Process Synchronization

Indian Statistical Institute

https://www.isical.ac.in/~mandar/courses.html#os

Cooperating processes

Reference: Section 4.4

Cooperating process: shares data with other processes

Independent process: does not share data with other processes

Means of cooperation:

- Synchronization
- Communication

race-condition.c

- Parent and child process execute the following

```c
for (i = 0; i < NUM; i++) {
    fscanf(fp, "%d", &count);
    rewind(fp);
    fprintf(stderr, "Value read by %d: %d\n", pid, count);
    sleep(1);

    (pid) ? count++ : count-- ; // pid == 0 for child

    fprintf(fp, "%d\n", count);
    rewind(fp);
    fprintf(stderr, "Value written by %d: %d\n", pid, count);
    sleep(1);
}
```

- What number does the file contain finally?

Producer-consumer problem

Problem: a *producer* process generates output that is used by a *consumer* process as input

Examples:

- print program (producer) + printer driver (consumer)
- `$ ls -l | more`

Implementation:

- Producer and consumer access a shared buffer
- Concurrent access is allowed
- Consumer must wait if buffer is empty
- **Unbounded buffer:** no limit on size of buffer
- **Bounded buffer:** buffer is of fixed size
    - producer must wait if buffer is full

Race condition

Reference: Section 6.1

producer

```
p = produce();
while (count==n); // full
buffer[in] = p;
in = in+1 % n;
count++;
```

consumer

```
while (count==0); // empty
p = buffer[out];
out = out+1 % n;
count--;
consume(p);
```

```
                                    MOV count R0 /* R0 = 5 */
MOV count R0 /* R0 = 5 */
ADD #1 R0    /* R0 = 6 */
MOV R0 count /* count=6 */
                                    SUB #1 R0    /* R0 = 4 */
                                    MOV R0 count /* count=4 */
```

Race condition

Reference: Section 6.1

producer

```
p = produce();
while (count==n); // full
buffer[in] = p;
in = in+1 % n;
count++;
```

consumer

```
while (count==0); // empty
p = buffer[out];
out = out+1 % n;
count--;
consume(p);
```

---

producer	consumer
	`MOV count R0 /* R0 = 5 */`
`MOV count R0 /* R0 = 5 */`	
`ADD #1 R0     /* R0 = 6 */`	
`MOV R0 count /* count=6 */`	
	`SUB #1 R0     /* R0 = 4 */`
	`MOV R0 count /* count=4 */`

Race condition: several processes manipulate the same data concurrently s.t. outcome depends on the order in which the processes execute

Critical section problem (CSP)

Reference: Section 6.2

Critical section (CS): segment of code in which processes access shared data

Synchronization scheme:

```
while (1) {
   entry_section();
   critical_section();
   exit_section();
   remainder_section();
}
```

CSP solutions

Desiderata:

- Mutual exclusion: At most one process may execute code from CS at any given time
- Progress: If no process is executing in CS and some processes want to enter CS, only processes not in the remainder section can participate in deciding which process next enters CS. Selection of a process cannot be postponed indefinitely.
- Bounded waiting: there exists a bound on # of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Assumptions:

- Basic machine instructions (load, store, test) are executed atomically

Two-process solutions

Algorithm 1: strict alternation

```
shared int turn = 0;

P_i()
{  while (1) {
      while (turn != i);  // wait
      critical_section();
      turn = j;  // j = 1-i
      remainder_section();
   }
}
```

- Mutual exclusion – satisfied
- Progress – not satisfied

Two-process solutions

Algorithm 2:

```
shared char want[2] = {0,0};

P_i()
{  while (1) {
      want[i] = 1;
      while (want[j]);
      critical_section();
      want[i] = 0;
      remainder_section();
   }
}
```

- Mutual exclusion – satisfied
- Progress – not satisfied

Two-process solutions

Algorithm 3:

```
shared char want[2] = {0,0};
shared int turn = 0;

P_i()
{  while (1) {
      want[i] = 1;
      turn = j;
      while (want[j] && turn!=i);
      critical_section();
      want[i] = 0;
      remainder_section();
    }
}
```

- Mutual exclusion, progress, bounded waiting – satisfied

Two-process solutions

Incorrect solutions:

```
shared char want[2] = {0,0};
shared int turn = 0;

1. P_i()
2. { while (1) {
3.     turn = j;                          want[i] = 1;
4.     want[i] = 1;                       while (want[j] && turn!=i);
5.     while (want[j] && turn!=i); OR     critical_section();
6.     critical_section();                want[i] = 0;
7.     want[i] = 0;                       turn = j;
8.     remainder_section();               remainder_section();
9.   }
10.}
```

P_0: (3) $\rightarrow$ P_1: (3–5) $\rightarrow$ P_0: (4–5)

Multiple-process solution

(Lamport's) bakery algorithm

- Each process wanting to enter CS gets a token number
- Process with lowest token number enters CS
- If two processes have same token number, process with lower PID enters CS

Bakery algorithm

```
shared char choosing[N] = {0, ..., 0};
shared int number[N] = {0, ..., 0};

do {
   choosing[i] = 1; // why??
   number[i] = MAX(number[0], number[1], ..., number[n-1]) + 1;
   choosing[i] = 0; // why??
   for (j = 0; j < n; j++) {
       while (choosing[j]) ;
       while ((number[j] != 0) && (number[j],j < number[i],i)) ;
   }
   critical_section();
   number[i] = 0;
   remainder_section
} while (1);
```

Hardware solutions

Reference: Section 6.3

Disabling interrupts

- Critical section executes without preemption
- Adopted by non-preemptive Unix kernels

Atomic test-and-set

- TestAndSet primitive:

```
char TestAndSet(char *flag)
{ char rv = *flag; *flag = 1; return rv; }
```

- CSP solution:

```
shared char lock = 0;
...
while (TestAndSet(&lock));  critical_section();  lock = 0;
...
```

Hardware solutions

Atomic swap

- Swaps the contents of two words atomically
- CSP solution:

```
shared char lock = 0;
...
key = 1;
do Swap(lock,key) while (key != 0);
critical_section();
lock = 0;
```

Hardware solutions

n-process CSP:

```
shared char lock, waiting[N];
...
waiting[i] = 1;
key = 1;
while (waiting[i] && key) { key = TestAndSet(lock); }
waiting[i] = 0;

critical_section();

j = i+1 % n;
while (j!=i and waiting[j]==0) { j = j+1 % n; }
if (j==i) then lock = 0;
else waiting[j] = 0;
```

Semaphores

Reference: Section 6.4

Definition: a counting semaphore S is an integer variable that can be accessed only through two atomic operations, $wait(S)$ (or $P(S)$) and $signal(S)$ (or $V(S)$)

$wait(S)$: `while (S <= 0); S--;`
$signal(S)$: `S++;`

n-process CSP:

```
shared semaphore mutex = 1;
P_i()
{ wait(mutex); critical_section(); signal(mutex); ... }
```

Semaphores

Synchronization problems

- Example:
  P_1, P_2 are 2 concurrent processes
  Statement S_2 in P_2 should be executed after S_1 in P_1

```
shared semaphore synch = 0;
P1:  ...  S1; signal(synch); ...
P2:  ...  wait(synch); S2; ...
```

- Careless use may lead to deadlock

```
P1:  wait(S1); wait(S2); ...  signal(S1); signal(S2);
P2:  wait(S2); wait(S1); ...  signal(S2); signal(S1);
```

Busy waiting vs. blocking

Spinlocks/busy waiting

- Processes execute a loop while waiting for entry to critical section $\Rightarrow$ waste of CPU cycles
- Useful in multiprocessor systems if locks are held for short intervals ($\because$ context switch can be avoided)

Blocking

```
typedef struct {int value; struct process *L;} semaphore;
```

$wait(S)$:

```
S.value--;
if (S.value < 0) {
  add process to S.L;
  sleep();
}
```

$signal(S)$:

```
S.value++;
if (S.value <= 0) {
  remove process P from S.L;
  wakeup(P);
}
```

Implementation issues

- For bounded waiting, `L` should be maintained as a FIFO queue
- $wait(S)$, $signal(S)$ must be atomic
  $\Rightarrow$ two processes cannot be in $wait/signal$ simultaneously
  $\Rightarrow$ $wait$ / $signal$ must be implemented as CS (!!)
  - use hardware solutions (disable interrupt, etc.), OR
  - software solutions
    - busy waiting is limited to the CS in $wait$ / $signal$ only
    - CS in $wait/signal$ is short ($\sim$10 instructions) and occupied for very short periods of time
      (CS in application programs may be long and almost always occupied)

Binary semaphores

Definition: semaphore whose integer value can be either 0 or 1

$wait_b(S)$:

```
if (S.value == 0) {
  add process to S.L;
  sleep();
}
else S.value = 0;
```

$signal_b(S)$:

```
if (S.value == 0) {
  P = dequeue(S.L);
  if (P == NULL) S.value = 1;
  else wakeup(P);
}
```

Binary semaphores

Implementing counting semaphores:

```
binary_semaphore S1 = 1, S2 = 0; int C = m;
```

$wait(S)$:

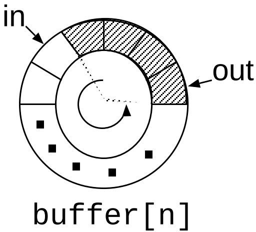
```
wait_b(S3); // why?
wait_b(S1);
C--;
if (C < 0) {
  signal_b(S1);
  wait_b(S2);
}
else signal_b(S1);
signal_b(S3);
```

$signal(S)$:

```
wait_b(S1);
C++;
if (C <= 0) signal_b(S2);
signal_b(S1);
```

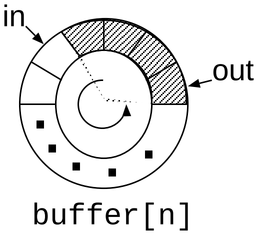
Producer-consumer implementation

- Initially, `in = out = 0`
- Empty queue: `in == out`
  full queue: `in+1 % n == out`
- Buffer can hold at most n-1 items



Producer-consumer implementation

- Initially, `in = out = 0`
- Empty queue: `in == out`
  full queue: `in+1 % n == out`
- Buffer can hold at most n-1 items



producer

```
p = produce();
while (in+1 % n == out);
/* buffer full, skip */
buffer[in] = p;
in = in+1 % n;
```

consumer

```
while (in == out); // skip
p = buffer[out];
out = out+1 % n;
consume(p);
```

Producer-consumer problem

Reference: Section 6.5.1

```
semaphore full = 0, empty = n, mutex = 1;
```

producer

```
while (1) {
  p = produce();
  wait(empty);
  wait(mutex);
  AddToBuffer(p);
  signal(mutex);
  signal(full);
}
```

consumer

```
while (1) {
  wait(full);
  wait(mutex);
  p = RemoveFromBuffer();
  signal(mutex);
  signal(empty);
  consume(p);
}
```

Readers-writers problem

Reference: Section 6.5.2

- Shared object is accessed by several concurrent processes
  - **Reader:** processes that only read the shared object
  - **Writer:** processes that update (read + write) the shared object
- Synchronization constraint: two or more readers can access shared data simultaneously; any writer must have exclusive access
- Variants:
  - *first* readers-writers problem: readers have (non-preemptive) priority
  - *second* readers-writers problem: writers have (non-preemptive) priority

Readers-writers problem

```
semaphore mutex = 1, write_access = 1, readcount = 0;
```

reader

```
wait(mutex);
readcount++;
if (readcount == 1)
   wait(write_access);
signal(mutex);

read();

wait(mutex);
readcount--;
if (readcount == 0)
   signal(write_access);
signal(mutex);
```

writer

```
wait(write_access);
write();
signal(write_access);
```

Readers-writers problem: preventing starvation I

Reference: `http://www.cs.umd.edu/~hollings/cs412/s96/synch/synch1.html`

- Keep track of *active* and *waiting* readers and writers

  `shared int nr_active, nr_waiting, nw_active, nw_waiting;`

  `shared semaphore lock = 1;` (to protect the above shared variables)

  `shared semaphore r_sem = 0, w_sem = 0, data = 1;`

- **Entry section:**

```
wait( lock );
 if ( nw_active + nw_waiting == 0 )
  {
     nr_active++;
     signal( r_sem );  // signal in advance to prevent waiting
  }
else
  nr_waiting++;
signal( lock );
wait( r_sem );
```

Readers-writers problem: preventing starvation II

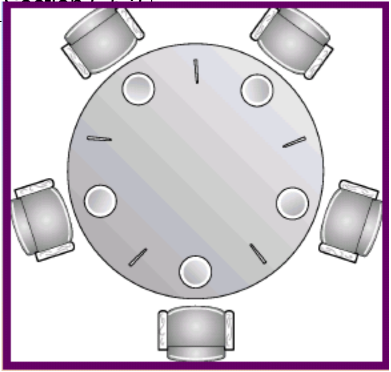
READING...

- **Exit section:**

```
wait( lock );
nr_active--;
if (( nr_active == 0 ) && ( nw_waiting > 0 )) // last reader
  {
     while ( nw_waiting > 0 )  // waiting writers enter
     {
        signal( w_sem );
        w_active++;
        w_waiting--;
     }
   }
signal( lock );
```

Dining philosophers problem

Reference: Section 6.5.[illegible]



```
philosopher()
{ while(1) {
     think();
     get_chopsticks();
     eat();
     release_chopsticks();
  }
}
```

- Only the nearest chopsticks can be used
- Only free chopsticks can be used
- Chopsticks must be acquired one by one

Dining philosophers problem

```
semaphore chopstick[5] = {1,1,1,1,1};

while (1) {
  think();
  wait(chopstick[i]);
  wait(chopstick[(i+1) % 5]);
  eat();
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
}
```

Deadlock avoidance:

- Philosophers can pick up chopsticks only if both are available, OR
- Odd philosphers pick up left chopstick first;
  even philosophers pick up right chopstick first.

Summary

- Manipulation of shared data $\Rightarrow$ race conditions
- Shared data should be accessed within critical sections
- Solutions to CSP:
  - Algorithm 3 (slide 9), Bakery algorithm
    - assume only atomic loads, stores, tests
  - Hardware solutions
    - assume atomic TestAndSet or Swap instructions
  - Semaphores
    - semaphores use one of the above solutions internally to ensure mutually exclusive access to semaphore variable
    - may be implemented as system calls which block if necessary
- Example synchronization problems: producer-consumer, reader-writer, dining philosopher
  - abstractions of problems that occur in real systems