University of Glasgow

Systems Programming – Part 2
Concurrent Systems Programming

Dr Lauritz Thamsen
lauritz.thamsen@glasgow.ac.uk
https://lauritzthamsen.org

University of Glasgow

Topics of Part 2 of SP(GA)

- Intro to Concurrency (with Processes and Threads)
- **Process/Thread Synchronisation**
- More on Process Management (from an OS Perspective)
- Concurrency Beyond Threads & Limits of Scalability
- Virtual Memory & Levels of Storage

Process/Thread Synchronisation

A little disclaimer regarding processes and threads here: The following will talk about cooperating processes (like much of the early literature on this and many OS textbooks, including the one we are using), while for practical programming in C, it makes sense to think about cooperating threads

University of Glasgow

Lecture Outline

- **Race Conditions and Critical Sections**
- Atomic Instructions, Mutexes, Semaphores
- Semaphore Examples
- Starvation, Livelock, Priority Inversion, Deadlock

Shared Memory Example

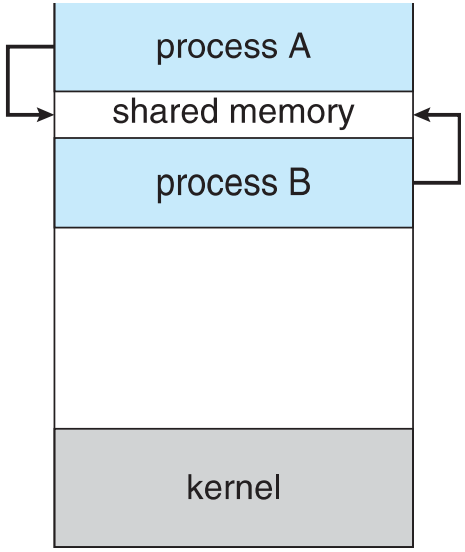
- Consider two processes P_A and P_B, communicating via shared memory
- Assume shared memory consists of a single integer X = 20

P_A

```
...
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
...
```

P_B

```
...
my_X = shared_mem.X
if (my_X >= 15)
   my_X = my_X - 15
shared_mem.X = my_X
...
```



- What is the final value of X?
  - P_A executes fully before P_B: P_A's check for X >= 10 succeeds → X = 10; P_B's: check for X >= 15 fails → X = 10
  - P_B executes fully before P_A: P_B's check for X >= 15 succeeds → X = 5; P_A's check for X >= 10 fails → X = 5
- But enter preemptive scheduling/parallel execution/…

Another Shared Memory Example

- Consider two threads T0 and T1 attempting to increment a `count` variable with a C compiler that implements `count++` as

```
register1 = count
register1 = register1 + 1
count = register1
```

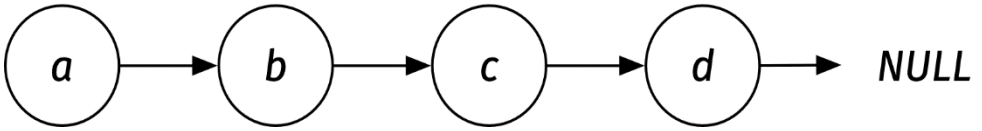
- Consider this execution interleaving with "count = 5" initially:

```
T0: register1 = count          {T0 register1 = 5}
T0: register1 = register1 + 1  {T0 register1 = 6}
T1: register1 = count          {T1 register1 = 5}
T1: register1 = register1 + 1  {T1 register1 = 6}
T1: count = register1          {count = 6}
T0: count = register1          {count = 6}
```

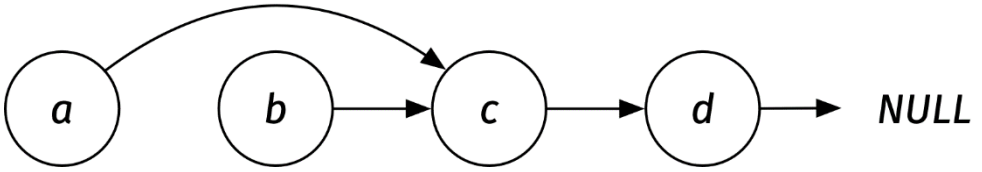
Q: Should the value of `count` be 6? What has gone wrong? How can we fix it?

And One More!

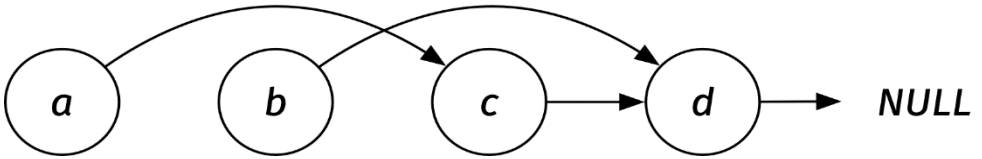
- Consider a singly linked list:



- Two threads simultaneously start removing the elements from the list.
- The first thread removes “b” by moving the next pointer of “a” to “c”:



- The second thread removes “c” by moving the next pointer of “b” to “d”:



- The resulting list is **inconsistent**, i.e. it still contains node “c” (the item deleted by the second thread)!
- We can avoid inconsistency by ensuring that simultaneous updates to the same part of the list cannot occur

University of Glasgow

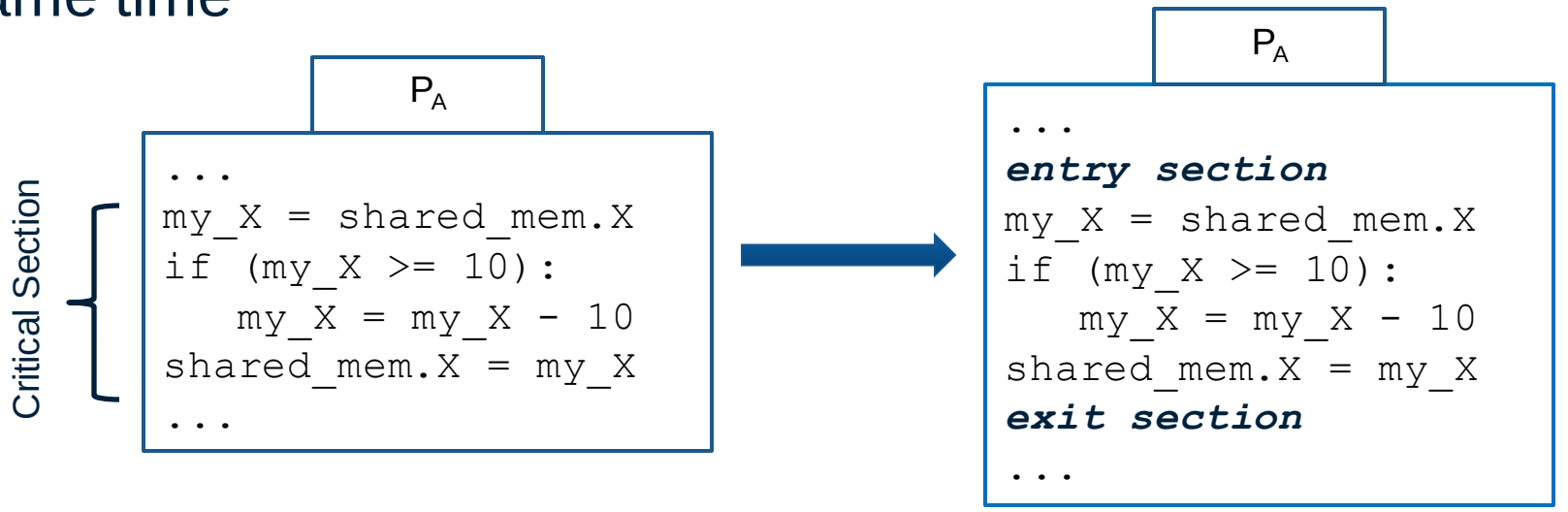
A Race Condition

- A **race condition** refers to a dependency on the execution sequence (or scheduling) of multiple processes/threads

- A race condition is a problem as soon as one or more of the possible outcomes is undesirable

Shared Memory Example Revisited

- How would you alleviate this problem?
- **Critical Section**: A section of code where a process accesses shared resources
  - Shared memory, variables, database, file, …
- Goal: No two processes should be in their critical section at the same time



University of Glasgow

Critical Section Solution Properties

- **Mutual exclusion**: No two processes are in their critical section at the same time

- **Progress**: Entering one's critical section should only be decided by its contenders and in due time (assuming no process already in its critical section); a process cannot immediately re-enter its critical section if other processes are waiting for their turn

- **Bounded waiting**: It should be impossible for a process to wait indefinitely to enter its critical section, if other processes are allowed to do so

Peterson's Solution

P_A

```
...
flag_A = True
turn = B
while flag_B == True and turn == B:
   pass
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
flag_A = False
...
```

P_B

```
...
flag_B = True
turn = A
while flag_A == True and turn == A:
   pass
my_X = shared_mem.X
if (my_X >= 10):
   my_X = my_X - 10
shared_mem.X = my_X
flag_B = False
...
```

- **Mutual exclusion**: If P_A is in its critical section, then either *flag_B = False* (i.e., P_B is out of its critical section) or turn = A (i.e., P_B is waiting in the while loop)
- **Progress**: P_A cannot immediately reenter its critical section, as turn = B means P_B will be given the go next
- **Bounded waiting**: P_A will wait at most one turn before it can enter its critical section again

University of Glasgow

Outline

- Race Conditions and Critical Sections
- **Atomic Instructions, Mutexes, Semaphores**
- Semaphore Examples
- Starvation, Livelock, Priority Inversion, Deadlock

Peterson's Solution

- Is that good enough?
  - **Busy waiting**: the waiting process eats up CPU cycles unnecessarily (a.k.a., spinlock)
  - **Memory reordering**: Modern CPUs tend to reorder the execution of memory accesses (to avoid pipeline stalls)
- Better solution: **atomic** instructions at the **hardware** level
  - "Test and set"
  - "Compare and swap"

value passed **by reference**

```
test_and_set(value):
    my_value = value
    value = True
    return my_value
```

```
compare_and_swap(value, expected, new_value):
    my_value = value
    if value == expected:
        value = new_value
    return my_value
```

Peterson's Solution Revisited

test_and_set

```
...
lock   while test_and_set(lock) == True:
           pass
       my_X = shared_mem.X
       if (my_X >= 10):
           my_X = my_X - 10
       shared_mem.X = my_X
unlock lock = False
       ...
```

```
test_and_set(value):
    my_value = value
    value = True
    return my_value
```

compare_and_swap

```
...
while compare_and_swap(lock, False, True) == True:   } lock
    pass
my_X = shared_mem.X
if (my_X >= 10):
    my_X = my_X - 10
shared_mem.X = my_X
lock = False                                          } unlock
...
```

```
compare_and_swap(value, expected, new_value):
    my_value = value
    if value == expected:
        value = new_value
    return my_value
```

- **Mutual Exclusion**: Only one process will execute *test_and_set* or *compare_and_swap* with the lock value originally being False
- **Progress/Bounded waiting**: There is nothing stopping a process from immediately re-entering its critical section!

Beyond test_and_set and compare_and_swap

- *Test_and_set* / *compare_and_swap* work, but are a bit clunky and low-level
- Alternative: **mutex locks** and **semaphores**
  - Internal state: a single integer value
    - For mutexes can only take values 0 or 1, for semaphores only values >= 0
  - API offers two **atomic** functions:

```
acquire(mutex):
    while mutex == 0:
        pass
    mutex = 0
```

```
release(mutex):
    mutex = 1
```

```
wait(semaphore):
    while semaphore <= 0:
        pass
    semaphore -= 1
```

```
signal(semaphore):
    semaphore += 1
```

- But still busy waiting/spinlocking?
- Can augment semaphores with a list of blocked processes each:
  - wait(semaphore) would instead add processes to said list if value is <= 0
  - signal(semaphore) would instead remove one process from said queue

University of Glasgow

Semaphore Usage

- Set semaphore value to *number* or *size* of shared resources (i.e., number of acceptable concurrent users of the resource)
  - 1 if only 1 user is allowed, N for an N-sized queue, etc.
- Decrease (wait) the semaphore every time a resource is used
- Increase (signal) the semaphore every time a resource is released

```
wait(semaphore):
    while semaphore <= 0:
        pass
    semaphore -= 1
```

```
signal(semaphore):
    semaphore += 1
```

University of Glasgow

Outline

- Race Conditions and Critical Sections
- Atomic Instructions, Mutexes, Semaphores
- **Semaphore Examples**
- Starvation, Livelock, Priority Inversion, Deadlock

Bounded Buffer Problem

- Also known as the producer-consumer problem
- Assume a list where items are placed by producers, and removed by consumers
- Assume we want our list to never contain more than N items
- Goal: Allow producers of items to add them to the list, but have them wait first if the list is full
- Goal: Allow consumers of items to remove an item from the list, but have them wait first if the list is empty

```
semaphore mutex = 1
semaphore empty = N
semaphore full  = 0
```

```
producer():
    while True:
        # Produce an item
        wait(empty)
        wait(mutex)
        # Add item to list
        signal(mutex)
        signal(full)
```

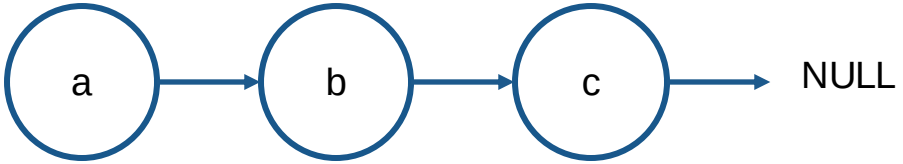
```
consumer():
    while True:
        wait(full)
        wait(mutex)
        # Remove an item from list
        signal(mutex)
        signal(empty)
        # Do stuff with item
```

Reading hint:

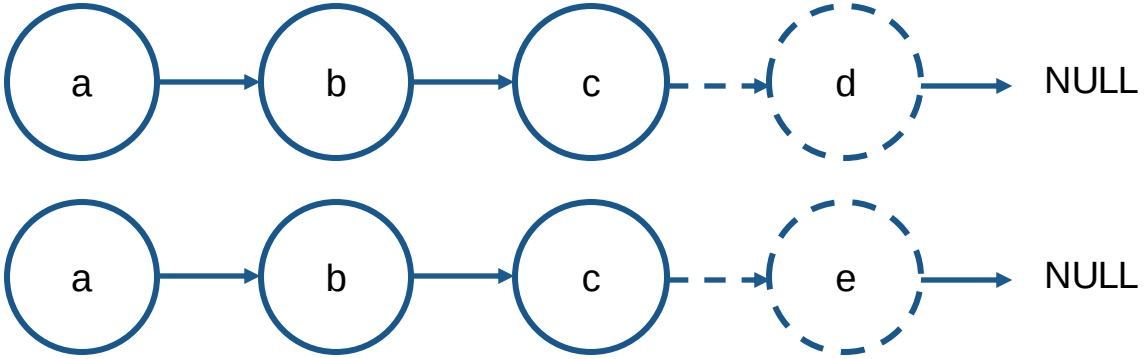
- `empty`: list is still a bit empty
- `full`: list is still at least a bit full
- Starts *fully* empty, can become *fully* full (`N` transferred from `empty` to `full`)

Why Protect the List?

- Consider a singly linked list

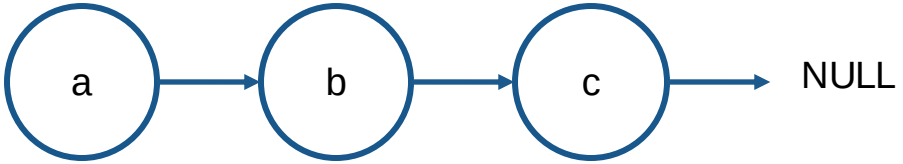


- What happens if two processes **simultaneously add an element**?

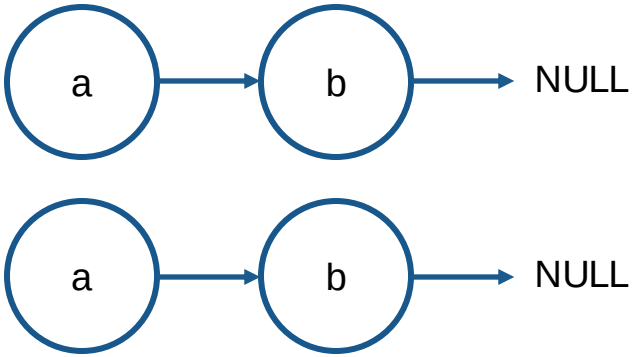


… also for Removing an Element

- Consider a singly linked list

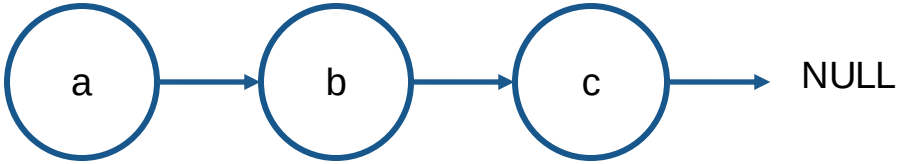


- What happens if two processes **simultaneously remove the last element**?



… and Producing and Consuming?

- Consider a singly linked list



- What happens if two processes **simultaneously produce and consume**?

Readers-Writers Problem

- Assume a variable shared among many processes, some of which only read its value (readers) while others also need to update it (writers)
- Goal 1: When a writer accesses the variable, no other process should be able to either read or update it
- Goal 2: When a reader accesses the variable, more readers can also access it, but writers should wait until no reader accesses it

```
semaphore rw_mutex = 1
semaphore mutex    = 1
int read_count     = 0
```

```
writer():
   while True:
      wait(rw_mutex)
      # Update the value
      signal(rw_mutex)
```

```
reader():
   while True:
      wait(mutex)
      read_count += 1
      if (read_count == 1)
         wait(rw_mutex)
      signal(mutex)
      # Read value
      wait(mutex)
      read_count -= 1
      if (read_count == 0)
         signal(rw_mutex)
      signal(mutex)
```

University of Glasgow

Outline

- Race Conditions and Critical Sections
- Atomic Instructions, Mutexes, Semaphores
- Semaphore Examples
- **Starvation, Livelock, Priority Inversion, Deadlock**

University of Glasgow

Matters of Life and Death

- What would happen to writers if readers keep on arriving at the system?
  - Writers would “starve”
- **Starvation**: Processes/threads unable to enter their critical section because of “greedy” contenders
  - Analogy: trying to get on an extremely busy motorway with no one giving you some space
- **Livelock**: a special case of starvation where competing parties both try to “avoid” each other at the same time
  - Analogy: bumping into a person in a corridor, then both going left/right at the same time only to bump into each other again
- **Priority inversion**: a special case of starvation where a lower priority process keeps a higher priority process waiting
  - Analogy: having to sleep but being kept awake by social media notifications…

Matters of Life and Death

- What would happen in the producer/consumer problem if order of locking/unlocking mutex and empty/full was reversed?

```
semaphore mutex = 1
semaphore empty = N
semaphore full  = 0
```

```
producer():
    while True:
        # Produce an item
        wait(mutex)
        wait(empty)
        # Add item to list
        signal(full)
        signal(mutex)
```

```
consumer():
    while True:
        wait(mutex)
        wait(full)
        # Remove an item from list
        signal(empty)
        signal(mutex)
        # Do stuff with item
```

- Assume a consumer goes first…
  - *mutex* → locked; consumer waiting on *full*
  - producer waiting on *mutex*
- **Deadlock**: all parties of a group waiting indefinitely for another party (incl. themselves) to take action
  - e.g. each of two threads waiting for a resource held by the other thread

University of Glasgow

Beyond Semaphores…

- Semaphores/mutexes allow for mutual exclusion, but once a process is blocked, that's it!
- **Monitors**: A combination of **semaphores** and **condition variables**
  - Each condition variable "associated" with a semaphore
  - Allows for processes to have both mutual exclusion and wait (block) on a condition
  - *cond_wait(condvar, mutex)*: unlock the mutex to wait for a condition, then atomically reacquire the mutex when condition is met
  - *cond_signal(condvar)*: unblock one of the processes waiting on condition

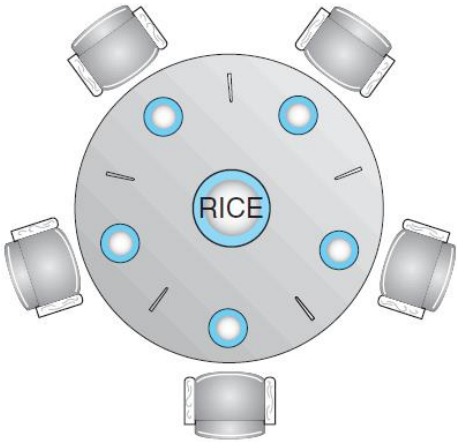
```
semaphore mutex = 1
condvar   empty = N
condvar   full  = 0
list      items = 0
```

```
producer():
   while True:
      # Produce an item
      wait(mutex)
      while len(items) == N:
         cond_wait(empty, mutex)
      # Add item to list
      cond_signal(full)
      signal(mutex)
```

```
consumer():
   while True:
      wait(mutex)
      while len(items) == 0:
         cond_wait(full, mutex)
      # Remove an item from list
      cond_signal(empty)
      signal(mutex)
      # Do stuff with item
```

Food for Thought: Dining Philosphers

- Philosophers sitting on a round table, plates in front of them, but only as many chopsticks as there are philosophers (as in the image)
- Philosophers spend the most time thinking, but now and then get hungry
- They need two chopsticks to eat and will not grab a chopstick from their neighbour's hand
- Philosophers can only pick up one chopstick at a time
- When done eating, they will put the chopsticks back on the table



Source: A. Silberschatz, "Operating System Concepts", 9th Ed., 2012.

Using a Mutex Lock in C

```
pthread_mutex_t m;
bool teaIsReady = false;

void *me(void* arg) {
  (void)arg;
  printf("Me: Waiting for my tea ...\n");
  pthread_mutex_lock(&m);
  while (!teaIsReady) {
    pthread_mutex_unlock(&m);
    printf("Me: (Unamused) // do nothing\n");
    pthread_mutex_lock(&m);
  }
  pthread_mutex_unlock(&m);
  printf("Me: (Happy) ... finished waiting.\n");
  return NULL;
}
```

```
void *teaRobot(void* arg) {
  (void) arg;
  printf(" Tea Robot: Making tea ...\n");
  usleep(3);
  pthread_mutex_lock(&m);
  teaIsReady = true;
  pthread_mutex_unlock(&m);
  printf(" Tea Robot: Done!\n");
  return NULL;
}
```

```
int main() {
  pthread_t t1; pthread_t t2; int err;
  err = pthread_mutex_init(&m, NULL); assert(err == 0);
  err = pthread_create(&t1, NULL, me, NULL); assert(err == 0);
  err = pthread_create(&t2, NULL, teaRobot, NULL); assert(err == 0);
  err = pthread_join(t1, NULL); assert(err == 0);
  err = pthread_join(t2, NULL); assert(err == 0);
  pthread_mutex_destroy(&m);
}
```

University of Glasgow

Recommended Reading

- Silberschatz, Galvin, Gagne, Operating Systems Concepts, Sections 5.1-7