# CS2106
# Introduction to OS

Office Hours, Week 7

## Agenda

- Most interesting questions from last week
  - Plus some unanswered questions
- Pthreads
  - For lab3

## Livelocks and Deadlocks

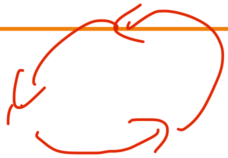
while ( condition ≠ True ) {

}

**Is livelock the same as busy waiting?**

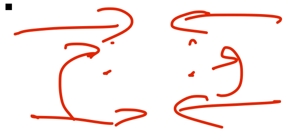
- No
- Busy waiting: looping in a while loop and using the CPU time waiting for something to happen
  - Does not need to involve multiple processes
  - e.g., you can busy wait while waiting for some data to arrive
- Livelock: situation when 2+ processes cannot make progress, although they are doing something
  - They could be busy waiting or executing some code that doesn't lead to progress

# 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
2. Or there may absolutely never be able to get out of a livelock
   - Think of HLL attempt 3

want[0]=1;
while (want[1]) { }

want[1]=1
while (want[0]) { }

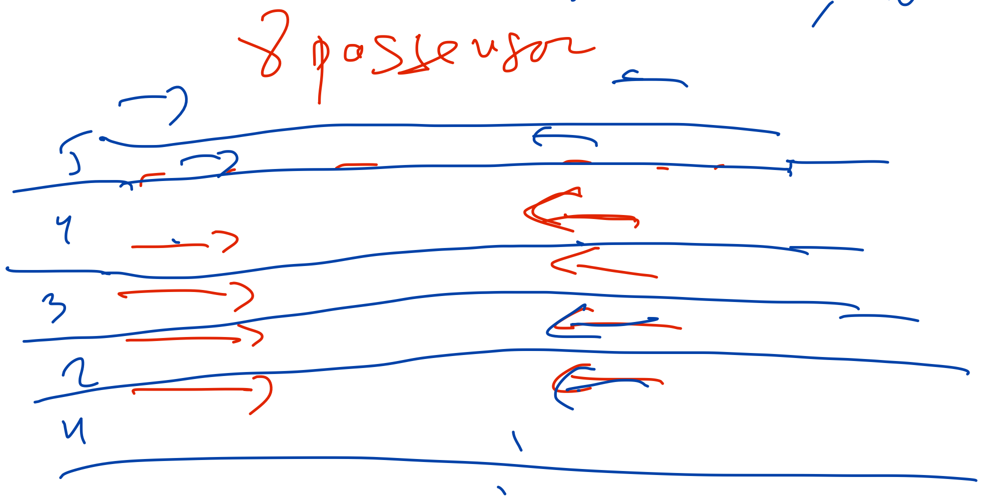
- **Note**: The livelock situation where it's guaranteed that the processes will never progress (type 2) are also called deadlocks by some OS books.

## Livelocks

**Is the probability of livelock happening proportional to the number of processes?**

- Can't be answered generally
- For example, if a livelock (in some context) happens only when **all** processes do something at the same time, then the probability of a livelock actually decreases with the number of processes

2N processes, N+1 lanes



## Semaphores

**Why we allow more than 1 process in the CS in general semaphore? It wouldn't solve the problem of race conditions.**

- Once we allow multiple processes into CS, it is not a critical section anymore.
  - It wouldn't solve the problem of data races
- But the concept of allowing a fixed number of processes to execute a code section can have many uses
  - e.g., the safe distancing problem in restaurants
  - many other synchronization problems (we will see next week)

wait( )

$\}$

signal( )

## Semaphores

lock()

CS ✓

unlock()

**How are binary semaphores different from locks?**

- Binary semaphores are one implementation of a lock.
- Note: It is always better to use a specialized type of binary semaphores if they exist on your system, rather than using a general semaphore initialized to 1
- This special semaphore is called mutex.
- Example:

```
Semaphore S=1;                 10
P0:                                        P1:
S.signal(); //S==2                         S.wait(); // S==1
S.wait();   //S==1                                  S=0
    //here both P0 and P1 can enter CS
```

In contrast, the value of a mutex is always capped at 1.

## Semaphores

**Can you explain the use of semaphores as general synchronization tool again? Which of the codes are critical sections?**

- Semaphores can be used in many problems beyond critical section
- The general mechanism described in class doesn’t have a CS
- We just want to ensure that execution of procedure B waits until execution of procedure A is done
  - For whatever reason
  - Perhaps, some dependency of B on the result of A

## Semaphore as General Synchronization Tool

- Execute *B* in $P_1$ **only after** *A* executed in $P_o$
- Use semaphore *s* initialized to 0
- Code:

```
P0                P1
 .                 .
 .                 .
 .                 .
 A               wait(s)
signal(s)          B
 .                 .
```

P

s = 1

*Imagine that A produces something and B consumes what A produces*

## Miscellaneous/Advanced

a++ a--

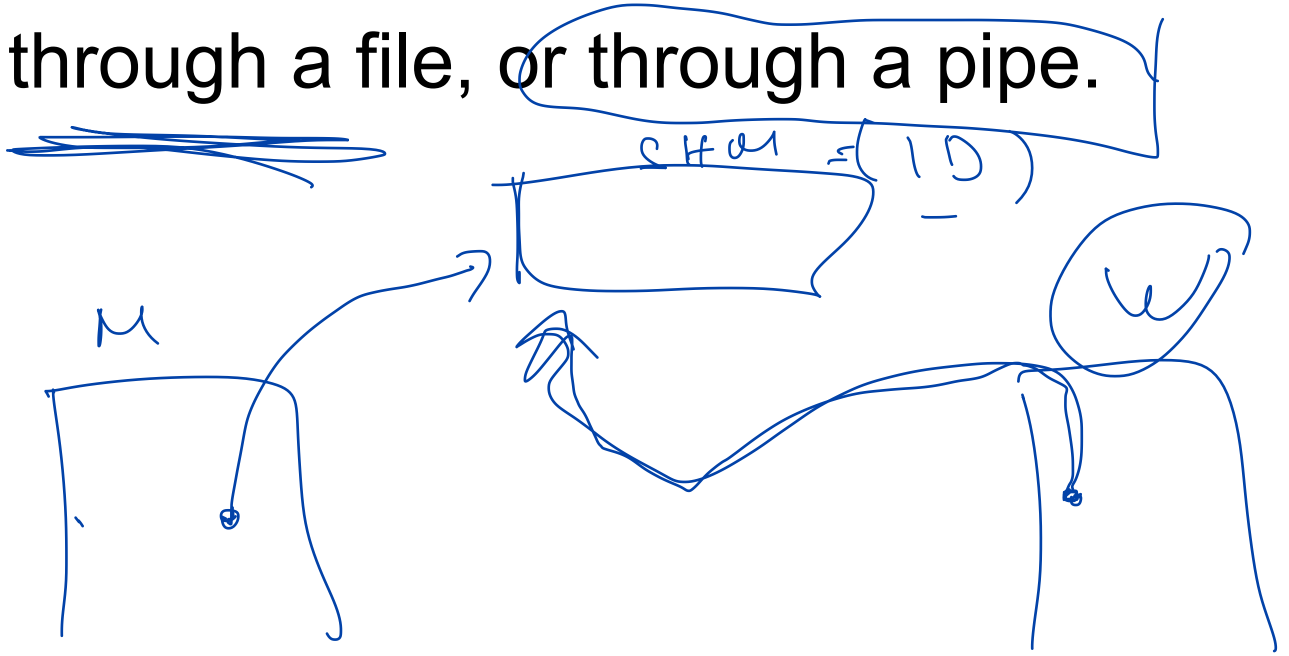
**Can allowing of interleaving be done using machine learning? In the beginning the system only allows one, and then by trial and error allows more, this would exponentially decrease in time and only need to be done once.**

- For a running program:
  - No! You can't do trial-and-error testing when the program is already running.
- As a help in development of concurrent programs:
  - ML is a bad approach to learn where to put synchronization primitives.
  - Program synthesis methods are more suitable.

## Miscellaneous

In the shared memory pseudocode, how does the master tell the worker about the shmid?

- Great question
- Alternative method of communication is needed
- E.g., through a file, or through a pipe.



## Miscellaneous

**If I put a pointer inside the shared memory, will it dereference into the same value on all processes?**

- No!
- The same pointer in two different processes have different meaning
  - And generally point to different physical locations
  - Even the shared memory region may be attached at different addresses within each process
  - This will become clear in Week 10
- Excellent question

## Miscellaneous

**What is meant by "No amount of buffering helps when the sender is always faster than the receiver"**

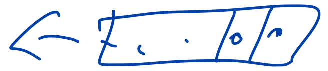
- Asynchronous message passing is often used in real-time systems
- In such systems, every task is executed periodically
- If the sender task frequency is higher than receiver task frequency, the buffer will always overflow regardless of its capacity

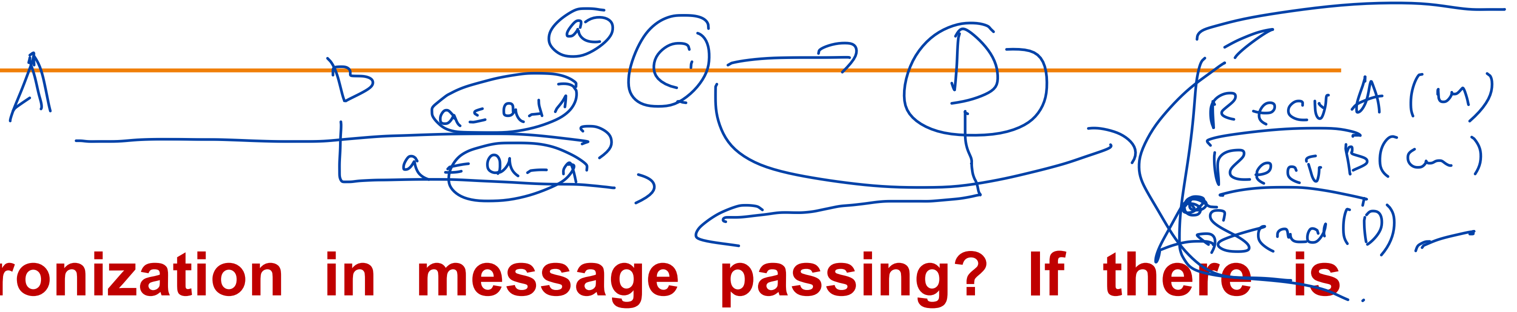
every sec.

```
For (   ) {
    .
    :
    send Msg ( )
    sleep ( )
}
```

every 2 sec.

```
For ( ) {
    :
    :
    receive Msg ( )
    sleep ( )
}
```





**Why don't we need synchronization in message passing? If there is process A, send 'a += 1' to process C, process B send 'a -= 1' to process C. Process D reads value from process C. Depending on the order of A and B, we'll get diff value in process D?**

a
a+1 -1
a+2
a

- Great question
- Note that in case of shared memory, **a += 1** and **a -= 1** can interleave and make mess
- In case of MP, the integrity of the messages is guaranteed by the OS
  - Regardless of the order in which the two messages are sent
  - The **final** content of variable **a** is always the same (example, can't be a-1)
  - No synchronization is needed to guarantee atomic update of **a**.
- For proc. D to read **a** from C, C must send a message
  - C needs to send the message after receiving from both A and B

# POSIX Threads

An example thread API

## POSIX Threads: $\boxed{\textit{pthread}}$

- Standard defined by the IEEE
  - Supported by most Unix variants
- Defines the API as well as the behavior (specification)
  - However, **implementation** is not specified
  - So, ***pthread*** can be implemented as user / kernel thread
- Will show a few example to highlight the differences between process and thread only

## Basics of Pthread

- Header File:
  - `#include <pthread.h>`
- Compilation:
  - `gcc XXXX.c -lpthread`

    -l xyz
- Useful datatypes:
  - `pthread_t`: Data type to represent a thread object
  - `pthread_attr`: Data type to represents attributes of a thread

## Pthread Creation

- Thread creation syntax:

```
int pthread_create(
        pthread_t   *tidCreated,
        const pthread_attr_t *threadAttributes,
        void* (*startRoutine) (void*),
        void *argForStartRoutine  );
```

- Returns:
  - `0` to indicate success; `!0` for errors
- Parameters:
  - **`tidCreated:`** Thread Id for the created thread
  - **`threadAttributes:`** Control the behavior of the new thread
  - **`startRoutine:`** Function pointer to the function to be executed by thread
  - **`argForStartRoutine:`** Arguments for the `startRoutine` function

## Pthread Termination

exit(20);

- Thread termination syntax:

NULL

```
void pthread_exit( void* exitValue );
```

- Parameters:
  - `exitValue:` Value to be returned to whoever synchronize with this thread (more later)

- If `pthread_exit()` is not used, a *pthread* will terminate automatically when the end of the **`startRoutine`** is reached
  - In this case, there is no way to return an exit value

## Pthread Creation & Termination: Example

```
//header files not shown

void* sayHello( void* arg )
{
     printf("Just to say hello!\n");
     pthread_exit( NULL );
}

int main()
{
    pthread_t tid;

    pthread_create( &tid, NULL, sayHello, NULL );
    printf("Thread created with tid %i\n", tid);

    return 0;
}
```

[illegible] exit(0).

# Pthread: Sharing of memory space

```
//header files not shown
int globalVar;

void* doSum( void* arg)
{     int i;
      for (i = 0; i < 1000; i++)
            globalVar++;
}

int main()
    pthread_t tid[5];     //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );
    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

- What is the sum that we get (or should get)?

## Pthread: Simple Synchronization

- To wait for the termination of another pthread:

```
int pthread_join( pthread_t threadID,
                  void ** status);
```

pthread exit(--)

- Returns:
  - 0 to indicate success; `!0` for errors
- Parameters:
  - `threadID:` **TID** of the pthread to wait for
  - `status:` Exit value returned by the target pthread

## Pthread: Sharing of memory space V2.0

```
//header files not shown
int globalVar;

void* doSum( void* arg)
{ //same as before }

int main()
    pthread_t tid[5];     //5 threads id
    int i;
    for (i = 0; i < 5; i++)
        pthread_create( &tid[i], NULL, doSum, NULL );

    //Wait for all threads to finish
    for (i = 0; i < 5; i++)
        pthread_join( tid[i], NULL );

    printf("Global variable is %i\n", globalVar);
    return 0;
}
```

## Pthread: A lot more!

- There are more interesting stuff about ***pthread***:
  - Yielding (giving up CPU voluntarily)
  - Advanced synchronization — Semaphores & pthreads
  - Scheduling policies
  - Binding to kernel threads, CPU cores etc.

- As we cover new topics, you can explore the ***pthread*** library to see the application!

- Linux Pthread Man Pages
  - "`man pthread`……" for more

Thank you!