OPERATING SYSTEMS: PROCESS COMMUNICATION AND SYNCHRONIZATION

Threads and communication and synchronization mechanisms

Content

- **Communication and synchronization.**
- Semaphores.
- The readers/writers problem.
  - Semaphores based solution.
- Mutex and condition variables.

Communication mechanisms

- **Communication mechanisms** allow for information **transfer** between two processes.

- Files.
- Pipes (pipes, FIFOs).
- **Shared memory variables.**
- Message passing.

Synchronization mechanisms

- Synchronization mechanisms allow to enforce that a process stops its execution until an event happens in another process.

- Concurrent languages constructs (threads).
- Operating system services:
  - Signals (asynchrony).
  - Pipes (pipes, FIFOs).
  - **Semaphores.**
  - **Mutex and condition variables.**
  - Message passing.

- Synchronization operations must be **atomic**.

Content

- Communication and synchronization.
- **Semaphores.**
- The readers/writers problem.
  - Semaphores based solution.
- Mutex and condition variables.

Semaphores

- Synchronization mechanism.
- Within the same machine.
- Object with an associated integer value.
- Two **atomic** operations.
  - **wait**
  - **signal**

POSIX semaphores

- Synchronization mechanism for processes or threads running in the same machine
- POSIX semaphores come in two forms:
  - **Named semaphores**: can be used by different processes just knowing the name. Does not required shared memory.
  - **Unnamed semaphores**: can be used only by the process that created it (with threads) or by other using a shared memory region.

```
#include <semaphore.h>
sem_t * semaphore; //named
sem_t semaphore; // un-named
```

POSIX semaphores

- int sem_init(sem_t *sem, int shared, int val);
  - Initializes unnamed semaphore.
- int sem_destroy(sem_t *sem);
  - Destroys unnamed semaphore.
- sem_t *sem_open(char *name, int flag, mode_t mode, int val);
  - Opens (creates) a named semaphore.
- int sem_close(sem_t *sem);
  - Closes a named semaphore.
- int sem_unlink(char *name);
  - Deletes a named semaphore.
- int sem_wait(sem_t *sem);
  - Performs wait operation on a semaphore.
- int sem_trywait (sem_t *sem);
  - Try wait. If blocked returns -1 without doing anything.
- int sem_post(sem_t *sem);
  - Performs **signal** operation on a semaphore.

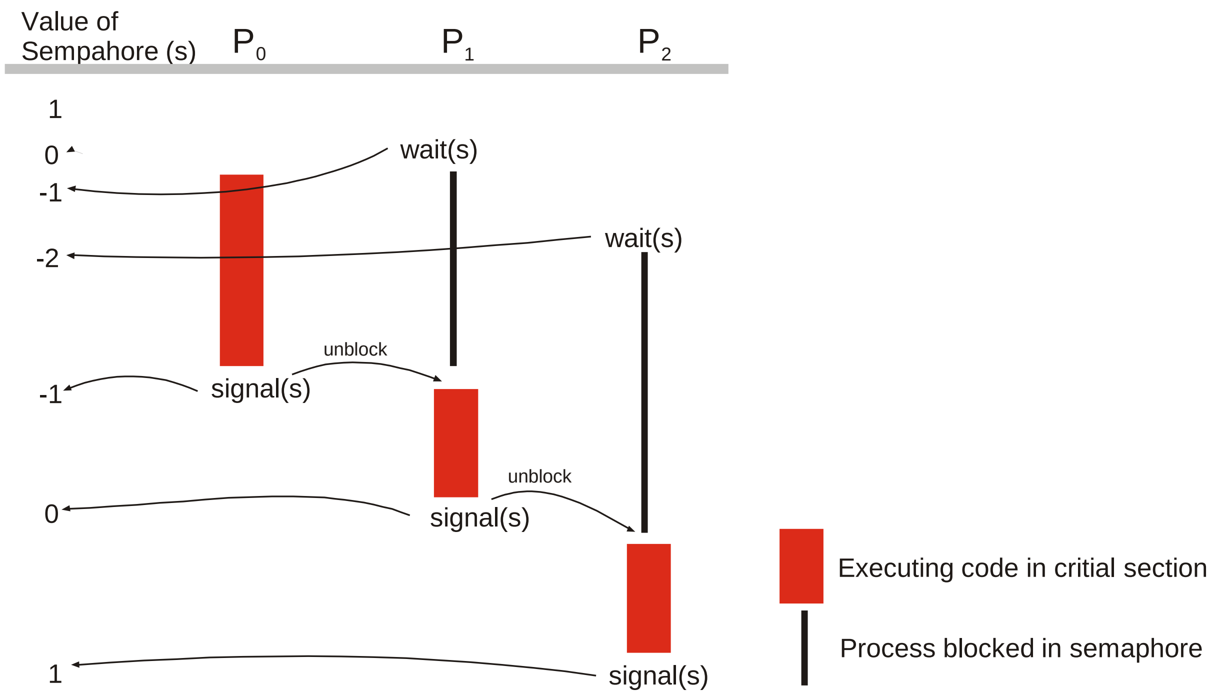
Critical sections with semaphores

sem_wait(s); */* critical section entry*/*

< critical section>

sem_post (s); */* critical section exit */*

- Semaphore must have initially value 1.



Operations on semaphores


```
sem_wait(s) {
   s = s - 1;
   if (s < 0) {
      <Block process>
   }
}

sem_post (s) {
   s = s + 1;
   if (s <= 0) {
      <Unblock a blocked process by wait operation>
   }
}
```

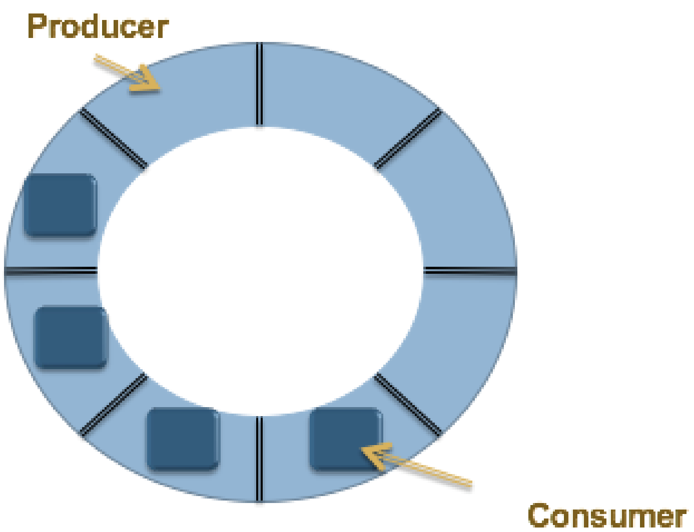
Unnamed semaphores: Producer consumer with semaphores


```
#define MAX_BUFFER        1024    /* buffer size*/
#define DATA_SIZE  100000    /* number of data items to produce */

sem_t elements;                   /* elements in buffer*/
sem_t holes;                      /* holes in buffer*/
int buffer[MAX_BUFFER];           /* common buffer*/

void main(void)
{
  pthread_t th1, th2;  /* threads identifiers*/

  /* Initialize semaphores*/
  sem_init(&elements, 0, 0);
  sem_init(&holes, 0, MAX_BUFFER);
```



Unnamed semaphores: Producer consumer with semaphores


```
  /* create threads*/
  pthread_create(&th1, NULL, producer, NULL);
  pthread_create(&th2, NULL, consumer, NULL);

  /* wait for termination*/
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  sem_destroy(&holes);
  sem_destroy(&elements);

  exit(0);
}
```

Unnamed semaphores: Producer & Consumer threads


```
void producer() {   /* Producer code*/
  int pos = 0;  /* position in buffer*/
  int data;     /* data to be produced */
  int i;

  for(i=0; i < DATA_SIZE; i++ )   {
    data = i;          /* produce data*/

    sem_wait(&holes);  /* Reduce holes by 1*/
    buffer[pos] = i;
    pos = (pos + 1) % MAX_BUFFER;
    sem_post(&elements);  /* one element
     more*/
  }
 pthread_exit(0);
}
```

```
void consumer()  { /* Consumer code */
  int pos = 0;
  int data;
  int i;

  for(i=0; i < DATA_SIZE; i++ ) {
    sem_wait(&elements);    /* one element less*/
    data= buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    sem_post(&huecos);    /* one hole more*/

    store_data(data); /* consume data */
  }
  pthread_exit(0);
}
```

Content

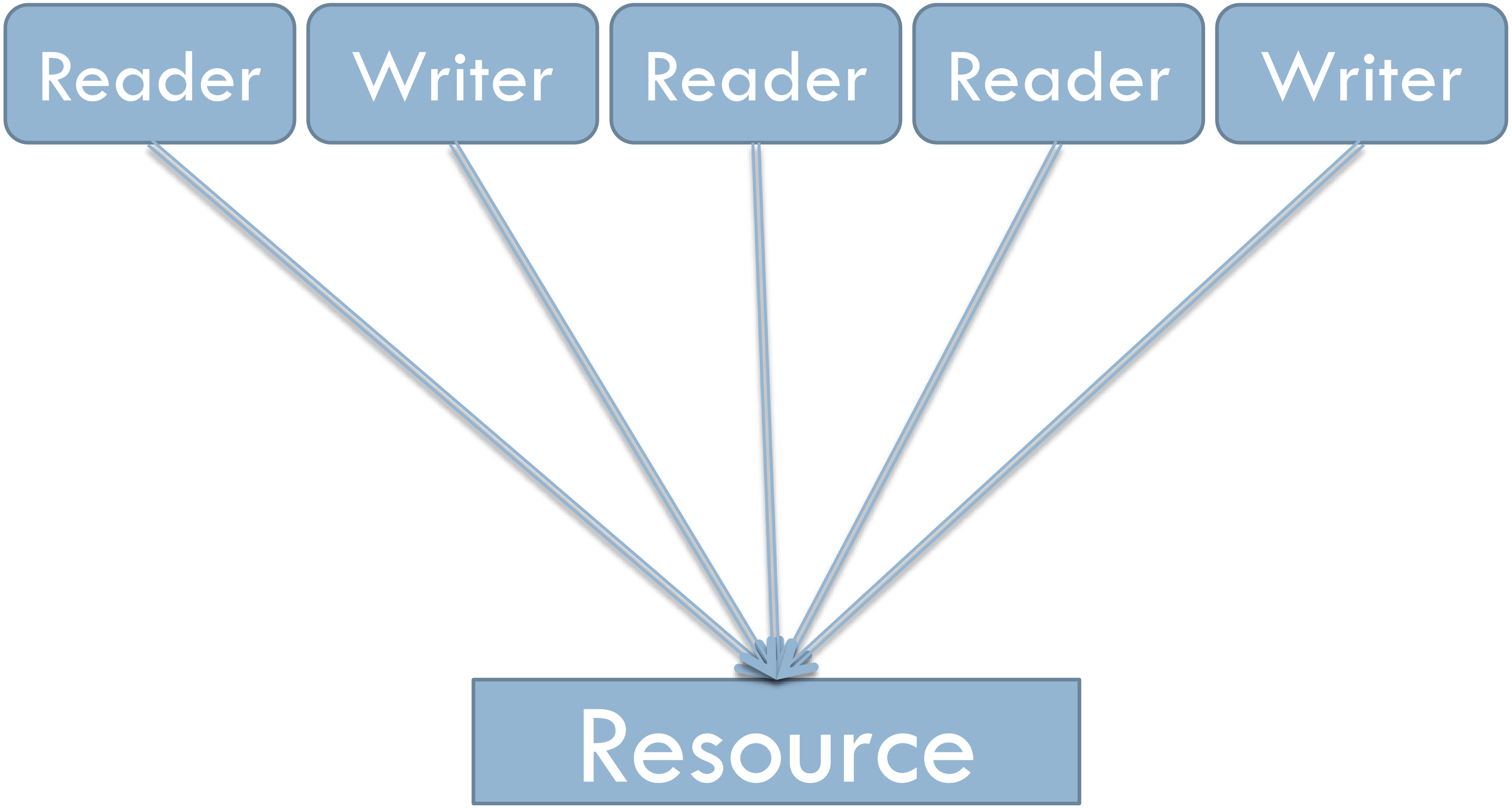
- Communication and synchronization.
- Semaphores.
- **The readers/writers problem.**
  - Semaphores based solution.
- Mutex and condition variables.

Readers-writes problem

- Problem when using a shared storage area.
  - Multiple processes reading information.
  - Multiple processes writing information.

- **Conditions:**
  - Any number of readers may read from the data area concurrently.
    - Read concurrency allowed.
  - Only one writer may modify information at once.
    - No concurrent writes allowed.
  - During a write no reader may perform a query.
    - No read/write concurrency allowed.

Readers-writers problem



Differences with other problems

- **Mutual exclusion**
  - Mutual exclusion would only allow a process to access at the same time to information.
  - No concurrency allowed between readers.
  - Higher contention.
- **Producer/Consumer**:
  - In producer/consumer both processes modify the shared data area.

- Goals of additional constraints:
  - **Provide a more efficient solution.**

Management alternatives

- **Readers have priority.**
  - If there is some reader in critical section other readers may enter.
  - A writer can only enter the critical section if there is no process there.
  - **Problem**: Starvation for writers.

- **Writers have priority.**
  - When a writer wishes to access to the critical section no more readers are admitted.

Readers have priority


```
int nreaders; sempahore rd=1; semaphore wr=1;
```

Lector

```
for(;;) {
  semWait(rd);
  nreaders++;
  if (nreaders==1)
    semWait(wr);
  semSignal(rd);

  perform_read();

  semWait(rd);
  nreaders--;
  if (nreaders==0)
    semSignal(wr);
  semSignal(rd);
}
```

Escritor

```
for(;;) {
  semWait(wr);
  perform_write();
  semSignal(wr);
}
```

Unnamed semaphores: Readers-writers with semaphores


```
int data = 5;       /* resource*/
int nreaders = 0; /* number of readers */
sem_t sem_rd;       /* controls access to nreaders*/
sem_t mutex;        /* controls access to data*/

void main(void) {
  pthread_t th1, th2, th3, th4;

  sem_init(&mutex, 0, 1);
  sem_init(&sem_rd, 0, 1);

  pthread_create(&th1, NULL, reader, NULL);
  pthread_create(&th2, NULL, writer, NULL);
  pthread_create(&th3, NULL, reader, NULL);
  pthread_create(&th4, NULL, writer, NULL);
```

Unnamed semaphores: Readers-writers with semaphores


```
 pthread_join(th1, NULL);
 pthread_join(th2, NULL);
 pthread_join(th3, NULL);
 pthread_join(th4, NULL);

 /* cerrar todos los semaforos */
 sem_destroy(&mutex);
 sem_destroy(&sem_rd);

 exit(0);
}
```

Unnamed semaphores: Readers & writers threads


```
void reader() { /* reader code */
  sem_wait(&sem_rd);
  nreaders = nreaders + 1;
  if (nreaders == 1) sem_wait(&mutex);
  sem_post(&sem_rd);

  printf(``%d\n'', data);  /* read data and print
      it*/

  sem_wait(&sem_rd);
  n_readers = n_readers - 1;
  if (n_readers == 0) sem_post(&mutex);
  sem_post(&sem_rd);
  pthread_exit(0);
}
```

```
void writer() {   /* writer code */
  sem_wait(&mutex);
  data = data + 2;     /* modify resource
     */
  sem_post(&mutex);

  pthread_exit(0);
}
```

Named semaphores: Names

- To synchronize processes without using shared memory.
- Name: string similar to a file name.
  - If name is relative, can only be accessed by the creator and its children.
  - If name if absolute (starts with "/"), the semaphore can be shared by any process knowing the name and having permissions.
- Standard way to create semaphores to be used by parent and children
  - Unnamed semaphores not valid, as parent and children DO NOT share memory.

Named semaphores: Creation and use

- To create it:

```
sem_t *sem_open(char *name, int flag, mode_t mode,int val);
```

  - Flag = O_CREAT create it.
  - Flag: O_CREAT | O_EXECL. Create, but returns -1 if exist.
  - Mode: access permissions;
  - Val: initial value of the semaphore (>=0);
- To use it:

```
sem_t *sem_open(char *name, int flag);
```

  - With flag 0. It does not exist returns -1.
- Important:
  - All processes must know the "name" and use the same.

Named semaphores: Readers - Writers


```
int dato = 5;       /* recurso */
int n_lectores = 0; /* num lectores */
sem_t *sem_lec; sem_t *mutex;


int main (int argc, char *argv[]) {


int i, n= 5; pid_t pid;
/* Named semaphore*/
if((mutex=sem_open("/tmp/sem_1", O_CREAT, 0644, 1))==(sem_t *)-1)
   { perror("It is possible to create"); exit(1); }
if((sem_lec=sem_open("/tmp/sem_2", O_CREAT, 0644, 1))==(sem_t *)-1)
   { perror(" It is possible to create "); exit(1); }
```

```
/* Crea los procesos */
for (i = 1; i< atoi(argv[1]); ++i){
   pid = fork();
   if (pid ==-1)
      { perror("No se puede crear el proceso");
        exit(-1);}
   else if(pid==0) {  /child
      reader(getpid());   break;
    }
    writer(pid); /* parent */
  }
   sem_close(mutex);  sem_close(sem_lec);
   sem_unlink("/tmp/sem_1");
   sem_unlink("/tmp/sem_2");
}
```

Named semaphores: Readers & writers processes


```
void reader (int pid) {
  sem_wait(sem_lec);
  n_lectores = n_lectores + 1;
  if (n_lectores == 1)
      sem_wait(mutex);
  sem_post(sem_lec);
  printf("  lector %d  dato: %d\n", pid, dato);  /* leer dato */
  sem_wait(sem_lec);
  n_lectores = n_lectores - 1;
  if (n_lectores == 0)
      sem_post(mutex);
  sem_post(sem_lec);
}
```

```
void writer (int pid) {
sem_wait(mutex);
   dato = dato + 2;      /* modificar
el recurso */
   printf("escritor %d  dato: %d\n",
pid, dato);  /* leer dato */
   sem_post(mutex);
}
```

Content

- Communication and synchronization.
- Semaphores.
- The readers/writers problem.
  - Semaphores based solution.
- **Mutex and condition variables.**

Mutex and condition variables

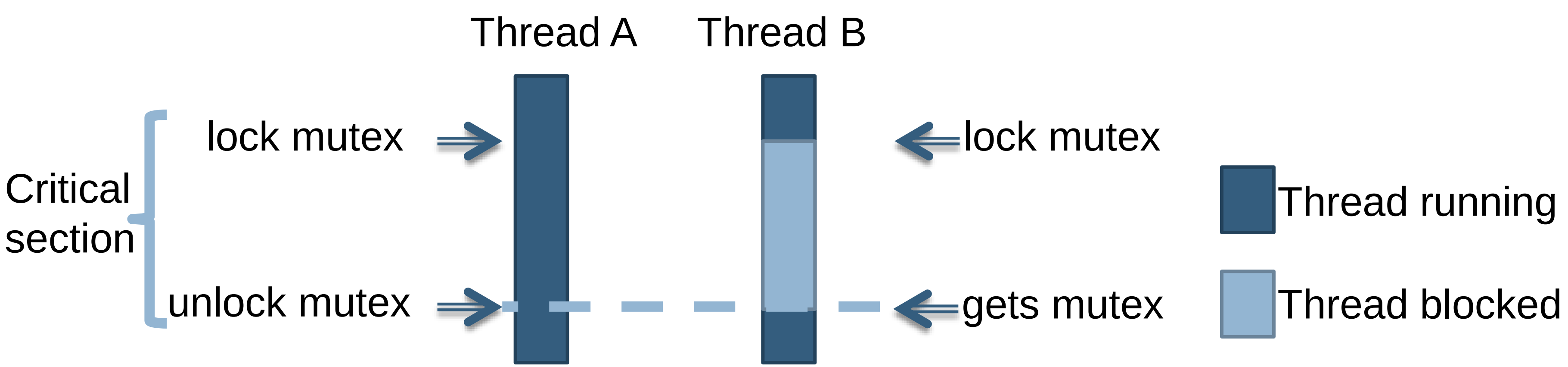
- A **mutex** is a synchronization mechanism for threads.

- It is a binary semaphore with to **atomic** operations.
  - **lock(m)** Try to block the mutex. If the mutex is already blocked the calling thread is suspended.
  - **unlock(m)** Unblocks the mutex. If there are processes blocked in the mutex one is unblocked.

Critical sections with mutexes


```
lock(m);      /* entry into critical section */
<critical section>
unlock(s);   /* exit from critical section */
```

- **unlock** operation must be performed by the thread that performed **lock**

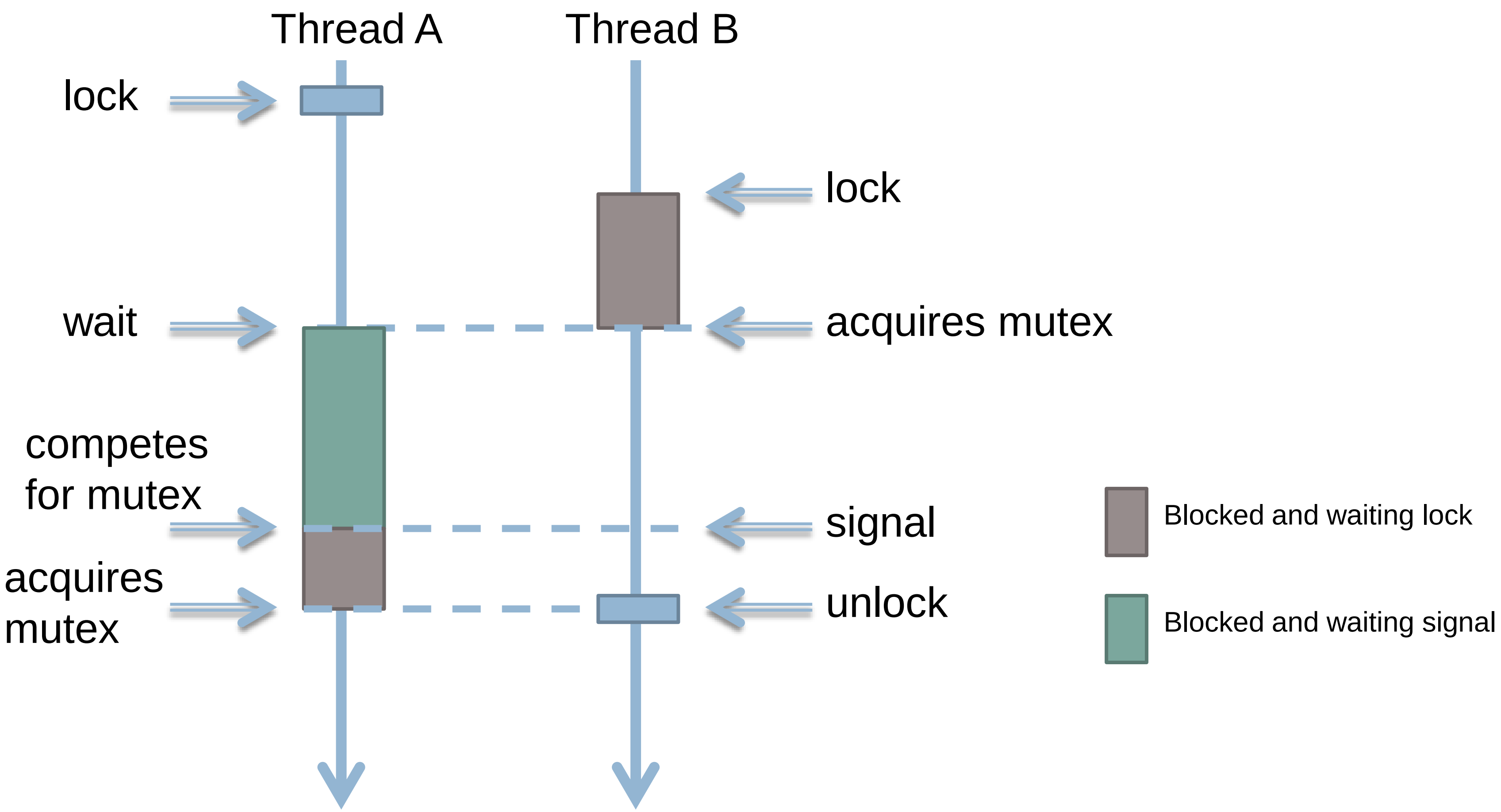


Condition variables

- Synchronization variables associated to a mutex.
- Two **atomic** operations:
  - **wait** Blocks running thread and releases mutex.
  - **signal** Unblocks one or more threads suspended in the condition variable. Unblocked threads contend for acquiring the mutex again.

- It is convenient to run them in a **lock/unlock** block.

Condition variables



Using mutexes and condition variables

Thread A

lock(mutex); /* access to resource */

<check data structures>

while (resource is busy) {

    wait (condition, mutex);

}

<mark resource as busy>

unlock (mutex);

Thread B

lock (mutex); /* access to resource*/

<mark resource as free>

signal (condition, mutex);

unlock (mutex);

Important to use while

POSIX services

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

- Initialize mutex.

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

- Destroy mutex.

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

- Try to get access to mutex.
- Blocks thread if mutex is already acquired by other thread.

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

- Unblock mutex.

POSIX services

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

- Initialize a condition variable.

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

- Destroy a condition variable.

POSIX services

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

- Unblocks one or more threads that are suspended in the condition variable **cond**.
- Has no effect if there is no thread waiting (difference with semaphores).

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

- All blocked threads in condition variable **cond** are unblocked.
- Has no effect if there is no thread waiting.

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

- Suspend thread until another thread signals condition variable **cond**.
- Automatically releases the **mutex**. When thread is unblocked it contends again for the **mutex**.

Producer-Consumer with mutexes


```
#define MAX_BUFFER      1024    /* size of buffer*/
#define DATA_SIZE  100000   /* number of data items to be produced*/

pthread_mutex_t mutex; /* mutex to access shared buffer */
pthread_cond_t non_full;  /* can we add more elements? */
pthread_cond_t non_empty;  /* can we remove elements? */
int n_elements;          /* number of elements in buffer */
int buffer[MAX_BUFFER];   /* common buffer */

int main() {
   pthread_t th1, th2;

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&non_full, NULL);
   pthread_cond_init(&non_empty, NULL);
```

Producer-Consumer with mutexes


```
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(&non_full);
pthread_cond_destroy(&non_empty);

exit(0);
}
```

Producer


```
void producer() {   /* Producer code */
 int data, i ,pos = 0;
 for(i=0; i < DATA_SIZE; i++ ) {
   data= i;       /* generate data */
   pthread_mutex_lock(&mutex);       /* access to buffer*/
   while (n_elements == MAX_BUFFER)  /* when buffer is full*/
     pthread_cond_wait(&non_full, &mutex);
   buffer[pos] = data;
   pos = (pos + 1) % MAX_BUFFER;
   n_elements ++;
   pthread_cond_signal(&non_empty);   /* buffer is not empty */
   pthread_mutex_unlock(&mutex);
 }
 pthread_exit(0);
}
```

Consumer


```
void consumer() {   /* consumer code */
  int dato, i ,pos = 0;
  for(i=0; i < DATA_SIZE; i++ ) {
    pthread_mutex_lock(&mutex);    /* access to buffer */
    while (n_elements == 0)       /* when buffer empty */
      pthread_cond_wait(&non_empty, &mutex);
    dato = buffer[pos];
    pos = (pos + 1) % MAX_BUFFER;
    n_elements --;
    pthread_cond_signal(&non_full);   /* buffer is not full */
    pthread_mutex_unlock(&mutex);
    printf("Consumed %d \n", data);     /* Use data*/
  }
  pthread_exit(0);
}
```

Readers writers with mutex


```
int data= 5;                /* resource*/
int nreaders = 0;           /* number of readers */
pthread_mutex_t data_mutex;         /* Control access to data*/
pthread_mutex_t mutex_rd; /* Controls access to nreaders */

main(int argc, char *argv[])  {
   pthread_t th1, th2, th3, th4;

   pthread_mutex_init(&data_mutex, NULL);
   pthread_mutex_init(&mutex_rd, NULL);

   pthread_create(&th1, NULL, Lector, NULL);
   pthread_create(&th2, NULL, Escritor, NULL);
   pthread_create(&th3, NULL, Lector, NULL);
   pthread_create(&th4, NULL, Escritor, NULL);
```

Lectores-escritores con mutex


```
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  pthread_join(th3, NULL);
  pthread_join(th4, NULL);

  pthread_mutex_destroy(&data_mutex);
  pthread_mutex_destroy(&mutex_rd);

  exit(0);
}
```

Writer


```
void writer() {   /* writer code*/
  pthread_mutex_lock(&data_mutex);
  dato = dato + 2;     /* modify resource*/
  pthread_mutex_unlock(&data_mutex);
  pthread_exit(0);
}
```

Reader


```
void reader() { /* codigo del lector */
  pthread_mutex_lock(&mutex_rd);
  nreaders++;
  if (nreaders == 1) pthread_mutex_lock(&data_mutex);
  pthread_mutex_unlock(&mutex_rd);

  printf("%d\n", data);  /* read data and print it*/

  pthread_mutex_lock(&mutex_rd);
  nreaders--;
  if (nreaders == 0) pthread_mutex_unlock(&data_mutex);
  pthread_mutex_unlock(&mutex_rd);

  pthread_exit(0);
}
```

OPERATING SYSTEMS: PROCESS COMMUNICATION AND SYNCHRONIZATION

Threads and communication and synchronization mechanisms