Working With Threads

OPERATING SYSTEMS COURSE

THE HEBREW UNIVERSITY

SPRING 2023

Outline

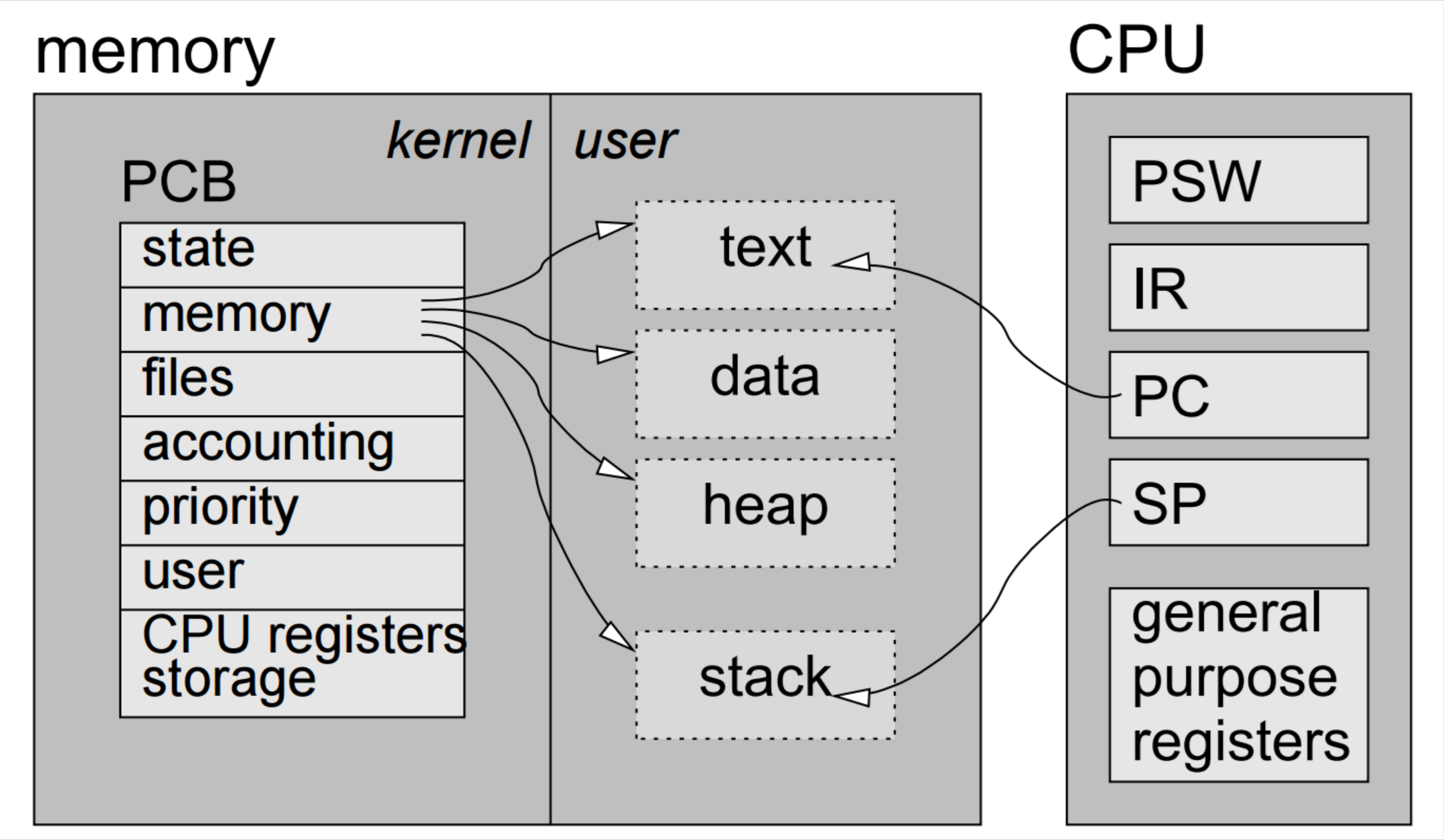
- Background – Threads and processes
- Concurrency & Threads Managements
  - Pthread
  - Mutex
  - Monitor
  - Atomic variable

Processes & Threads

Multi-threads / Multi-processes motivations

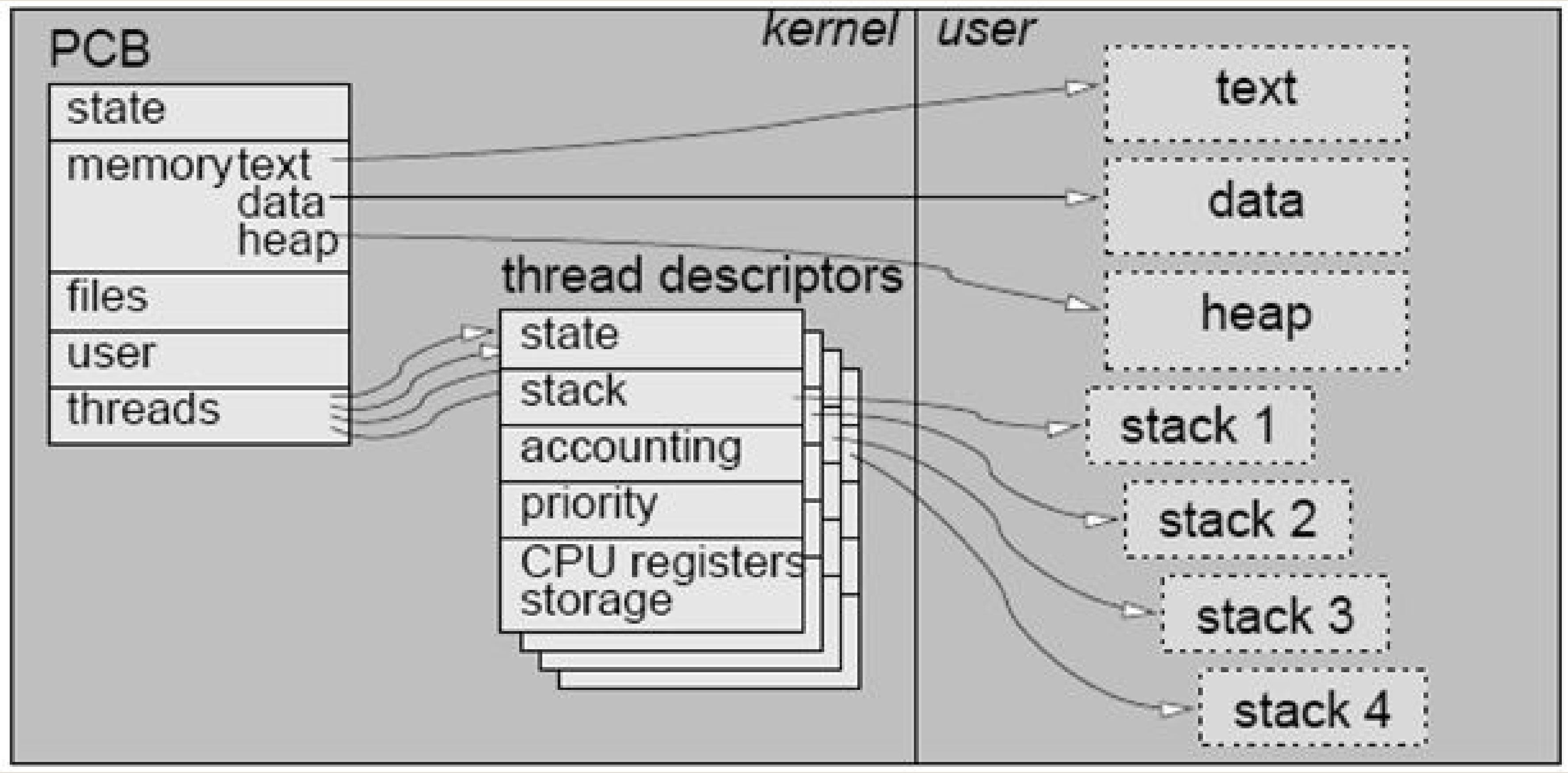
- Using multi-processors
- Some operations are blocking (such as IO access), and we might use the processors meanwhile
- We have several tasks that need to run

Process

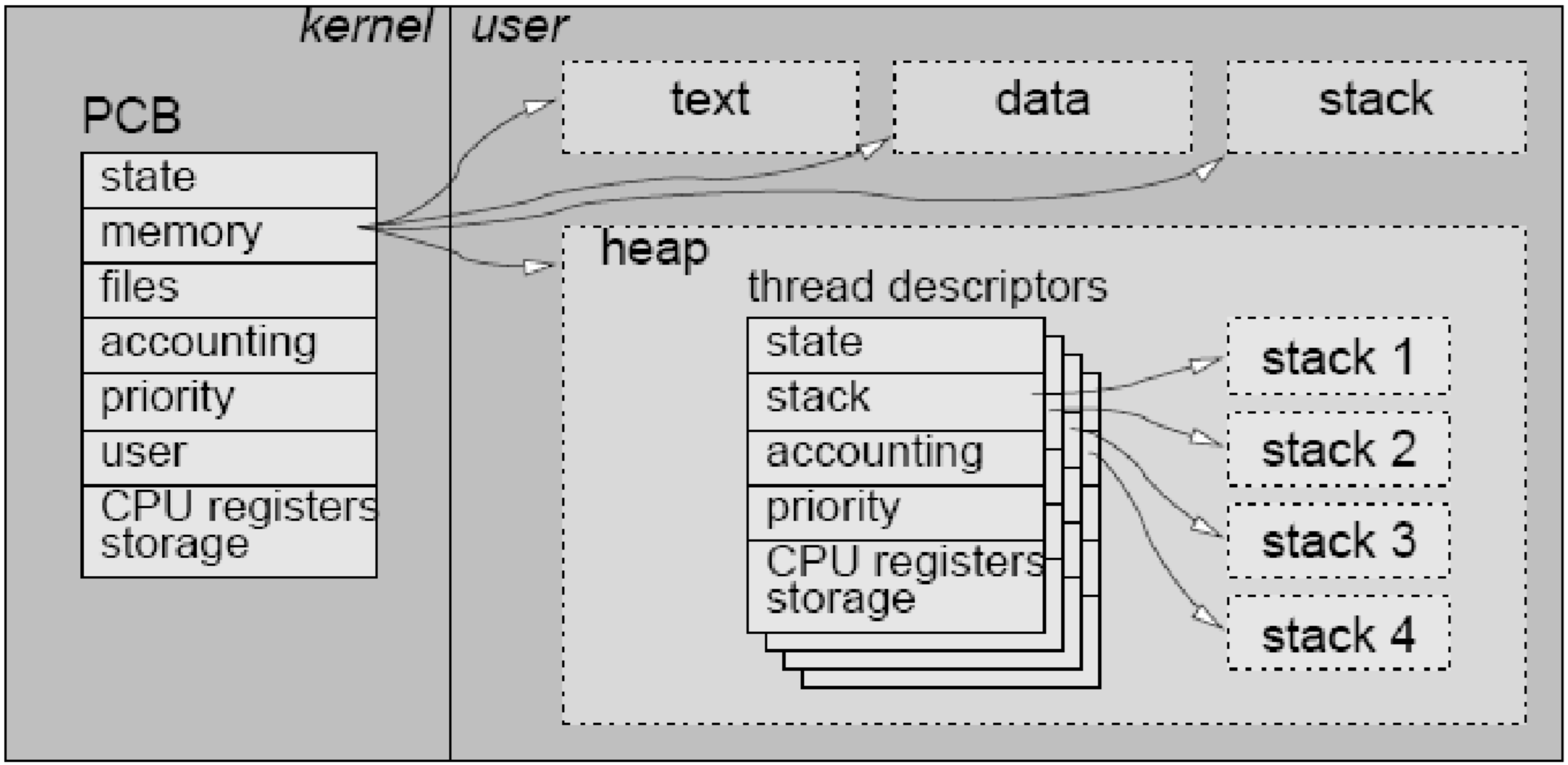


memory
CPU
kernel
user
PCB
state
memory
files
accounting
priority
user
CPU registers storage
text
data
heap
stack
PSW
IR
PC
SP
general purpose registers

Kernel Level Threads



User Level Thread



Comparison

	Processes	Kernel Threads	User Threads
Interaction between instances	protected from each other, require operating system to communicate	share address space, simple communication, useful for application structuring	
Context-switch overhead	high overhead: all operations require a kernel trap, significant work	medium overhead: operations require a kernel trap, but little work	low overhead: everything is done at user level
Blocking granularity	independent: if one blocks, this does not affect the others		if a thread blocks the whole process is blocked
Multi-core utilization	can run on different processors in a multiprocessor system		all share the same processor
OS dependency	system specific API, programs are not portable		the same thread library may be available on several systems
Scheduling	one size fits all		application-specific thread management is possible

Fast Recap

- Multi-Threads/Multi-Process Motivation
- Threads & Process data structures
- Cons & Pros

Thread Management

Creating Threads

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer

```
int pthread_create (
  pthread_t *thread,
  const pthread_attr_t *attr=NULL,
  void *(*start_routine) (void *),
  void *arg) ;
```

Terminating Thread Execution

- A thread terminates in one of the following ways:
  - The thread makes a call to the `pthread_exit(void *status)` subroutine, specifying an exit status
    - This value will be available to a different thread in the same process that calls `pthread_join`
  - The thread returns from its `start_routine`
  - The thread is cancelled (via `pthread_cancel(pthread_t thread)`).
  - The entire process is terminated due to a call to either
    - Exit subroutines
    - Return from the main thread

    Note: exec subroutines has the same effect.

Example
Creating 5 threads and exiting


```c
#define NUM_THREADS 5

void *print_hello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    return NULL;  // Equivalent to calling pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, print_hello, (void *) &t);
        if (res < 0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
}
```

Example
Creating 5 threads and exiting


```
#define NUM_THREADS 5

void *print_hello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    return NULL;  // Equivalent to calling pthread
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, print_hello, (void *) &t);
        if (res < 0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
}
```

Doesn't work – the program may terminate before running all the threads

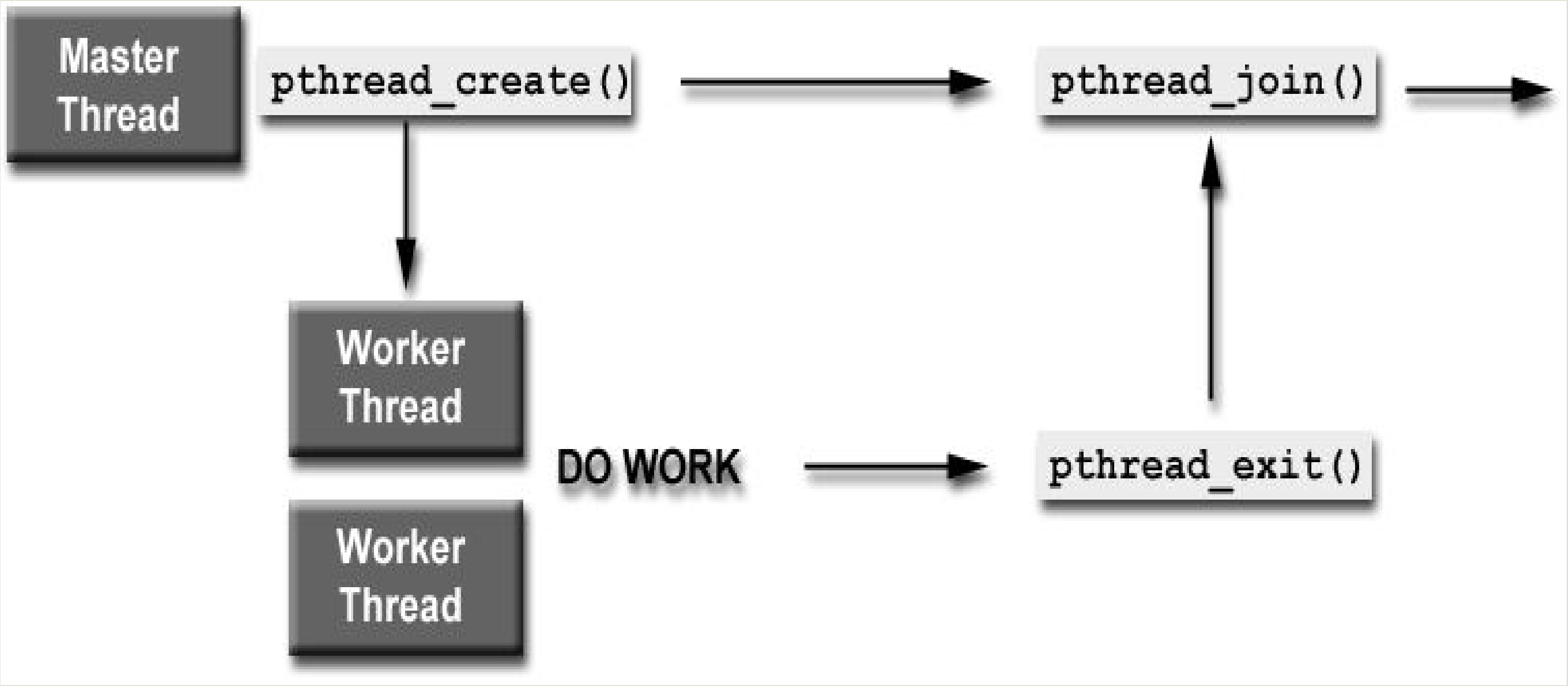
Joining Threads


```
int pthread_join(pthread_t thread,
                 void **ret_val)
```

- blocks the calling thread until the specified thread `thread` terminates
- The programmer is able to obtain the target thread's termination return status if specified through

```
pthread_exit(void *status)
```

Working with Threads Flow



Example
Creating 5 threads and exiting


```c
#define NUM_THREADS 5

void *print_hello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    return NULL;  // Equivalent to calling pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for (t = 0; t < NUM_THREADS; t++) {
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL, print_hello, (void *) &t);
        if (res < 0) {
            printf("ERROR\n");
            exit(-1);
        }
    }
         // main CONTINUES IN THE NEXT SLIDE
```

Example Cont.


```
// main thread waits for the other threads
for(t=0; t<NUM_THREADS; t++) {
    res = pthread_join(threads[t], (void **) &status);
    if (res<0) {
        printf("ERROR\n");
        exit(-1);
    }
    printf("Completed join with thread %d status= %d\n",
          t, *status);
}
```

Concurrency & Threads Management

Concurrency pros and cons

- Concurrency is good for users
  - Different types of multitasking: working on the same problem, background execution, etc.
  - Improves the performance (CPUs utilization).

- Concurrency is challenging
  - Access to shared data structures
  - Danger of deadlock due to resource contention
  - Preventing Starvation –
    - Starvation - where a process/thread is denied necessary resources for ever

The Critical Section Problem

- n processes $P_0, \ldots, P_{n-1}$
- No assumptions on processing speeds
  - Models inherent non-determinism of process scheduling
- No assumptions on process activity when executing within critical section and remainder sections

- The problem: Implement a general mechanism for entering and leaving a critical section.

Success Criteria

1. *Mutual exclusion*: Only one process is in the critical section at a time.
2. *Progress*: If processes want to get into the critical section one of them will eventually get into the critical section.
3. *Starvation free*: No process will wait indefinitely while other processes continuously enter the critical section (also called “Bounded waiting”).
4. *Generality*: It works for *N* processes and many processors.
5. *No blocking in the reminder*: No process running outside its critical section may block other process

Peterson's Algorithm for 2 threads


```
bool want[0] = false;
bool want[1] = false;
int turn;
```

Thread (i = 0)	Thread 1 (i = 1)

`want[i] = true;`
`turn = 1-i;`
`while (want[1-i] && turn == 1-i) { // busy wait }`
 	
`// critical section ...`
`want[i] = false;`
	`want[i] = true;`
	`turn = 1-i;`
	`while (want[1-i] && turn == 1-i) { // busy wait }`
	
	`// critical section ...`
	`want[i] = false;`

Peterson’s Algorithm Summary

- Peterson’s algorithm creates a critical section mechanism without any help from the OS.
- All the success criteria hold for this algorithm (except generality).
- It does use busy wait.

Mutex

A mechanism designed to avoid race condition

Race Condition Example

Thread 1:

```
int a = counter;
a++;
counter = a;
```

Thread 2:

```
int b = counter;
b--;
counter = b;
```

Race Condition Example

Thread 1:

```
int a = counter;
a++;
counter = a;
```

Thread 2:

```
int b = counter;
b--;
counter = b;
```

Possible results:
new counter = counter
new counter = counter -1
new counter = counter +1

Reminder from class

- Critical section is a peace of code that access **shared resources**.

- Mutual exclusion algorithms are used to avoid the simultaneous execution of a critical section.
  - Pay attention we protect shared variables, not pieces of code!

- Common protection mechanism – Mutex
  - We create a mutex to protect each shared variable.

Mutex Work Flow

- A typical sequence in the use of a mutex is as follows:

1. Create and initialize a mutex variable
2. Several threads attempt to lock the mutex
3. Only one succeeds and that thread owns the mutex
4. The owner thread performs some set of actions
5. The owner unlocks the mutex
6. Another thread acquires the mutex and repeats the process
7. Finally the mutex is destroyed

Creating and Destroying Mutex

- Mutex variables must be declared with type pthread_mutex_t, and must be initialized before they can be used:
  - Statically, when it is declared - for example:
    `pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`
  - Dynamically,

    `pthread_mutex_init(mutex, attr)`

    This allows setting mutex attributes (default settings use NULL)
- `pthread_mutex_destroy(mutex)` should be used to free a mutex object when is no longer needed

Locking a Mutex

- The `pthread_mutex_lock(*mutex)` routine is used by a thread to acquire a lock on the specified mutex variable
- If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked

Unlock a Mutex

- `pthread_mutex_unlock(*mutex)` will unlock a mutex if called by the owning thread. Calling this routine is required after a thread has completed its use of protected data
- An error will be returned if:
  - If the mutex was already unlocked
  - If the mutex is owned by another thread

- Pay attention – it means that in pthread, mutex designed only for mutual exclusion.

Fixed Example

Thread 1:

```
pthread_mutex_lock(
        &mut_counter);

int a = counter;
a++;
counter = a;

pthread_mutex_unlock(
        &mut_counter);
```

Thread 2:

```
pthread_mutex_lock(
        &mut_counter);

int b = counter;
b--;
counter = b;

pthread_mutex_unlock(
        &mut_counter);
```

`// Checking return values omitted for brevity`

Beware of Deadlocks!

Thread 1: **Thread 2:**

// Locking two different resources

```
lock(a_mutex);
lock(b_mutex);

a=b+a;
b=b*a;

unlock(b_mutex);
unlock(a_mutex);
```

```
lock(b_mutex);
lock(a_mutex);

b=a+b;
a=b*a;

unlock(a_mutex);
unlock(b_mutex);
```

Beware of Deadlocks!

Thread 1: **Thread 2:**

// Locking two different resources

```
lock(a_mutex);
lock(b_mutex);

a=b+a;
b=b*a;

unlock(b_mutex);
unlock(a_mutex);
```

Can't lock, Thread 2 owns it!

```
lock(b_mutex);
lock(a_mutex);

b=a+b;
a=b*a;

unlock(a_mutex);
unlock(b_mutex);
```

Beware of Deadlocks!

Thread 1:

Thread 2:

// Locking two different resources

Thread 1:

```
lock(a_mutex);
lock(b_mutex);

a=b+a;
b=b*a;

unlock(b_mutex);
unlock(a_mutex);
```

Can't lock,
Thread 2 owns it!

Thread 2:

```
lock(b_mutex);
lock(a_mutex);

b=a+b;
a=b*a;

unlock(a_mutex);
unlock(b_mutex);
```

Can't lock,
Thread 1 owns it!

Beware of Deadlocks!

Thread 1: **Thread 2:**

// Locking two different resources

```
lock(a_mutex);
lock(b_mutex);

a=b+a;
b=b*a;

unlock(b_mutex);
unlock(a_mutex);
```

Can't lock, Thread 2 owns it!

```
lock(b_mutex);
lock(a_mutex);

b=a+b;
a=b*a;

unlock(a_mutex);
unlock(b_mutex);
```

Can't lock, Thread 1 owns it!

Monitors (Conditional Variables)

A mechanism designed to synchronize between threads

Thread synchronization

- pthread_mutexes_t was designed for a specific type of synchronization – mutual exclusion.

- Many times other synchronization types are needed
  - Bounded-Buffer (Consumer-Producer)
  - Barrier
  - Reader-Writers

- This can be implemented with pthread_mutex_t and a loop

Simple synchronization example – Thread 1 needs to use thread 2's var

Thread 1:

```
while (true) {
  pthread_mutex_lock(
         &mut_var);

  if (canUseVar) {
    // use var
    canUseVar = false;
  }

  pthread_mutex_unlock(
         &mut_var);
}
```

Thread 2:

```
pthread_mutex_lock(
       &mut_var);

init(var);
canUseVar = true;

pthread_mutex_unlock(
       &mut_var);
```

Simple synchronization example – Thread 1 needs to use thread 2's var

Thread 1:

```
while (true) {
  pthread_mutex_lock(
          &mut_var);

  if (canUseVar) {
    // use var
    canUseVar = false;
  }

  pthread_mutex_unlock(
          &mut_var);
}
```

Thread 2:

```
pthread_mutex_lock(
        &mut_var);

init(var);
canUseVar = true;

pthread_mutex_unlock(
        &mut_var);
```

Complicated and inefficient (busy waiting)!

Using Monitors (Conditional Variables)

- Monitors were designed for threads synchronization
  - No need for busy waiting

- A Monitor receives a mutex, and uses it to achieve mutual exclusion and synchronization simultaneously

- Two main functions:
  - Wait(monitor , mutex) – Release the mutex and blocks the current thread
  - Signal(monitor) – Unblocks a random thread. The thread will wait for the mutex to be available and then it will resume.

- The concept is to use Signal when a **certain condition is met**
  - Therefore they are often called Conditional Variables (CV)

Simple synchronization example – Thread 1 needs to use thread 2's var

Thread 1:

```
pthread_mutex_lock(
       &mut_var);

if (canUseVar) {
  pthread_cond_wait(
         &cv_var, &mut_var);

}
// use var
canUseVar = false;

pthread_mutex_unlock(
       &mut_var);
```

Thread 2:

```
pthread_mutex_lock(
       &mut_var);

init(var);
canUseVar = true;

pthread_cond_broadcast(
       &cv_var);

pthread_mutex_unlock(
       &mut_var);
```

Conditional variables in Pthread

- The relevant procedures involving pthreads condition variables:
  - pthread_cond_init(pthread_cond_t *cv, NULL);
  - pthread_cond_destroy(pthread_cond_t *cv);
  - pthread_cond_wait(pthread_cond_t *cv,
    pthread_mutex_t *mutex);
  - pthread_cond_signal(pthread_cond_t *cv);
  - pthread_cond_broadcast(pthread_cond_t *cv)

- Condition variables must be declared with type pthread_cond_t, and must be initialized before they can be used
  - Statically, when it is declared. For example:
    `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
  - Dynamically

    ```
    pthread_cond_init(cond, attr);
    ```

    Upon successful initialization, the state of the condition variable becomes initialized.
- `pthread_cond_destroy(cond)` should be used to free a condition variable that is no longer needed

pthread_cond_wait

- `pthread_cond_wait(cv, mutex)` is called by a thread when it wants to be blocked and wait for a condition to be true
- It is assumed that the thread has locked the mutex indicated by the `mutex`, found that the condition does not occur, and therefore the thread needs to `wait`
- `wait` causes the thread to release the mutex, and blocks until awakened by a `pthread_cond_signal(cv)` call from another thread
- When it is awakened, it waits until it can acquire the mutex, and once acquired, it returns from the `pthread_cond_wait(cv, mutex)` call

pthread_cond_signal

- **pthread_cond_signal(cv)** checks to see if there are any threads waiting on the specified condition variable. If not, then it simply returns
- If there are threads waiting, then one is awakened
- There can be **no assumption** about the order in which threads are awakened by `pthread_cond_signal(cv)` calls
  - It is natural to assume that they will be awakened in the order in which they waited, but that may not be the case…
- Use loop or **pthread_cond_broadcast(cv)** to awake all waiting threads

Bounded Buffer Reader Using CV

- Data strctures:
  Queue<T> q
  pthread_mutex_t qM,
  pthread_cond_t qCV

- Reader Flow:
  pthread_mutex_lock (qM)
  If (q.empty())
  Wait (qCV, qM); //waiting for an element to be written
  Element e = q.pop(); //here we have both a lock and element
  Pthread_mutex_unlock (qM)
  ....

Barrier Example (1)


```
#define NTHREADS 5

pthread_mutex_t *n_done_mutex;

pthread_cond_t *barriar_cv;

int n_done = 0;

main() { // Checking of return values omitted for brevity
    pthread_t tids[NTHREADS];
    int i;
    void *retval;

    pthread_mutex_init(&n_done_mutex, NULL);
    pthread_cond_init(&barriar_cv, NULL);

    for (i = 0; i < NTHREADS; i++)
        pthread_create(&tids[i], NULL, barrier, (void *) &i);

    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);

    printf("done\n");
}
```

Barrier Example (2)


```
void *barrier(void *arg) {
    // Checking of return values omitted for brevity
    int* id = (int *) arg;
    printf("Thread %d -- waiting for barrier\n", id);
    pthread_mutex_lock(n_done_mutex);
    ndone = ndone + 1;
    if (ndone < NTHREADS) {
        pthread_cond_wait(barriar_cv, n_done_mutex);
    }
    else {
        pthread_cond_broadcast(barriar_cv);
    }
    pthread_mutex_unlock(n_done_mutex);
    printf("Thread %d -- after barrier\n", id);
}
```

Barrier example Output

Thread 0 -- waiting for barrier
Thread 2 -- waiting for barrier
Thread 1 -- waiting for barrier
Thread 3 -- waiting for barrier
Thread 4 -- waiting for barrier
Thread 4 -- after barrier
Thread 0 -- after barrier
Thread 1 -- after barrier
Thread 2 -- after barrier
Thread 3 -- after barrier
done

Semaphores

For allowing X threads to access a shared data structure

Semaphores

- Semaphore can be initialized to any value
- Can do increments and decrements of semaphore value
- Thread blocks if semaphore value is equal to zero when a decrement is attempted
- As soon as semaphore value is greater than zero, one of the blocked threads wakes up and continues (and decrements it to zero again)
  - no guarantees as to which thread this might be

Creating and Destroying Semaphores

- Semaphores are created like other variables

    sem_t semaphore;

- Semaphores must be initialized

    Prototype:

        int sem_init(sem_t *sem, int pshared, unsigned int value);

        sem: the semaphore value to initialize

        pshared: share semaphore across processes – usually 0

        value: the initial value of the semaphore

- Semaphores must be released when they are not needed:

    int sem_destroy(sme_t *sem);

- Both functions return negative number on failure

Decrementing a Semaphore

- Prototype:

  int sem_wait(sem_t *sem);

  - *sem:* semaphore to try and decrement

- If the semaphore value is greater than 0, the *sem_wait* call return immediately

  otherwise it blocks the calling thread until the value becomes greater than 0

Incrementing a Semaphore

- Prototype:

  int sem_post(sem_t *sem);

  sem: the semaphore to increment

- Increments the value of the semaphore by 1

  if any threads are blocked on the semaphore, one of them will be unblocked

Atomic Variables

Retrieving, updating and saving a variable is

Multi Thread Counting

- What will be the problem with the following code?

```c
int counter;
void *foo(void * arg){
    for (int i = 0; i < 1000; ++i) {
        counter += 1;
    }
}

int main(int argc, char** argv)
{
    pthread_t threads[5];
    for (int i = 0; i < 5; ++i) {
        pthread_create(threads + i, NULL, foo, NULL);
    }

    for (int i = 0; i < 5; ++i) {
        pthread_join(threads[i], NULL);
    }
    printf("counter: %d\n", counter);

    return 0;
}
```

std::atomic as a solution

- std::atomic was introduced in c++11 to simplify the access into shared (primitive) variables
- In std::atomic the compiler guarantees safe access to the variable
  - T load() → Returns the value of the atomic variable
  - T store (T new_val)
    - Update the atomic variable with a new val
    - Return the old value
  - Operator =
    - If the atomic is Left Hand side do store
    - If Right hand side load

Fixed Example


```
std::atomic<int> counter;
void *foo(void * arg){
    for (int i = 0; i < 1000; ++i) {
        counter += 1;
    }
}

int main(int argc, char** argv)
{
    pthread_t threads[5];
    for (int i = 0; i < 5; ++i) {
        pthread_create(threads + i, NULL, foo, NULL);
    }

    for (int i = 0; i < 5; ++i) {
        pthread_join(threads[i], NULL);
    }
    printf("counter: %d\n", counter.load());

    return 0;
}
```