CS2106
Introduction to OS

Office Hours, Recess Week

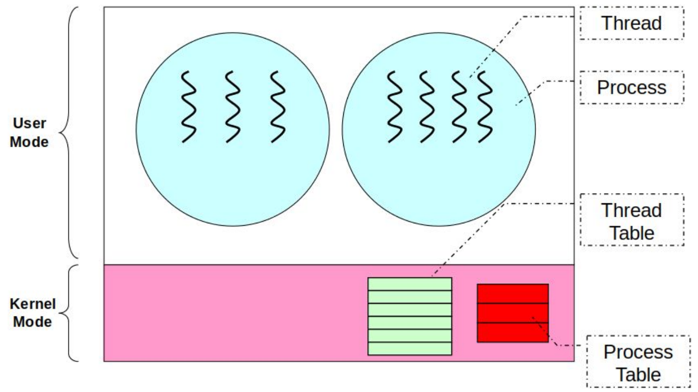
Agenda

- Threads
  - Kernel/User Threads
  - Threads vs Processes
- Synchronization
  - Peterson’s Algorithm
  - Critical Section Properties
  - Test and Set
  - Atomic Instructions
  - Busy Waiting

Kernel Threads

In the kernel threading model, all scheduling is done on threads instead of process? What is the point of the *process* table in this case?

- Excellent question.
- Processes and threads are different entities and therefore have different metadata to store.
  - For process, e.g., process's status, signal, and size information, as well as per-process data that is shared by the kernel threads.
  - For threads, e.g., scheduling, priority, state, CPU usage information of a kernel thread.



User Threads

If the OS is not aware of user threads, where is the stack and register information for each thread stored?

- Great question.
- Basically the program that manages these user threads must manage that information.
  - Most likely stored on the heap of the program itself

- If you search for "coroutines" or "green threads", you can see programming languages that implement this user threading concept, and more info on how they do it.

Threads vs processes

Why can't we use the thread model for process switch? Just keeping the hardware context different and the rest shared?

- For security reasons we want isolation between processes

- Threads share most of their data, so there's much less security/isolation between threads than processes.

Threads vs processes

Is there a point in having threads on a single-core? Isn't it similar to process switching?

- Switching between threads is quite cheap compared to process switching
- Trade-offs: less security/isolation between threads, but better performance.
- Do we need multi-threaded programs on a single-core machine?
  - Absolutely!
  - when a single-threaded process blocks on I/O, it cannot make progress
  - in multi-threaded programs, while one thread is blocked on I/O, another one can do something else.

Threads vs processes

How are updates on the stack done for threads? Let's say the stack looks like this: Thread X is from position 1 to 2. Thread B is position 3 to 4.

1. a = 1

2. b = 2

3. R = 1

4. Z = 2

Now we want to store something on the stack for thread X. How do we insert this into position 3? Isn't that where Thread B's stack is?

- The OS usually reserves a large enough space for each thread's stack and makes sure they don't overlap.
  - OS can leave some space in between the stacks
- If a thread uses too much stack space they'll get a stack-overflow error.

Threads

Are the general purpose registers shared between 2 threads of the same purpose?

- The purpose of the process doesn't necessarily matter, perhaps this is asking whether threads under the same process share the general purpose registers.
- There's only one set of general purpose registers (within the same CPU core & assuming no hyperthreading).
- So threads share the same GPR **hardware**, but they have their own GPR values that are private to themselves.
- When we switch between threads, the OS has to make sure the right GPR values are loaded.

Peterson’s Algorithm

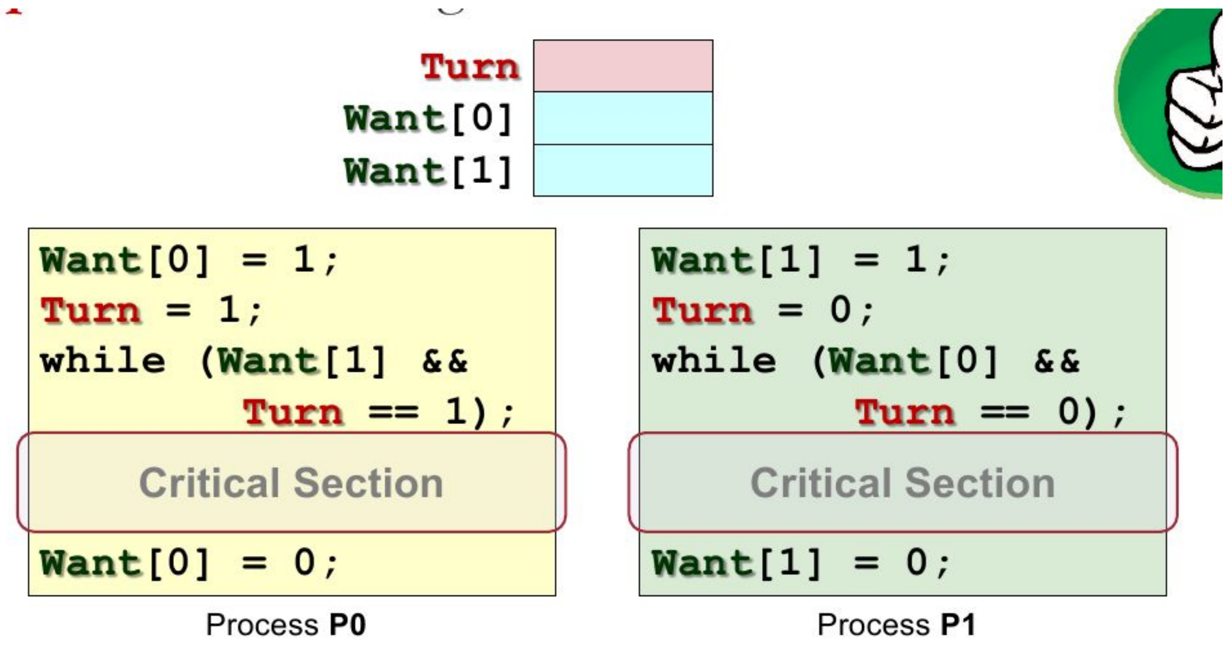
What does Peterson's not being general mean? Could you elaborate more? I thought that by letting every process go thru P.A, all processes will be synchronized due to them waiting on each other and taking turns...

- A version exists for N processes, but it's complicated.
- Especially: guaranteeing fairness and other properties.
- Also, may actually not work on modern hardware due to something that's called "relaxed consistency model of memory", but that's way too advanced to talk about.

Peterson's Algorithm

Is there a need to switch the "Turn variable" back to another process's index in Peterson's Algorithm after a process is done using the CS?

- No, and it wouldn't change anything
- Variable turn only matters as a tie-breaker when both processes want to enter CS
- Say P0 sets turn=1 after CS
  - it doesn't matter to P1 because want [0] == 0



```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
            Turn == 1);
   Critical Section
Want[0] = 0;
```

Process **P0**

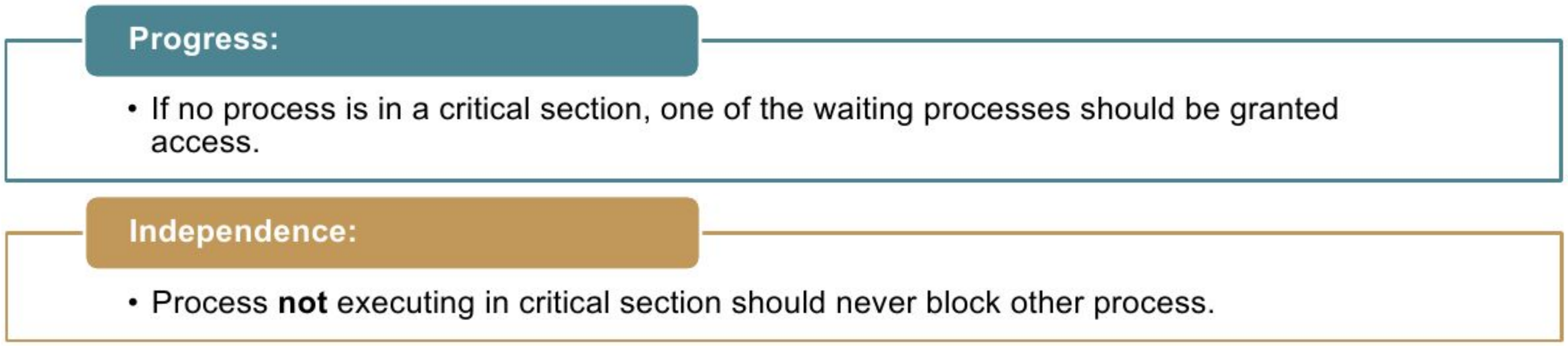
```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
            Turn == 0);
   Critical Section
Want[1] = 0;
```

Process **P1**

Critical Section Properties

Regarding the 4 properties, what's the difference between Progress and Independence? They seem quite similar. Attempt 3 shows that it is possible to have independence but not progress. Is it possible to have progress but not independence?

- Excellent question and observations.
- Progress and independence **are** related (the *OS Concepts* book treats them as one)
- It is not possible to have progress but no independence (vice versa can)
  - because an independent process may block others and prevent their progress



Progress:

- If no process is in a critical section, one of the waiting processes should be granted access.

Independence:

- Process **not** executing in critical section should never block other process.

Livelocks vs. Deadlocks

What are the differences & similarities between deadlocks and livelocks?

- **Deadlocked** processes are in BLOCKED state
  - No way to get out of it, as none of them is running
- **Livelocked** processes are busily doing something. Two types:

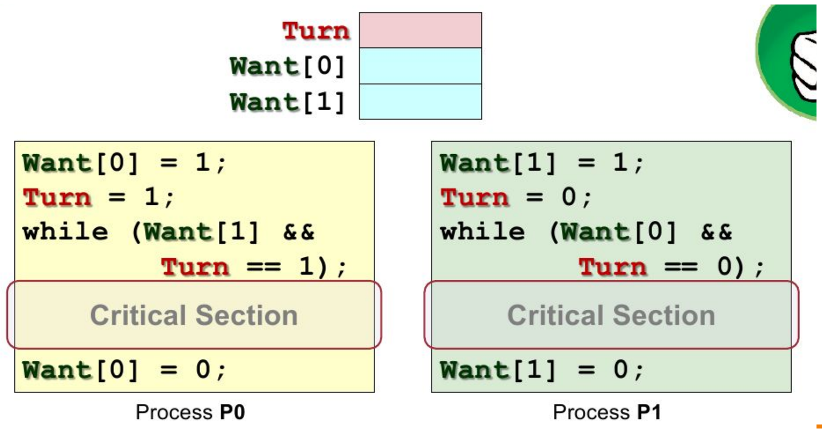
  1. They may be able to get out of the livelock by chance
    - Think of two people bypassing each other eventually

  2. Or there may absolutely never be able to get out of a livelock
    - Think of HLL attempt 3
- **Note**: The livelock situation where it's guaranteed that the processes will never progress (type 2) are also called deadlocks by some OS books.
  - just a naming difference

Critical Section Properties

What happens if a process hangs in the middle of a critical section. What should happen?

- Excellent question.
- Even in “correct” Peterson’s algorithm, P0 crashing during the CS will cause livelock for P1.
- **Hard to know what we (the CS implementer) “should” do!**
  - Critical sections are critical because the operations inside must be fully executed or not (atomic) by one process.
  - Cannot know in all cases whether we should allow other processes in or not.



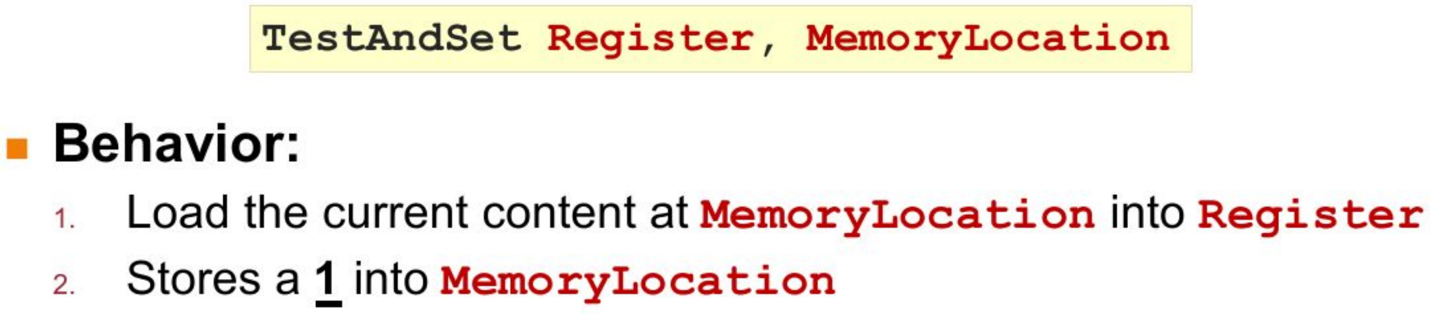
```
Want[0] = 1;
Turn = 1;
while (Want[1] &&
        Turn == 1);
Critical Section
Want[0] = 0;
```

```
Want[1] = 1;
Turn = 0;
while (Want[0] &&
        Turn == 0);
Critical Section
Want[1] = 0;
```

Test and Set

For test and set, is the purpose of "loading current content at memory location into register" so that the variable in the register can be used for while loops? (To break the while loops...)

- The purpose of reading the variable is to see **whether or not you read value 0** (unlocked).
- After you do test_and_set(), the value of the variable must always be 1 (locked), but you may not be the one who locked it.
- **You will be the one who locked it only if you get zero.** So, you either do something else and try again later, or loop until you get return value zero.



```
TestAndSet Register, MemoryLocation
```

- **Behavior:**
  1. Load the current content at `MemoryLocation` into `Register`
  2. Stores a 1 into `MemoryLocation`

- **returns the old value at MemoryLocation!**

Atomic Instructions

If an instruction is made to do a very long calculation, will that instruction still be considered atomic, because it might take up some time/computation or might even break it up etc...?

What's atomic depends on the context

- On a single-core processor, every instruction is atomic, because it cannot be interrupted (interrupts are checked for ***after*** every instruction).
  - It may take many cycles, but it’s functionally atomic (indivisible)
- Some instructions are intrinsically atomic on any machine
  - E.g., reading the value of a a properly aligned integer variable into a register
- But two "atomic" instructions on two different cores can well interfere with each other and behave non-atomically.
  - E.g., **`INC <variable_in_memory>`** to increment a variable in memory
  - The real challenge in implementing instructions like `test-and-set` is to ensure that they are indeed atomic in a multi-core system.

Busy Waiting

What is busy-waiting?

- Def. Using CPU cycles to wait for something to happen e.g.,

```
    while (var_shared_by_threads);
```

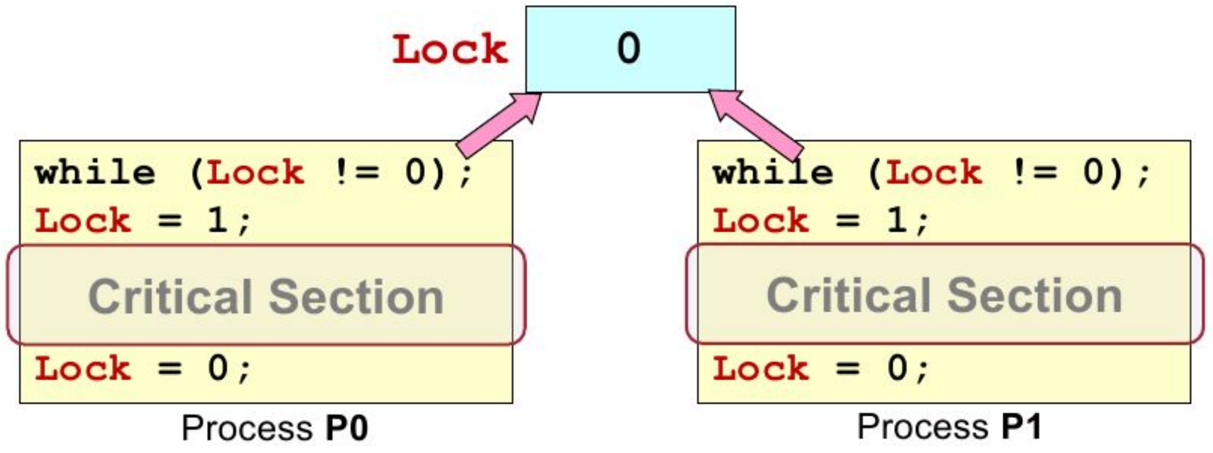
  is called *busy-waiting*
  - typically reading a shared variable and waiting for a particular value

- If critical sections are long and many threads are busy-waiting, this time can be substantial
  - Many CPU cycles are wasted without much progress

Busy Waiting

Why does HLL attempt 1 cause busy waiting?

- While P0 is in the critical section, P1 will be executing millions of jump and branch/test instructions, checking if Lock != 0.
- This is busy waiting.



```
while (Lock != 0);
Lock = 1;
    Critical Section
Lock = 0;
```

Thank you!