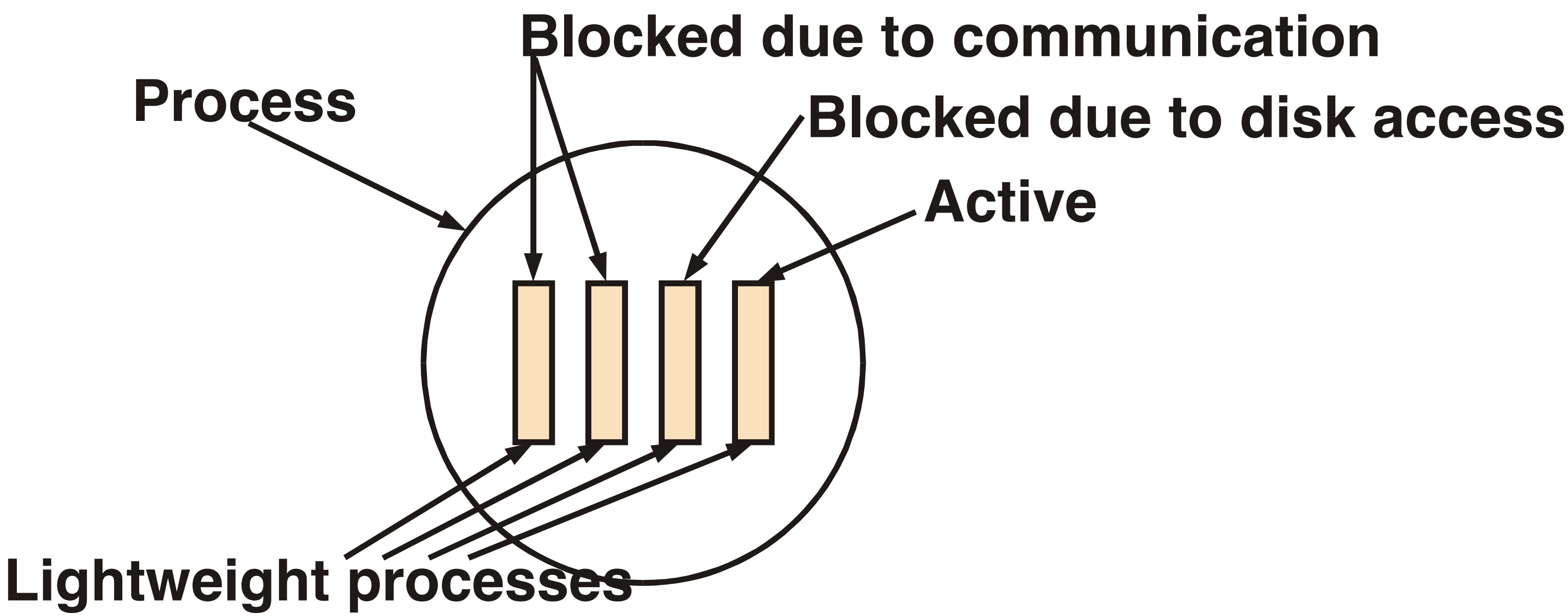
- Per *thread* (private)
  - PC, registers
  - Stack
  - Status (executing, ready, blocked)

- Per process (shared)
  - Space in memory
  - Global variables
  - Open files
  - Child processes
  - Timers
  - Signals and semaphors
  - Statistics



The status of a lightweight process

- Just like a conventional process, a lightweight process may be in the following states: executing, ready, or blocked (or suspended for medium-term scheduling)
  - The status of a process is the combination between the statae of the lightweight processes that it consists of

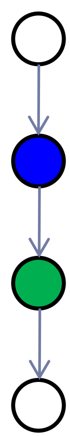


Threads and concurrency

- Lighweight processes enable easy app. paralelization
  - A program may be split into parts which may execute concurrently within lightweight threads
  - Encourages modular application design
- The basic idea is to always keep some lightweight process executing – while a thread is blocked another one may execute

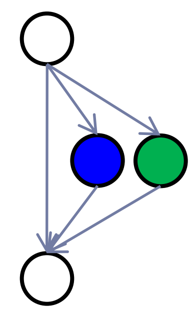
```
main(){
…
function1(args);
function2(args);
…

}
```



Sequential design

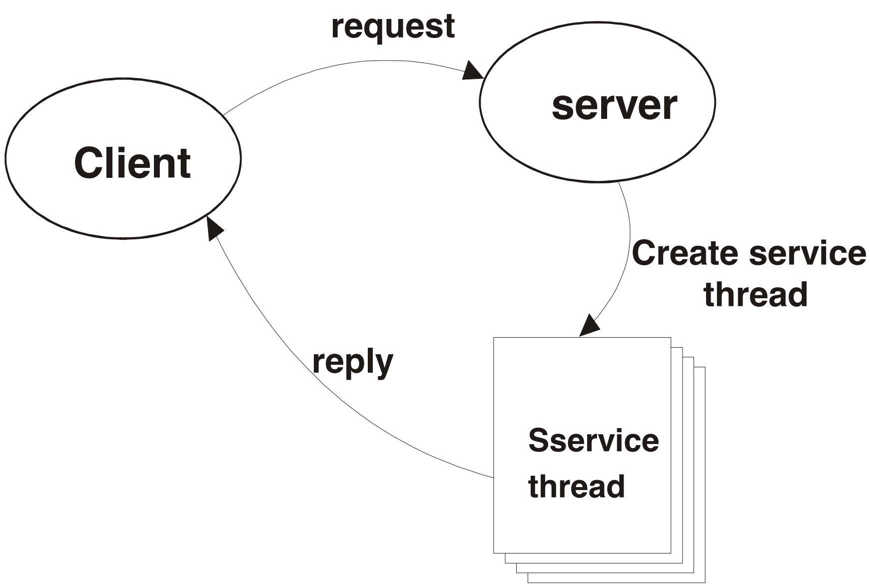
```
main(){
…
run_thread    (
   function1(args));
run_thread    (
   function2(args));
}
```



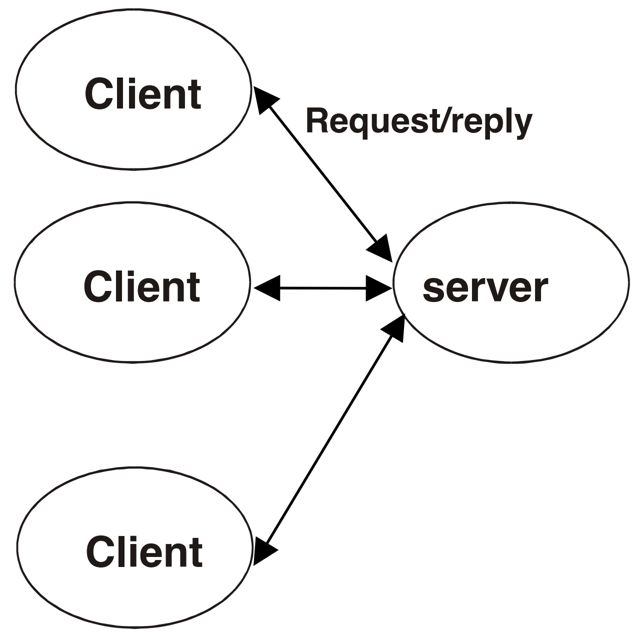
Parallel design

E.g.: concurrent servers with *threads*

- The concurrent processes must communicate and synchronize



Concurrent design

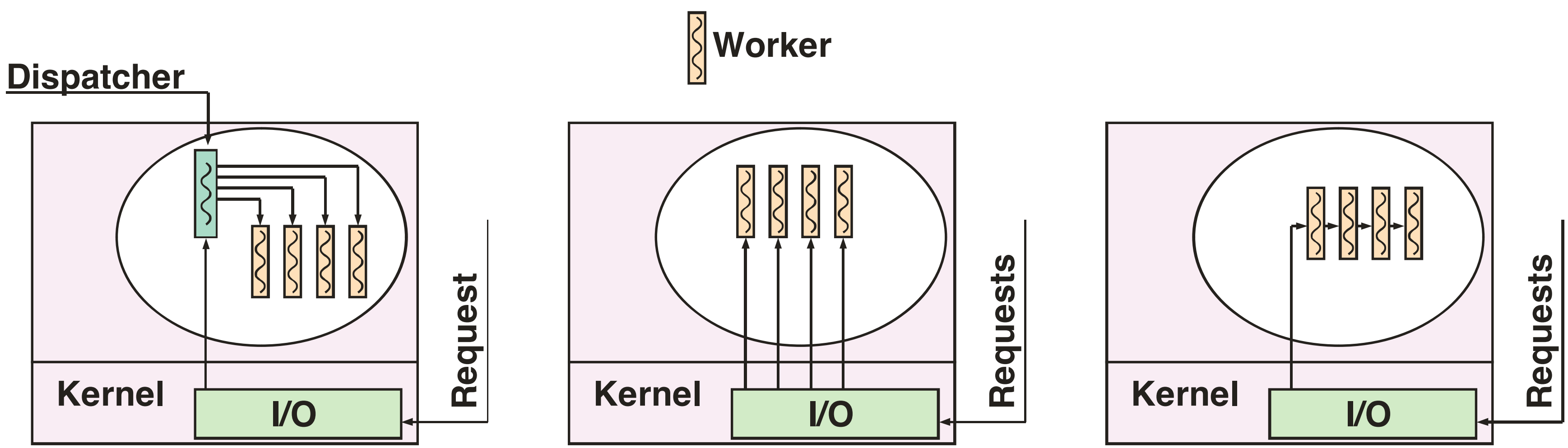


Sequential design

Server design using *threads*

Several alternatives to build parallel servers:

- A single process which accepts requests and either (1) distributes them to threads from a thread pool or (2) creates a new thread to service the request
- Set of similar threads which can read requests from a port
- Pipeline the work and have a specialized thread for each stage



Programming using threads

- UNIX offers the following thread management operations:
  - Create and identify threads
  - Associate attributes with threads
  - Schedule threads
    - Priorities, scheduling policies
  - Terminate threads
    - Does NOT imply the process' children must terminate immediately
    - Parent process may wait for children to finish – wait
    - Process may wait for another process to finish - join

Create and identify *threads*

- Creation:

```
int pthread_create(pthread_t* thread,
     const pthread_attr_t* attr, void * (*function)(void *),
     void * arg);
```

where:

thread	new thread id
attr	attributes of the new thread (or NULL)
function	the function that the new thread will execute
arg	pointer to the thread arguments

returns:

int	an integer if successful creation; -1 otherwise

- Getting the thread id:

```
pthread_t pthread_self(void)
```

Thread attributes

- Default values
  - **E.g.**: a thread is *joinable* (not independent) if not defined otherwise

- To modify the thread attributes:
  - Initialize an attribute object to use it when creating new threads

    ```
    int pthread_attr_init(pthread_attr_t *attr);
    ```

  - Destruct an attribute

    ```
    int pthread_attr_destroy(pthread_attr_t *attr);
    ```

  - Change termination status (Joinable / Detached)

    ```
    int pthread_attr_setdetachstate(
                        pthread_attr_t *a, nt detachstate);
    ```

    If `detachstate` is
    - PTHREAD_CREATE_DETACHED the process is independent
    - PTHREAD_CREATE_JOINABLE the process is NOT independent
  - Other attributes: stack size, scheduling policy, priority, etc

Terminating a thread

- Finalize the thread execution

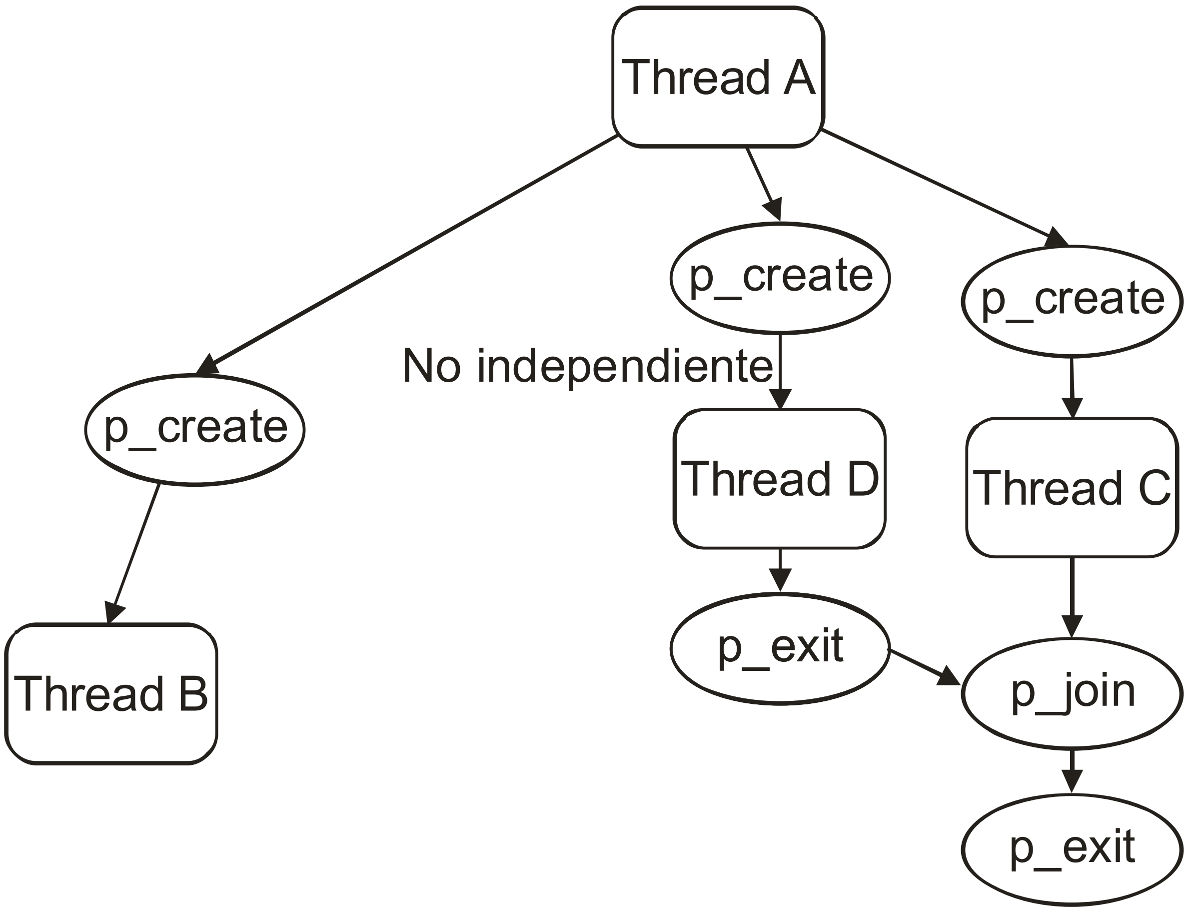
```
int pthread_exit(void *value);
```

- If a process is not independent (joinable) then need to call

```
int pthread_join(pthread_ thid, void **value);
```

  - Wait for process with id *thid* to finish
  - Return in second argument the value passed in *pthread_exit.*
  - Can't wait for termination of independent threads

E.g.: Thread hierarchy



E.g. I: Creation and termination

```
#include <stdio.h>    /* printf */
#include <pthread.h> /* For threads */
#define NUM_THREADS 10

int function(int *idThread){
        printf("Thread id = %d\ti=%d: ", pthread_self(),*idThread);
        pthread_exit(NULL);
}
int main () {
        pthread_t arrayThread[NUM_THREADS]; /* Array of threads */
        int i;
        /* CREATE THREADS */
        for(i=0;i<NUM_THREADS;i++)
        if (pthread_create(&arrayThread[i], NULL, (void *)funcion, &i)==-1)
                printf("Error\n");
        /* WAIT FOR TERMINATION */
        for(i=0;i<NUM_THREADS;i++)
                pthread_join (arrayThread[i], NULL);
        exit(0);
}
```

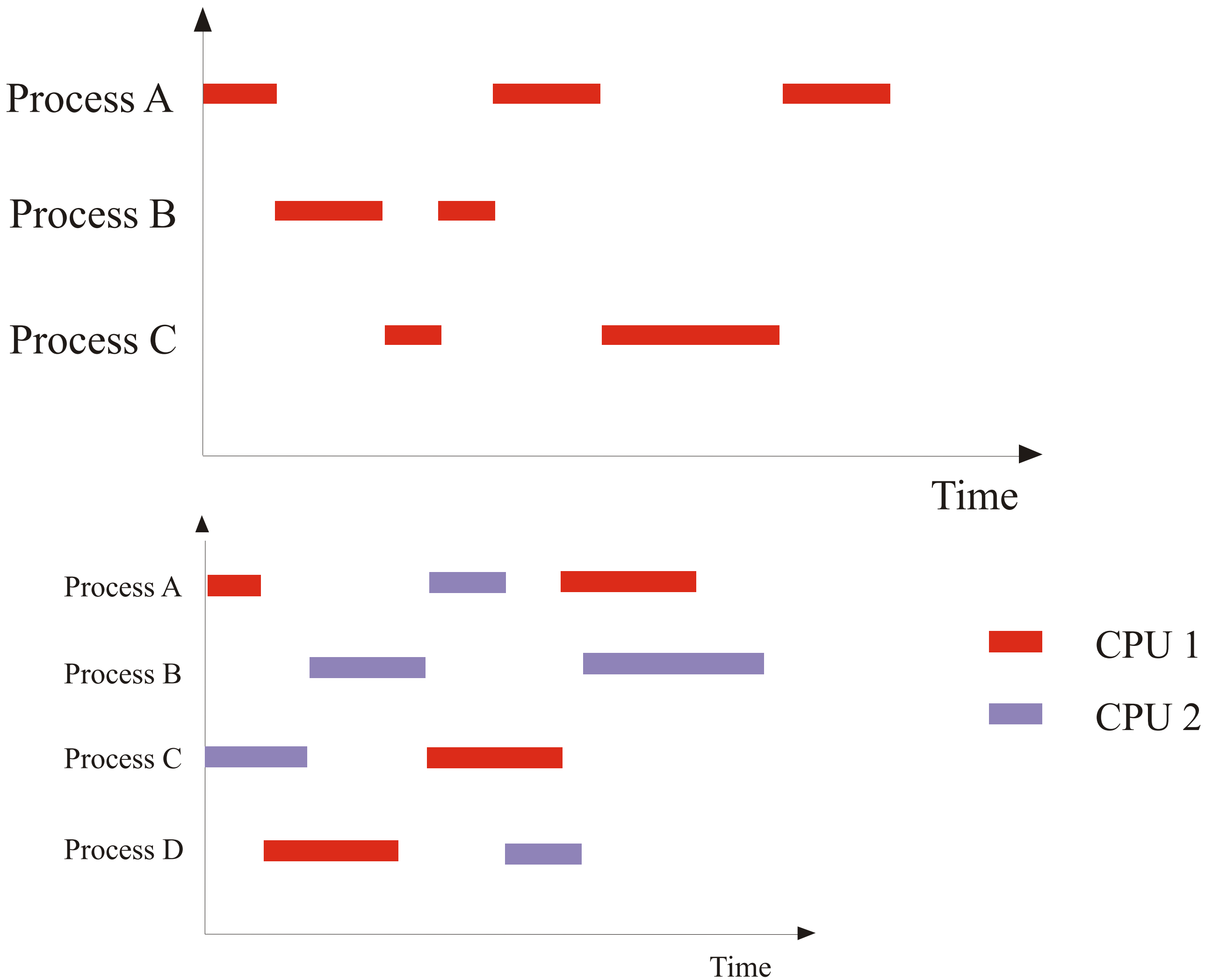
E.g. II: Modifying attributes

```
#include <stdio.h>    /* printf */
#include <pthread.h> /* For threads… */
#define MAX_THREADS 10
void func(void) {
  printf("Thread %d \n", pthread_self());
  pthread_exit(0);
}
main() {
  int j;
  pthread_attr_t attr;
  pthread_t thid[MAX_THREADS];
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
  for(j = 0; j < MAX_THREADS; j ++)
      pthread_create(&thid[j], &attr,  func, NULL);
  sleep(5);
  pthread_attr_destroy(&attr);
}
```

Concurrency

- Models:
  - Multiprogramming using one processor
  - Multiprocessor (shared memory system)
  - Multicomputer (distributed system)
- Advantages:
  - Faster processing
  - Multiuser systems w interaction
  - Better use of resources

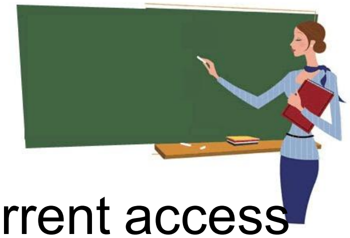
Monoprocessor vs. Multiprocessor



Types of concurrent processes

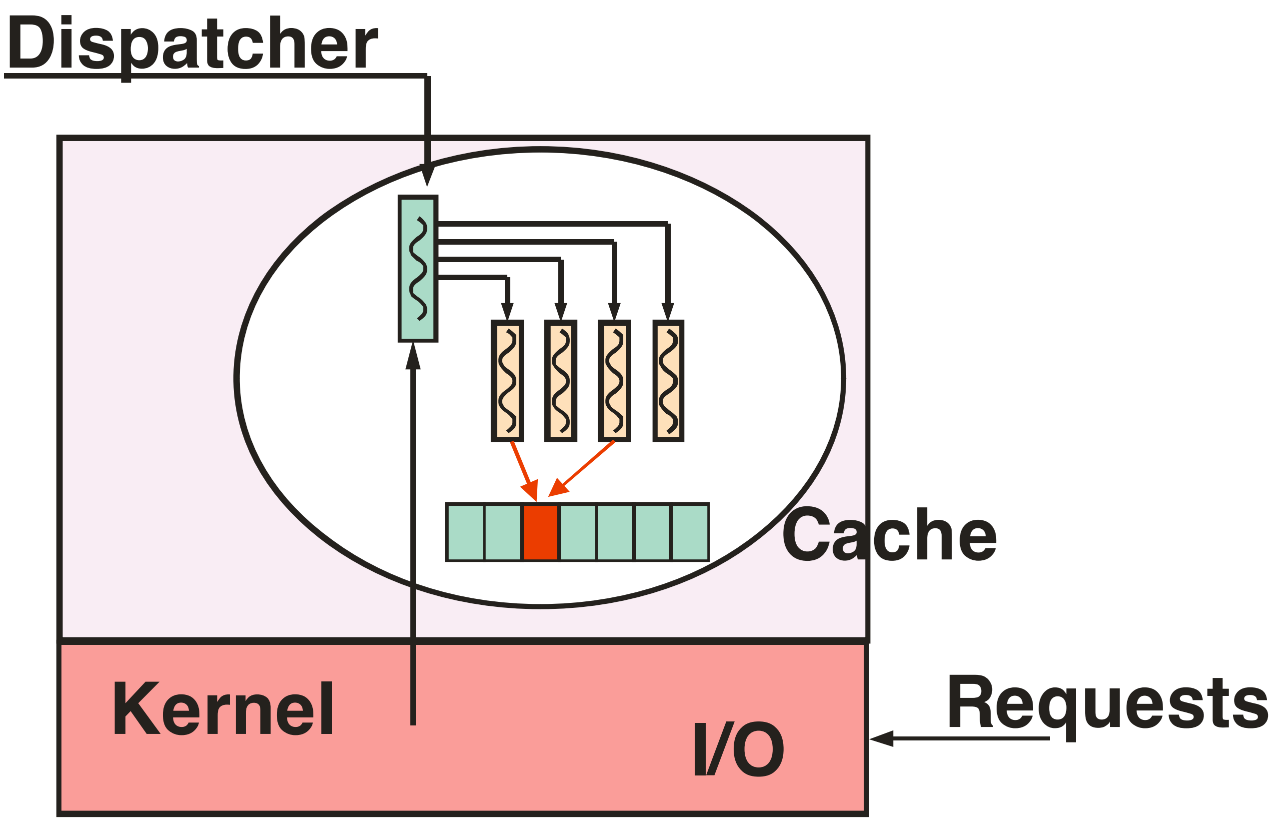
- Independent: successful execution does not depend on collaborating with other processes
- Cooperanting: collaboration is necessary to perform their function
- In both cases you have process interaction
  - they compete for resources
  - they share resources

Recap

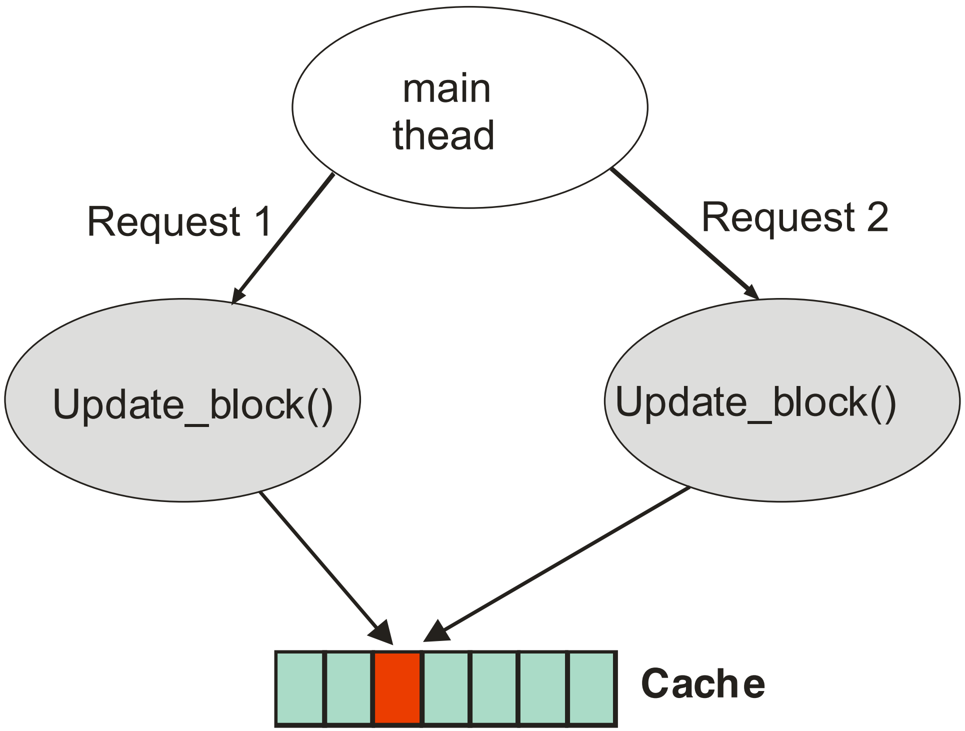


- A race condition involves unsafe (uncoordinated) concurrent access to shared resources
- A critical section (CS) is a code fragment via which various processes have access to shared resources
  - They may only execute on the processor on which they are entered
  - Usually enforced via semaphores
- An atomic section is an indivisible and uninterruptible code fragment
  - Guarantees that the invariants are observed and preserved by all operations
  - Often has a succeed-or-fail semantics
  - Usually enforced via mutual exclusion; may also be enforced via lock-free mechanisms, e.g. transactional memory

E.g.: Concurrent server



E.g.: Accessing a shared cache

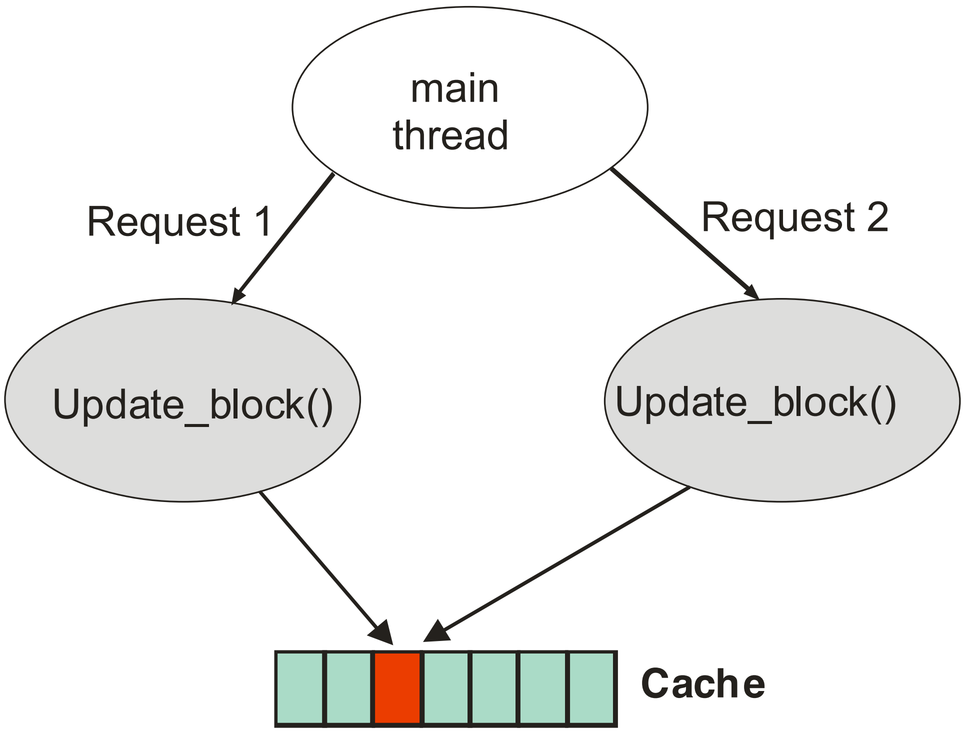


Race condition: `Update_block()`

```
Update_block(block) {
      b = Get_block();
      copy(b, block);
}

copy(b1, b2){
      for (i=0; i<length;i++)
        *b1++ = *b2++;
```

Accessing a shared cache



Race condition: `Update_block()`

```
Update_block(block) {
      b = Get_block();
      copy(b, block);
}

copy(b1, b2){
      for (i=0; i<length;i++)
        *b1++ = *b2++;
```

Non atomic operation

Critical section

- *n* processes
  - Each one of them has a code fragment: critical section
  - Can't have more than 1 process at a time in a critical section

- To solve the critical section problem you must:

```
Enter CS
CS code
Exit CS
```

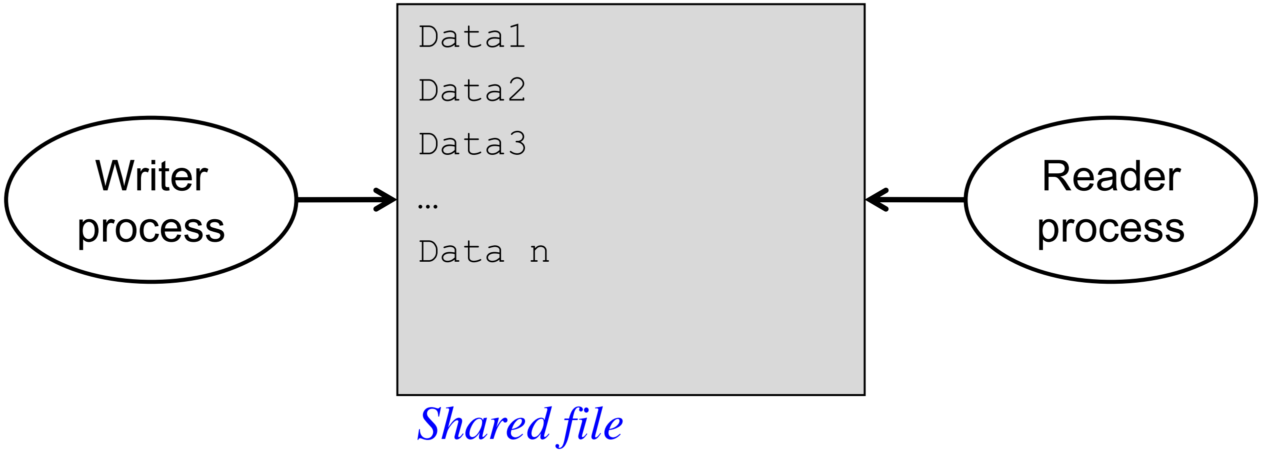
- Any mechanism solving the problem must ensure:
  - Mutual exclusion
  - Progress
  - Bounded wait (no starvation)

Communication mechanisms

- Files
- *Pipes* / FIFOS
- Shared memory variables
- Message passing

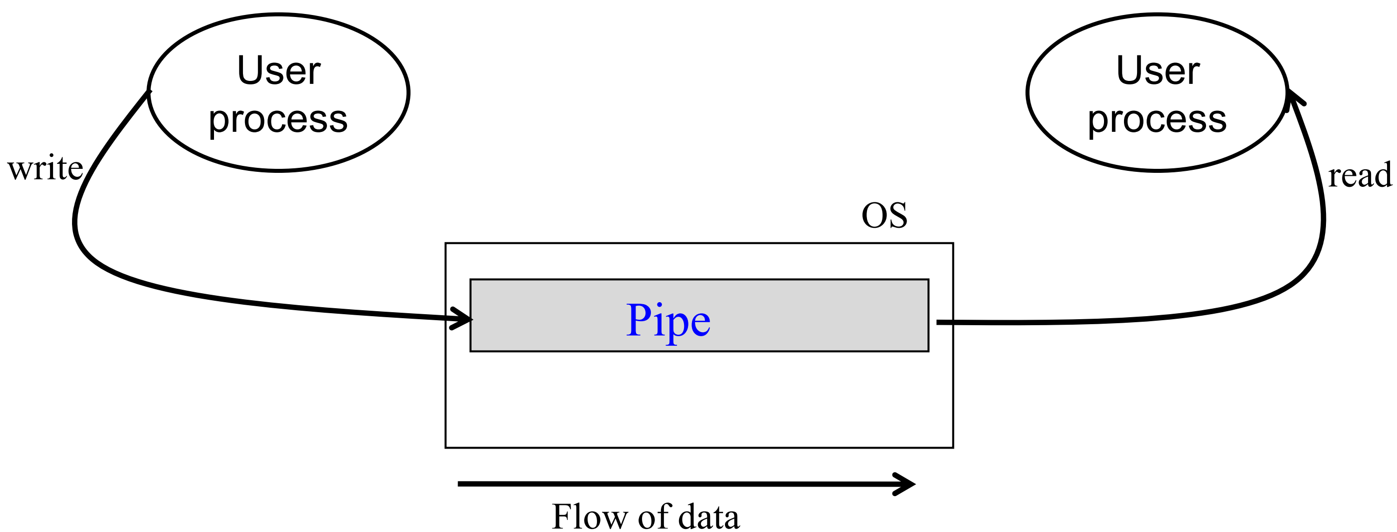
Files

- May be used for communicating between processes
  - Some write others read



Pipes

- A pipe is a communication and synchronization mechanism
  - Communication: shared file
  - Synchronization: due to the blocking semantics
- Aka a pseudo-archive identified by two descriptors: one for R, one for W
  - Same mechanism for R and W as for files
  - Atomic R and W



Synchronization mechanisms

- Signals (asynchronous communication)
- Pipes / FIFOS
- Semaphores
- Mutex and condition variables
- Message passing
- Monitors

Semaphores

- A semaphore [Dijkstra, 1965] is a synchronization mechanism for processes which execute on the same machine
- A protected variable which providea an abstraction for controlling access to a single resource
  - Keeps thack of the number of resources available at the time
  - Provides operations to adjust the number of resources in a race-free manner (and wait if no resource is available)
- Processes are trusted to follow the protocol. If not problems may occur!
- Even if they do follow the protocol, multi-resource deadlock may occur if different semaphore / resource

Semaphores (cont'd)

- Two atomic operations:
  - `P – aka wait`
  - `V – aka signal`
- Atomicity achieved via atomic primitive HW instruction or mutual exclusion algorithm
  - On uniprocessors: disable interrupts is enough

Operations on semaphores

- P wait
- V signal

```
wait(s)
{
   s = s - 1;
   if (s < 0) {
      <block the process>
   }
}
```

```
signal(s)
 {
     s = s + 1;
     if (s <= 0) {
        <unblock a process
         blocked on wait>
     }
}
```

s: integer value associated with the semaphore

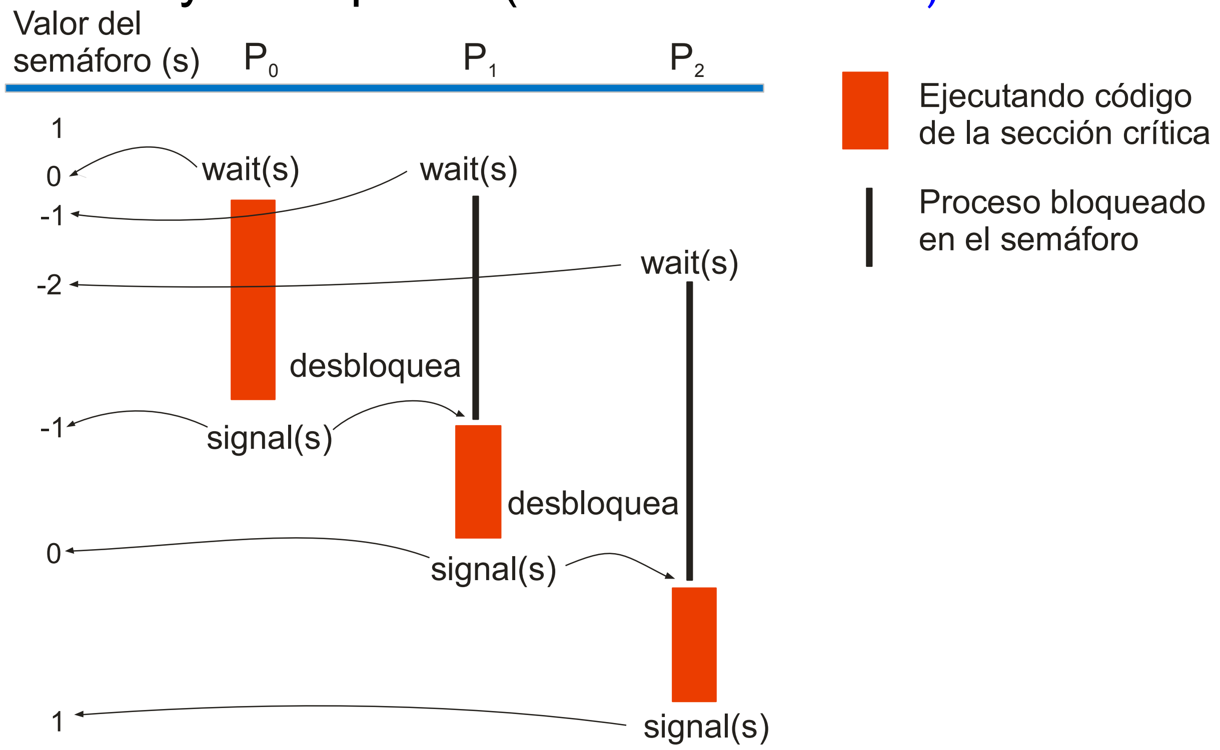
Use of semaphores to protect the critical section :

```
wait(s)
Critical section
signal(s)
```

Critical sections with semaphores

```
wait(s); /* enter critical section */
< critical section >
signal(s);  /* exit critical section */
```

Use a binary semaphore (with initial value 1)

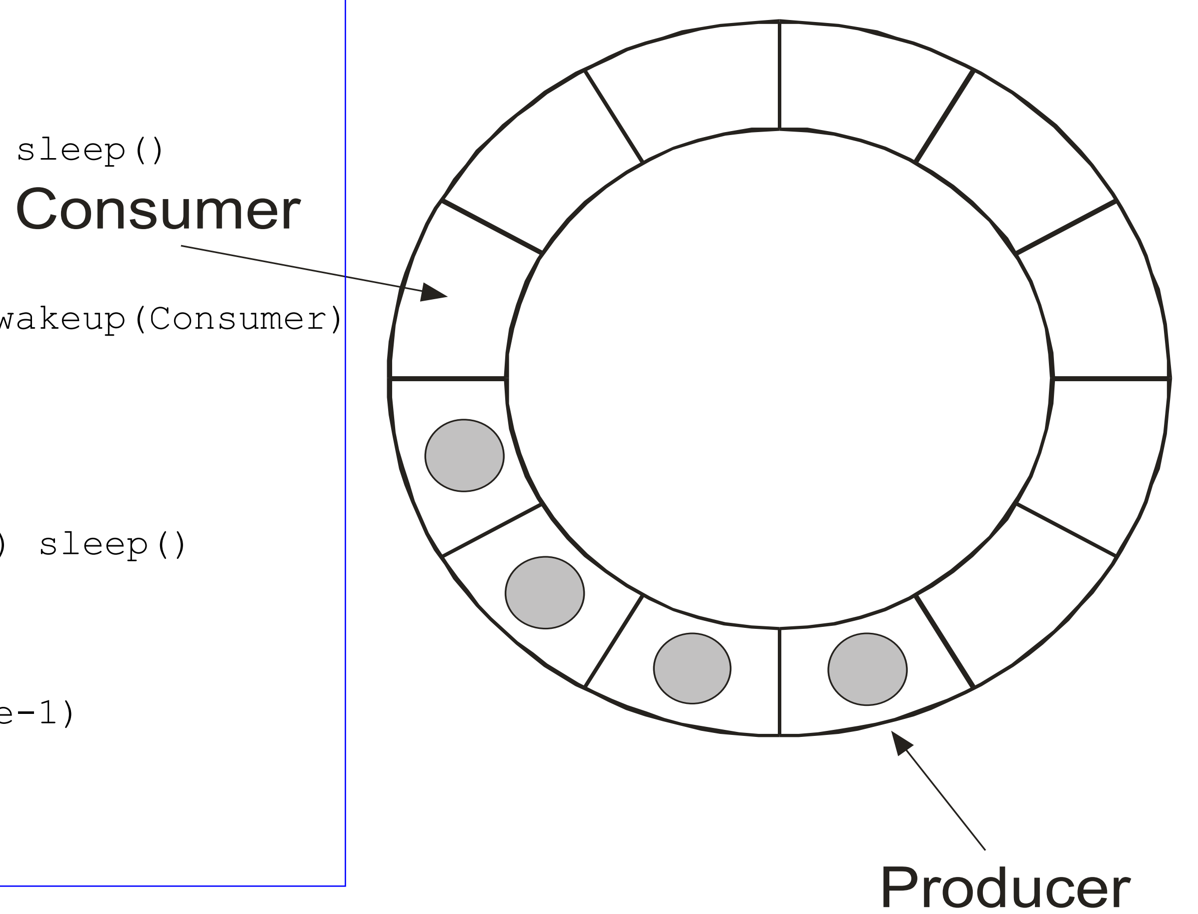


Producer-consumer

Use sleep and wakeup routines

Sleep blocks the caller until another process wakes it up

```
Producer() {
   while (1) {
        Produce item
        if buffer_full() sleep()
        insert item
        counter++
        if (counter==1) wakeup(Consumer)
        }}
Consumer() {
   while (1) {
        if buffer_empty() sleep()
        remove item
        counter—
        if (counter==size-1)
   wakeup(Producer)
        consume item
   }}
```



Producer-consumer

- Race condition!
  - Consumer interrupted before calling sleep()
  - Producer wakes up consumer, BUT
    - Consumer not yet sleeping so wakeup call is lost
  - Consumer never awakened again (counter won't be 1 again!)

Producer-consumer with semaphores

```
Producer() {
   while (1) {
         Produce item
         wait(emptyCount)
         insert item
         signal(fillCount)
   }
}
Consumer() {
   while (1) {
         wait(fillCount)
         remove item
         signal(emptyCCount)
         consume item
   }
}
```

```
Producer() {
   while (1) {
         Produce item
         wait(emptyCount)
         lock(mutex)
         insert item
         unlock(mutex)
         signal(fillCount)
   }}
Consumer() {
   while (1) {
         wait(fillCount)
         lock(mutex)
         remove item
         unlock(mutex)
         signal(emptyCCount)
         consume item
   }}
```

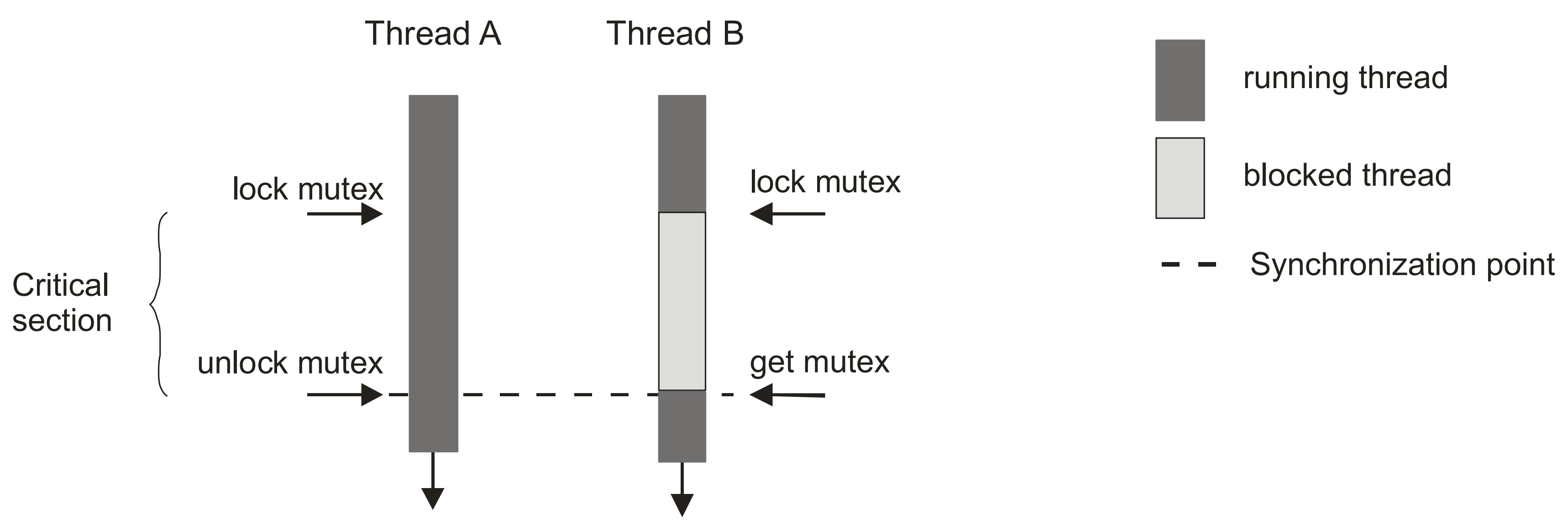
- OK solution for 1 producer and 1 consumer
- For multiple: race condition between producers or consumers
- Solution: Execute critical section with mutual exclusion!!

Mutex

- A mutex is a binary semaphore
  - Used to describe a construct which prevents multiple processes from executing the same code (mutual exclusion in a critical section) / acccessing same data at a given time
  - Usually has an owner – can't unlock if I'm not the one who locked the resource
- Implements two atomic operations
  - **lock(m)** – aka wait
    - If the mutex is already locked the process suspends
  - **unlock(m)** – aka signal
    - If there exists processes blocked waiting wake up one of them

Implementing critical sections using mutex

```
lock(m);     /* enter critical section*/
< critical section >
unlock(m);  /* exit critical section*/
```



Implementing critical sections using mutex

- Problem – not an efficient way to wait! (waste of CPU cycles)
- We need
  - A way for a thread to wait for something w/o wasting CPU cycles
  - A way for a thread to wake up another thread with some signal
- This abstraction is called a condition variable

Condition variables

- Synchronization variable associated with a mutex
- Implements two atomic operations:
  - **c_wait (cond,mutex)**
    - Blocks the calling thread and frees the mutex – aka thread goes to sleep
  - **c_signal(cond)**
    - Unblocks one or more sleeping processes waiting for the condition
    - The processes that wake up and the calling thread compete again for the mutex
- Blocking c.v. (Hoare-style) – signaled thread has priority
- Non-blocking c.v. – signaling thread has priority
  - Signal operation called notify

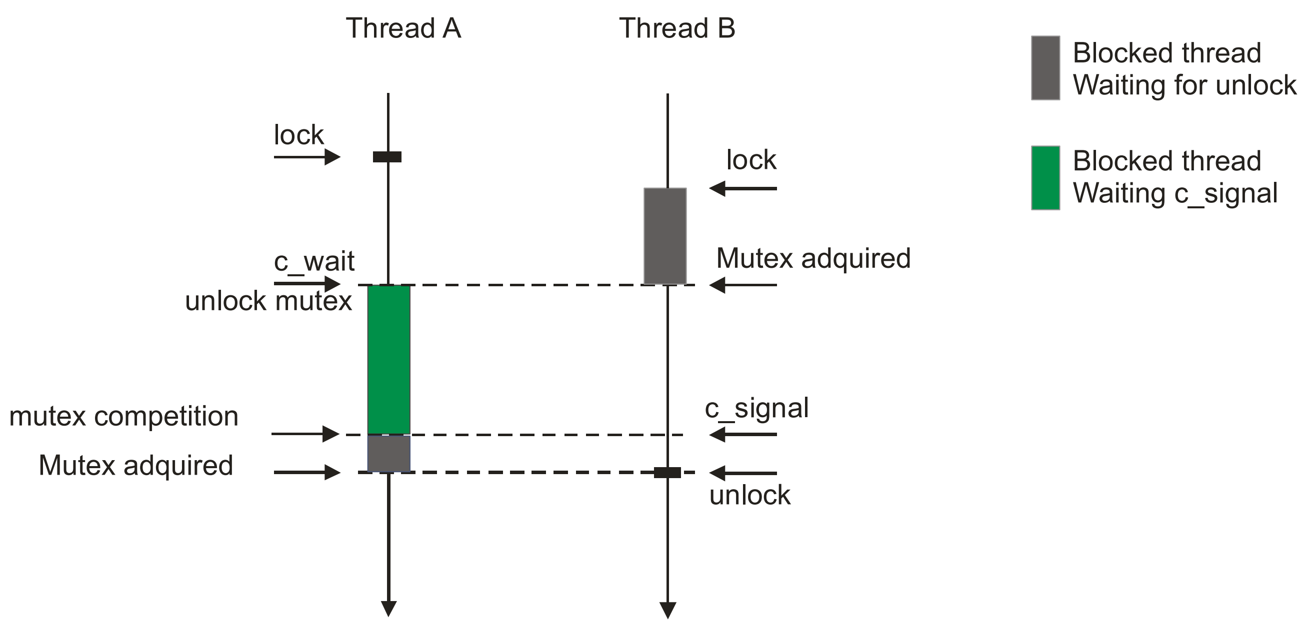
Using mutex and condition variables

Thread A

```
lock(m);
/* Critical section */
while (condition == FALSE)
       c_wait(c,m);
/* critical section */
unlock(m); /* exit critical sect */
```

Thread B

```
lock(m);
/* Critical section */
/* modify condition */
condition = TRUE;
c_signal(c);
/* critical section */
unlock(m); /* exit critical section */
```



Using mutex and condition variables

- Thread A

```
lock(mutex);  /* access resource */
code;
while (resource busy)
       wait(condition,mutex);
mark resource busy;
unlock(mutex);
```

- Thread B

```
lock(mutex);  /* access resource */
mark resource free;
signal(condition);
unlock(mutex);
```

- Important to use `while`

Producer-consumer with mutex and condition variables

```
Productor() {
   while(1){
          produce data
          lock(mutex)
          while buffer_full()
                 c_wait(full,mutex)
          insert item
          if (counter == 1)
              c_signal(empty)
          unlock(mutex)
   }
}
```

```
Consumidor() {
   while(1){
          lock(mutex)
          while buffer_empty()
              c_wait(empty,mutex)
          remove item
          if (counter == size - 1))
             c_signal(full);
          unlock(mutex);
          consume item
   }
}
```

- Important to wait on c.v. within critical section!
  - If lock after wait: race condition
- If wait did not unlock the mutex when going to sleep:
  - If wait after lock then deadlock

Producer-consumer with mutex and condition variables

```
Productor() {
   while(1){
          produce data
          lock(mutex)

          insert item
          unlock(mutex)
          c_signal(cond)
   }
}
```

```
Consumidor() {
   while(1){
          lock(mutex)
          if buffer_empty()
                 c_wait(cond,mutex)

          remove item

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

- What is the problem here?
  - May remove on an empty queue if producer interrupted before signal
  - while instead of if!

UNIX operations for mutex

- Initialization

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

- Destruction

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Lock a mutex; block the thread if lock taken

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

- Unlock a mutex

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

UNIX operations for condition variables

- Initialization

```
int pthread_cond_init(pthread_cond_t*cond,
                      pthread_condattr_t*attr);
```

- Destruction

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Suspend the thread and free associated mutex if wait on condition
  - Until another thread signals that the condition holds
  - When thread wakes up competes for mutex again

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

UNIX operations for condition variables

- Unblock threads and free mutex

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

  - Signal threads blocked on condition
  - No effect if no thread waiting (different from semaphores which increment regardless)

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - All threads suspended on condition wake up

E.g.:
Producer-consumer with mutex (I)

```
#define MAX_BUFFER          1024      /* buffer size */
#define _DATA_   100000    /* data */

pthread_mutex_t mutex;      /* mutex to control access to buffer */
pthread_cond_t not_full;    /* condition variable */
pthread_cond_t not_empty;    /* condition variable */
int n_elements;            /* nr elements in buffer */

int buffer[MAX_BUFFER];     /* shared buffer */

int main(int argc, char *argv[]){
    pthread_t th1, th2;

    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&not_full, NULL);
    pthread_cond_init(&not_empty, NULL);
```

E.g.: Producer-consumer with mutex (II)

```
    pthread_create(&th1, NULL, Producer, NULL);
    pthread_create(&th2, NULL, Consumer, NULL);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&not_full);
    pthread_cond_destroy(&not_empty);

    return 0;
}
```

E.g.:
Producer-consumer with mutex (III)

```
void Producer(void) {
   int data, i ,pos = 0;

   for(i=0; i<_DATA_; i++ ) {
         data = i;                              /* produce data */
         pthread_mutex_lock(&mutex);      /* access the buffer */

         while (n_elements == MAX_BUFFER)/* if buffer full */
                pthread_cond_wait(&not_full, &mutex); /* block */

         buffer[pos] = i;

         pos = (pos + 1) % MAX_BUFFER;
         n_elements = n_elements + 1;

         if (n_elements == 1)
                pthread_cond_signal(&not_empty); /* buffer not empty */

         pthread_mutex_unlock(&mutex);
   }
   pthread_exit(0);
}
```

E.g.:
Producer-consumer with mutex (IV)

```
void Consumier(void)  {
   int data, i ,pos = 0;

   for(i=0; i<_DATA_; i++ ) {
         pthread_mutex_lock(&mutex);         /* access the buffer */

         while (n_elements == 0)             /* if buffer empty */
                pthread_cond_wait(&not_empty, &mutex); /* block */

         data = buffer[pos];

         pos = (pos + 1) % MAX_BUFFER;
         n_elements = n_elements - 1 ;

         if (n_elements == MAX_BUFFER - 1);
                pthread_cond_signal(&not_full); /* buffer not full */

         pthread_mutex_unlock(&mutex);

         printf("Consume %d \n", data);     /* consume data */
   }
   pthread_exit(0);
}
```