University of Glasgow | School of Computing Science

# Networks & Operating Systems Essentials

Dr Angelos Marnerides

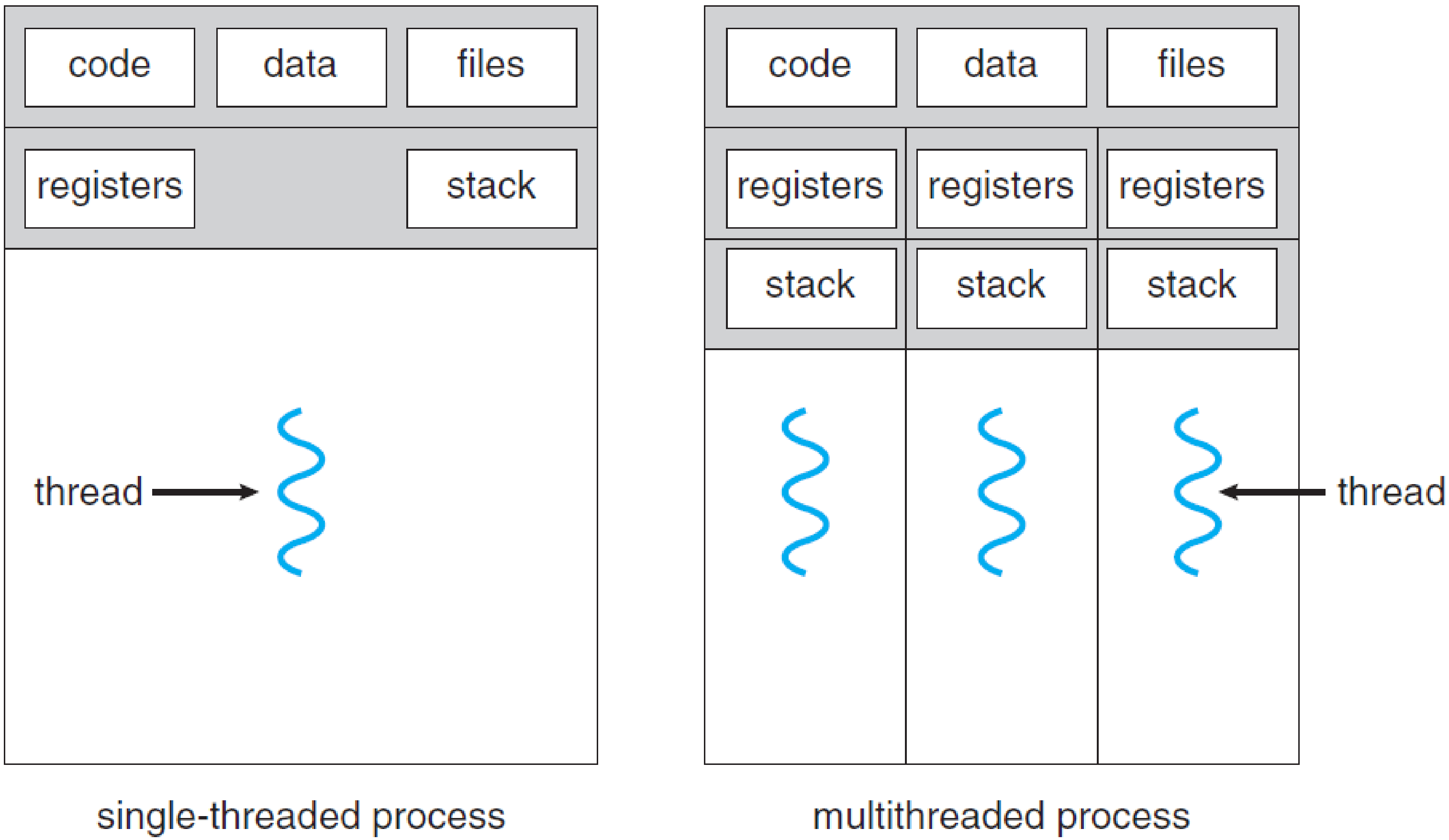
<*angelos.marnerides@glasgow.ac.uk*>

School of Computing Science

# Concurrency vs Parallelism

- Parallelism: multiple tasks (threads/processes) execute at the same time
  - Requires multiple execution units (CPUs, CPU cores, CPU vCores, etc.)
- Concurrency: multiple tasks make progress over time
  - Time-sharing system

# Threads




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

# Threads vs Processes

- Threads share memory and resources by default
  - Processes require extra IPC mechanisms (shared memory, message passing) to be configured explicitly
- Threads are faster/more economical to create
  - Each process has its own copy of the in-memory data, hence process creation means page table duplication (possibly also data duplication, if copy-on-write not used)
- In older operating systems, it was much faster to context switch between threads than processes
  - No longer the case, as the kernel keeps the same info for both, hence context switching has almost identical cost
- Can have different schedulers for processes and threads
  - Depends on how threads are implemented (user threads vs kernel threads)
- If a single thread in a process crashes, the whole process crashes as well
  - If a process dies, other processes are unaffected

# Aside: Amdahl's Law

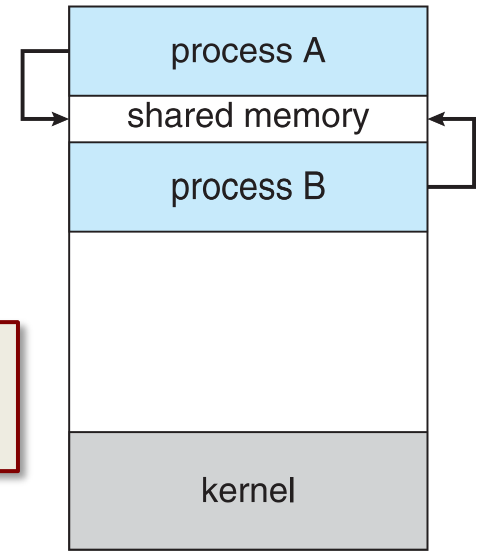
- Assume you have a program, code such that:
  - Part of the code can only be executed serially (*S*)
  - Part of the code can be parallelised across *N* cores
- Then the maximum speed-up we can gain by going parallel is given by:

$$speedup = \frac{1}{\left(S + \frac{1-S}{N}\right)}$$

- Note: if *N* approaches infinity, then *speedup = 1/S*

# A Banking Quiz…

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




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

$P_A$

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

$P_B$

```
   my_X = my_X - 15
shared_mem.X = my_X
...
```

A Race Condition!

- 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 time-sharing/preemptive scheduling/parallel execution/…

# A Banking Quiz…

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

$P_A$

Critical Section

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

$P_A$

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

# The Critical Section problem

- Properties of acceptable solutions:
  1. Mutual Exclusion: No two processes in their critical section at the same time
  2. Progress: Entering one’s critical section should only be decided by its contenders 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
  3. 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

# Peterson's solution

- Is that good enough?
  - Busy waiting: the waiting process eats up CPU cycles unnecessarily (a.k.a., spinlock)
  - Memory reordering: CPUs tend to reorder execution of mem 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

```
...
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
lock = False
...
```

lock

unlock

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

compare_and_swap

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

lock

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`/`compare_and_swap` with the lock value originally being `False`
- Progress/Bounded waiting: There's nothing stopping a process from immediately re-entering its critical section!
- Extra: More elaborate/better solutions exist, but not covered here.

# Beyond test_and_set/compare_and_swap

- Test_and_set/compare_and_swap work but are a bit clunky and a bit too low-level
- Enter *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
- Common understanding today: use *mutex* for locking and *semaphores* for signaling

# Discussion: atomic ops vs spinlocking

- Atomic operations provide a good solution to inter-process/thread synchronization
- But how does one do an atomic op in hardware?
  - Disable interrupts while atomic function is running
- How do you do that in a multi-processing system?
  - Good luck with that…
- What then?

# Semaphores how to

- Set semaphore value to *number/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

# Case study: 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
```

# Case study: 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: When a writer accesses the variable, no other process should be able to either read or update it
- Goal: 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)
```

# Matters of life and death…

- What would happen with 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
  – Think: trying to get on an extremely busy motorway with no one giving you some space
- Livelock: special case of starvation where competing parties both try to “avoid” each other at the same time
  – Think: 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: special case where lower priority processes can keep higher priority processes waiting
  – Think: 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

# Beyond semaphores…

- Semaphores/mutexes allow for mutual exclusion, but once a process is blocked that's it
- Enter Monitors
  - 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

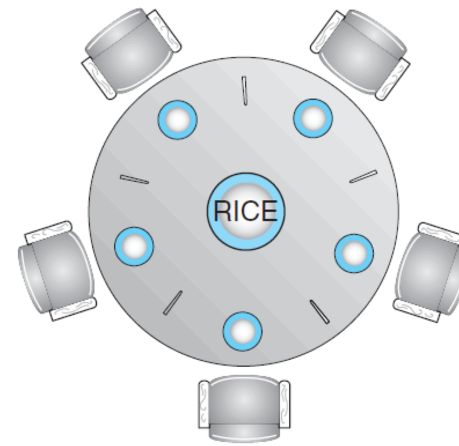
```
semaphore mutex = 1
condvar   empty = N
condvar   full  = 0
list      items = 0
```

```
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 philosophers

- Philosophers sitting on a round table, plate in front of them, but only as many chopsticks as there are philosophers (see image)
- Philosophers spend most time thinking, but every now and then get hungry…
- One needs two chopsticks to eat and won't grab a chopstick from their neighbour's hand
- Philosophers can only pick up one chopstick at a time
- When done eating, they'll put the chopsticks back on the table




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

# Recommended Reading

- Silberschatz et. al., ”Operating Systems Essentials”, Chapter 6, sections 6.3,6.4, 6.5