إدارة العمليات
Process Management

د. فادي تركاوي

الرسائل العابرة Message Passing

- أنواع تمرير الرسائل بين العمليات:
  - الإرسال القابل للحظر Blocking Send
    وفيه تقوم المعالجة بالإرسال و الإنتظار حتى يتم التأكد من وصول الرسالة للمستقبلة
  - الإرسال غير القابل للحظر Non-Blocking Send
    وفيه تقوم المعالجة بالإرسال بشكل دائم دون التأكد من وصول الرسالة للمعالجة المستقبلة

الإتصال المباشر و غير المباشر Direct and Indirect Communication

- الإتصال المباشر و فيه تقوم العملية المرسلة بتحديد العملية التي يتم الإرسال لها وفي هذا النوع من الإرسال يكون التراسل بين معالجتين فقط
  - send(*destination-process*, *message*)
  - receive(*source-process*, *message*)
- الإتصال الغير مباشر و فيه يتم الإتصال بين المعالجات المتعددة من خلال صندوق التراسل mailbox وفي هذا النوع من الإتصال يمكن أن يتم التراسل بين معالجات متعددة

التخزين المؤقت Buffering

- الإتصال بين المعالَجات تتم من خلال عدة تقنيات حسب العطيات المفروضة هذه التقنيات هي:
  - السعة صفر Zero Capacity هنا المرسل يقوم بالإنتظار إلى أن يتم إستلام الرسالة
  - السعة المحدودة Bounded Capacity وهنا يوجد سعة محددة يقوم فيها المرسل بتخزين رسائله و يقوم المستقبل بإستقبال هذه الرسائل بالتوازي وفي حال تم تعبئة السعة المحددة يقوم المرسل بالإنتظار إلى حين أستقبال المستقبل بعض الرسائل
  - السعة الغير محددوة Unbounded Capacity وهنا يقوم المرسل بالإرسال بشكل دائم بغض النظر عن فيما إذا قام المستقبل بالإستقبال أم لا

الإستدعاء للعمليات البعيدة (كإتصال بين عمليتين)
Remote Procedure Call

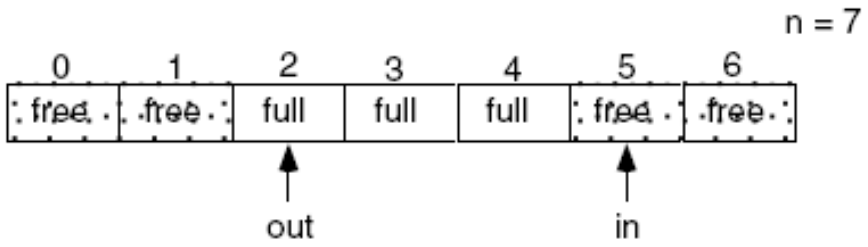
- يخفي عمليات الدخل و الخرج عن المستخدم
- يبدو و كأنه عملية داخلية ضمن المعالجة
- المرسل يشكل في أغلب الإحيان الزبون
- و المستقبل يشكل المخدم
- هذا الموديل من أنظمة الإتصال بين العمليات يشكل الجزء الأكبر للأنظمة الشبكية الحالية

الخيط التنفيذي النسيب Thread

- وهي عبارة عن معالجة صغيرة واحدة أو متعددة ضمن المعالجة الإساسية و تختلف عن المعالجة في أنها وحدة معالجة فقط و ليس وحدة مالكة للمصادر كمصادر الدخل و الخرج أو الذاكرة و الخيط يتم تنفيذة بالتوازي مع المعالجة الأم
- والخيط أو النسب يشترك مع الخيوط الأخرى بمجال العنونة و الكود البرمجي، Heap، المتغيرات العامة و مصادر الدخل و الخرج.
- الخيط البرمجي يملك: المسجلات، PC، مكدس و مؤشر المكدس
- الخيط البرمجي يستخدم بشكل أكبر في المخدمات وهو يوفر عملية الإتصال بين المعالجات المتعددة

Producer Consumer مشكلة المنتج المستهلك Problem

```
var buffer:  array[0..n-1] of items;     /* circular array */
in = 0
out = 0
```




```
/* producer */
repeat forever
        …
        produce item nextp
        …
        while (in+1 mod n == out)
              do nothing
        buffer[in] = nextp
        in = in+1 mod n
end repeat
```

```
/* consumer */
repeat forever
        while (in == out)
              do nothing
        nextc = buffer[out]
        out = out+1 mod n
        …
        consume item nextc
        …
end repeat
```

Atomicty الإفراد بالوصول

“Look in fridge, no milk” through “Put milk in fridge”
يجب أن تكون هذه العملية atomic

Time	You	Your Roommate
3:00	Arrive home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge, no milk
3:25	Buy milk, leave	Leave for grocery
3:30		
3:35	Arrive home	Arrive at grocery
3:36	Put milk in fridge	
3:40		Buy milk, leave
3:45		
3:50		Arrive home
3:51		Put milk in fridge
3:51	Oh, no! ***Too much milk!!***	

حلول المنطقة الحرجة Critical Section Solution

- أي حل للمنطقة الحرجة يجب أن يكون محقق للشروط الآتية
  - الإستبعاد المتبادل Mutual Exclusion و هذا يعني أنه إذا كانت معالجة في المنطقة الحرجة فهذا يعني أنه لن تستطيع أي عملية أو معالجة الدخول لهذه المنطقة
  - التقدم وهذا يعني أنه إذا كانت لا توجد أي عملية في المنطقة الحرجة و أنه إذا أرادت عملية الدخول للمنطقة الحرجة فيجب أن لا تؤجل بشكل غير محدد عن الدخول للمنطقة الحرجة

التقنيات المستخدمة لحل مشكلة المنطقة الحرجة

- Peterson Algorithm و هو معتمد على تقنية الإنتظار المشغول Busy Waiting
- Bakery Algorithm
- التقنيات المعتمدة على البنية الصلبة Hardware
- Semaphor وهو تقنية مزودة من قبل نظام التشغيل لتأمين حل المنطقة الحرجة مع التخلص من ال Busy Waiting
- Monitors وهي تقنية برمجية لحل مشكلة المنطقة الحرجة

Peterson`s Algorithm

- ***Peterson`s Algorithm***
  - ***وهو عبارة عن ألغوريثم بسيط لكي نقوم بتأمين الإستبعاد المتبادل Mutual Exlusion لمصدر من مصادر النظام بين عمليتين فقط***
- **من ميزات هذا الألغوريثم**
  - لا يوجد حاجة لإستخدام تقنيات البينة الصلبة للإستبعاد المتبادل
  - تحتوي على الإنتظار المشغول Busy Waiting

Peterson`s Algorithm

- المتحولات المستخدمة

  var flag: array [x,y] of boolean 1 , 0

  turn: 0 or1

  flag[k] means that process[k] is interested in the critical section

- القيم البدائية للمتحولات

  flag[0] := FALSE

  flag[1] := FALSE

  turn := random(0..1)

Peterson`s Algorithm

For Process i:

repeat

flag[i] := TRUE

turn := j

while (flag[j] and turn=j) {do no-op}

CRITICAL SECTION

flag[i] := FALSE

REMAINDER SECTION

until FALSE

مثال لتقنية بيترسون

Process 0	Process 1
flag[0] = TRUE turn = 1 check (flag[1] == TRUE and turn == 1) {do nothing} - Condition is false because flag[1] = FALSE - Since condition is false, no waiting in while loop - **Enter the critical section** - Process 0 happens to lose the processor	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) {do nothing} - Since condition is true, it keeps busy waiting until it loses the processor
- Process 0 resumes and continues until it finishes in the critical section - Leave critical section flag[0] := FALSE - Start executing the remainder (anything else a process does besides using the critical section) - Process 0 happens to lose the processor	
	check (flag[0] = TRUE and turn = 0) - This condition fails because flag[0] = FALSE - No more busy waiting - **Enter the critical section**

مثال لتقنية بيترسون

EXAMPLE 2	
Process 0	Process 1
flag[0] = TRUE turn = 1 - Lose processor here	
	flag[1] := TRUE turn := 0 check (flag[0] = TRUE and turn = 0) - Condition is true so Process 1 busy waits until it loses the processor
check (flag[1] == TRUE and turn == 1) {do-no-op} - This condition is false because turn = 0 - No waiting in loop - Enters critical section	

مثال لتقنية بيترسون

EXAMPLE 3	
Process 0	Process 1
flag[0] = TRUE - Lose processor here	
	flag[1] = TRUE turn = 0 check (flag[0] = TRUE and turn = 0) - Condition is true so, Process 1 busy waits until it loses the processor
turn := 1 check (flag[1] = TRUE and turn = 1) - Condition is true so Process 0 busy waits until it loses the processor	
	check (flag[0] = TRUE and turn = 0) - The condition is false so, Process 1 enters the critical section

تقنيات البنية الصلبة لحل مشكلة المنطقة الحرجة

- Exclusive access to memory location
- Interrupts that can be turned off
  - must have only one processor for mutual exclusion
- **Test-and-Set**: special machine-level instruction

Semaphors السيمافورات

- Synchronization tool that does not require busy waiting.
- Semaphore S – integer variable
- can only be accessed via two indivisible (atomic) operations

wait(S): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;

signal(S): $S := S + 1$;

السيمافورات Semaphors

- و هي تقنية تزامن للوصول بين المعالجات أو العمليات مع تجنب الإنتظار المشغول
- Semaphos S Integer Variable
- يتم الوصول للسيمافور من خلال عمليتين يجب أن تكونا Atomic هما wait و signal ولا يتم الوصول لهما بنفس الوقت من قبل أكثر من معالجة

wait(*S*): **while** $S \leq 0$ **do** *no-op*;
$S := S - 1$;

signal(*S*): $S := S + 1$;

- Too much milk:

Thread A	Thread B
milk.P( );	milk.P( );
if (!haveMilk)	if (!haveMilk)
buy milk;	buy milk;
haveMilk=true;	haveMilk=true;
milk.V( );	milk.V( );

  - "haveMilk" is a Boolean variable
  - "milk" is a semaphore initialized to 1

- Execution:

After:	milk	queue	A	B
	1			
A: milk.P( );	0		in CS	
B: milk.P( );	-1	B	in CS	waiting
A: milk.V( );	0		finish	ready, in CS
B: milk.V( );	1			finish

إستخدام السيمافور كتقنية تزامن بين معالجتين

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

p1	p2	p3	p4
Wait	Wait	Wait	Wait
Cs	Cs	Cs	Cs
signal	signal	signal	signal