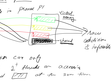
- By different Algorithm
- Race condition
  - ↳ shared memory (critical section)
  - ↳ Critical memory (locking)
- Sharing Global Memory page
  - ↳ sharing access processes
- Threads → output status
  - ↳ locking of process
- wished memory section at at the process → lock
- threads when the core process → share memory

Race

Critical Section Problem

multiple threads in process P1



Race condition can only occur if 2 threads or occurs for shared [...] at the same time.

Thread 1:

do {

entry section

Critical section → race condition can only occur in this part

remainder section

} while (true);

---

t1	t2
do { entry section 1	do { entry section 2
Critical Section 1	critical section 2
remainder section 1	remainder section 2
} while (true);	} while (true);

OS

(a race condition can only occur if more than 1 thread is creating its critical section) at any given time

① prevent the threads from being in the C.S. at the same time!

↳ mutual exclusion.

How to enforce mutual exclusion? (mutual exclusion)

↳ entry lock after when threads are!! X

↳ taps X (lock)

↳ engage only one thread currently in a time. (only one thread in process) → figure

No thread is starved

↳ assign that t1 executes the CS → 1000 times

t2 executes CS

t1 ex CS → 1 billion times

t2 exec CS

↳ there is a time to for any thread enter into CS after t1 has repeated on CS → bounded waiting

any solution to the Critical Section problem must satisfy

① Mutual Exclusion

② Progress

③ Bounded waiting

Peterson's Lit

t1	t2
$x_0, x_1, x_2 \ldots$	$y_0, y_1, y_2, \ldots$
$x_0 \neq$	$y_0 \neq$
$flag = T$	$flag = F$
$turn = 1$	$turn = 2$
do { entry section	do { entry
$flag[1] = T$	$flag[2] = T$
$turn = 2$	$turn = 1$
while ($flag[2]$ && $turn = 2$);	while ($flag[1]$ && $turn = 1$);
Critical section	Critical section
$x_0 \neq$	$y_0 \neq$
flag[1] = F	flag[2] = F
remainder	remainder
$x_0, x_1, x_2 \ldots$	$y_0, y_1, y_2 \ldots$
} while (true);	} while (true);

Stop

first C into region

in C region

see region into C

Hold

---

t1	t2
$x_0, x_1, x_2 \ldots$	$y_0, y_1, y_2, \ldots$
$x_0 \neq$	$y_0 \neq$
$flag[1] = T$	$flag[2] = T$
$turn = 2$	$turn = 1$
while ($flag[2]$ && $turn = 2$);	while ($flag[1]$ && $turn = 1$);
Critical section	Critical section
$x_0 \neq$	$y_0 \neq$
flag[1] = F	flag[2] = F
remainder	remainder
$x_0, x_1, x_2 \ldots$	$y_0, y_1, y_2, \ldots$
} while (true);	} while (true);

if

$turn = (i+1)\%n$

while ($flag[i=1]$ && $turn = i$);

flag[1] → indicate if thread 1 to enter

turn = 1 ... n

mutual exclusion

bounded waiting

progress

---

acquire & release lock

↳ provided by OS

and these are atomic.

Solution → MutexX

↳ acquire_lock()

Critical section

↳ release_lock()

→ Mutex

MutexX has a drawback

↳ busy waiting

↳ spin lock

acquire_lock() {

while (condition);

}

2 secs

get unlocked

t1

t2

CS

---

deadlock

P.C.P.

int main() {

while (true);

return;

}

↳ CPU → 100%

Semaphores

↳ counting Semaphore

Mutex → Binary Semaphore

Semaphore → lock & signal

class Semaphore {

int mutex_count = 1;

flag = T;

public: void wait() {

while (mutex_count <= 0);

if (mutex_count > 0)

→ mutex_count = mutex_count - 1;

→ cout << "wait" << mutex_count << endl;

→ cout << "wait" << flag << endl;

}

→ }

public: void signal() {

flag = true;

→ mutex_count = mutex_count + 1;

→ if (mutex_count <= 0)

→ if mutex_count > 0

→ for (int i = 0; i < n; i++)

→ if (flag[i] == true)

→ break;

}

}

sharing things you are from one thing

sharing → resource

CPU usage → 1

→ pause → 0

output status → R.C. status

any 2 at

a.c.

Ω - tree

Ω^{Ω} - tree

generalized form of a B.S.T

within a Larger fan-factor

---

binary search

→ get min()

→ get max()

→ get count ()

→ get ()

→ get

→ get_[]

→ traverse()

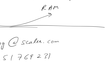
→ B.S.T

Search only with → B.S.T

+ balancing

	B.S.T	B.S.T	B.S.T
Avg	$\log n$	$\log n$	$\log n$
Worst	n	n	n

TC (find()

worst case → height of tree or worst case

TC (find() → B.S.T