fork ⇔ Process Tree

```
int fib(n){
if (n==0 || n==1)
return n;
return fib(n-1) + fib(n-2);
}
```

```
int fibfork(int n){
int ret1, ret2;
if (n == 0 || n == 1)
return n; /* end recursion */
if (fork() == 0)
ret1 = fibfork(n-1); /* first child */
if (fork() == 0)
ret2 = fibfork(n-2); /* second child */
/* parent */
wait();
wait();
return ret1 + ret2;
}
```

Write a function fibfork that implements the Fibonacci using recursion (with new processes created during each recursive step)

Recursive Factorial

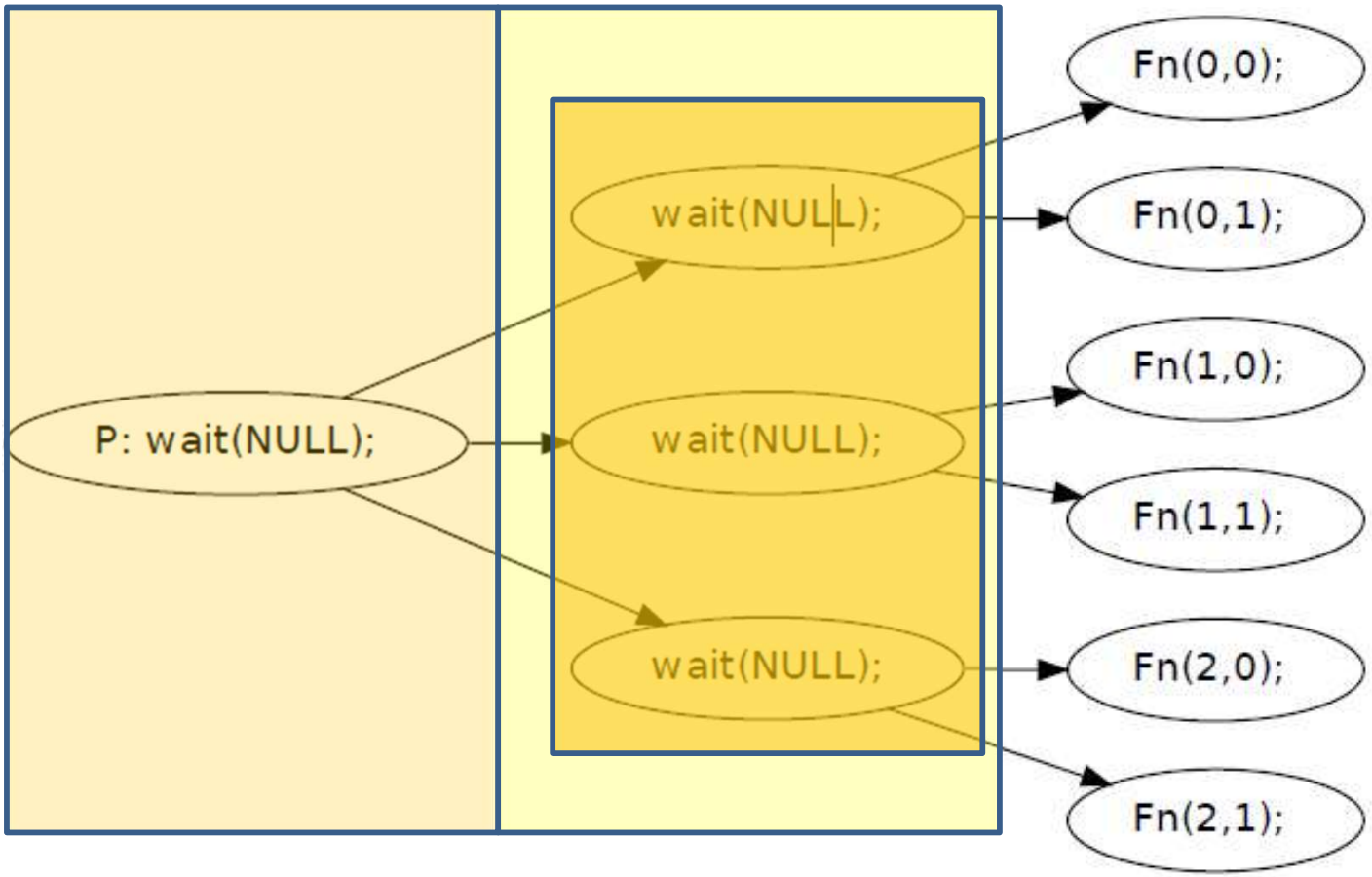
```
int fac (n){
return fac(n-1) * fac(n-2);
}
```

```
int facfork(int n){
int ret1, ret2;
if (fork() == 0)
ret1 = facfork(n-1); /* first child */
if (fork() == 0)
ret2 = facfork(n-2); /* second child */
/* parent */
wait();
wait();
return ret1 + ret2;
}
```

Write a function facfork that implements the factorial using recursion (with new processes created during each recursive step)

Code with fork ⇔ Process Tree

```
for (i = 0; i < 3; i++) {
  ret = fork();
  if (ret == 0) {
    for (j = 0; j < 2; j++) {
      ret = fork();
        if (ret == 0) {
          Fn(i, j);
          exit(0);
        }
    }
    for (j = 0; j < 2; j++)
      wait(NULL);
  }
}
for (i = 0; i < 3; i++)
wait(NULL);
```



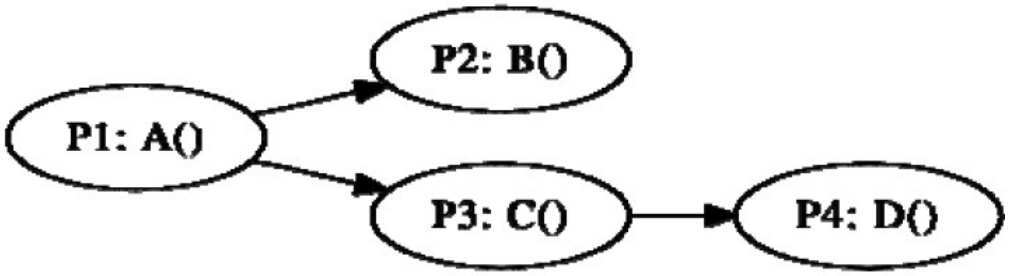
Draw the process tree for the given code or vice versa (write the function with the arguments that defines the given process tree)

Process Tree ⇔ Code with fork

```
ret = fork();
if (ret == 0){
  B();
  assert(0); // should never reach here
}

ret = fork();
if (ret == 0){
  ret = fork();
  if (ret == 0){
    D();
    assert(0); // should never reach here
  }
  C();
  assert(0); //should never reach here
}

A();
assert(0); // should never reach here
```



Sync Two Threads with Semaphores

T_0	T_1
$A_0()$;	$A_1()$;
$B_0()$;	$B_1()$;

Add semaphores (post/signal, wait) so that

- B0() is executed after A1()
- B1() is executed after A0()

```
SemA = Semaphore(0);
SemB = Semaphore(0);

T0():
    A0();
    signal(SemB);
    wait(SemA);
    B0();
```

```
T1():
    A1();
    signal(SemA);
    wait(SemB);
    B1();
```

N Threads

Μοιραζόμενες μεταβλητές:

```
count = 0;
lock = Semaphore(1);
barrier = Semaphore(0);
```

Κώδικας νήματος i:

```
Ai();

wait(lock);
count = count + 1;
signal(lock);

if (count == N)
      signal(barrier);

wait(barrier);

Bi();
```

Correct? Possible Deadlock?

N Threads - Deadlock

T_0	T_1	
`A0();`		
`wait(lock);`		
`count++`		$count = 1$
`signal(lock);`		
`wait(barrier);`		$count \neq 2$
	`A1();`	
	`wait(lock);`	
	`count++`	$count = 2$
	`signal(lock);`	
	`signal(barrier);`	$count == 2$
`B0();`		
	`wait(barrier);`	

N Threads

Is there a way that algorithm terminates without a deadlock?

T_0	T_1	
`A0();`		
`wait(lock);`		
`count++`		$count = 1$
`signal(lock);`		
	`A1();`	
	`wait(lock);`	
	`count++`	$count = 2$
	`signal(lock);`	
`signal(barrier);`		$count == 2$
	`signal(barrier);`	$count == 2$
`wait(barrier);`		
	`wait(barrier);`	

N Threads

```
count = 0;
lock = Semaphore(1);
barrier = Semaphore(0);
```

```
Ai();

wait(lock);
count = count + 1;
signal(lock);

if (count == N)
     signal(barrier);

wait(barrier);

Bi();
```

```
wait(lock);
count++;
signal(lock);

if (count == N);
     signal(barrier);

wait(barrier);
signal(barrier);
```

The new solution

Scheduling - SJF *Preemptive (1 CPU, 1 I/O Dev)*

	P1	*P2*	*P3*
Χρόνος Άφιξης	*0*	*2*	*4*
Ξεσπάσματα	*ΚΜΕ: 10* *Ε/Ε: 10*	*ΚΜΕ: 6* *Ε/Ε: 5* *ΚΜΕ: 2* *Ε/Ε: 2*	*ΚΜΕ: 1* *Ε/Ε: 1* *ΚΜΕ: 2* *Ε/Ε: 3*

t	Ουρά
0	P1/10
2	P1/8,
8	P1/8,
31	–

t	Ουρά ΚΜΕ	ΚΜΕ	Ουρά Ε/Ε	Ε/Ε
0	$P_1/10$	P_1		-
2	$P_1/8, P_2/6$	P_2	-	-
4	$P_1/8, P_2/4, P3/1$	P_3	-	-
5	$P_1/8, P_2/4$	P_2	$P_3/1$	P_3
6	$P_1/8, P_2/3, P_3/2$	P_3	-	-
8	$P_1/8, P_2/3$	P_2	$P_3/3$	P_3
11	$P_1/8$	P_1	$P_2/5$	P_2
16	$P_1/3, P_2/2$	P_2	-	-
18	$P_1/3$	P_1	$P_2/2$	P_2
20	$P_1/1$	P_1	-	-
21	-	-	$P_1/10$	P_1
31	-	-	-	-

	Ε/Ε
	–
	?
	P3
	–

SJF *Preemptive (2 KME, 1 EE)*

	P1	*P2*	*P3*
Χρόνος Άφιξης	*0*	*2*	*4*
Ξεσπάσματα	*KME: 10* *E/E: 10*	*KME: 6* *E/E: 5* *KME: 2* *E/E: 2*	*KME: 1* *E/E: 1* *KME: 2* *E/E: 3*

t	Ουρά ΚΜΕ	ΚΜΕ #1	ΚΜΕ #2	Ουρά Ε/Ε	Ε/Ε
0	$P_1/10$	P_1	-		-
2	$P_1/8, P_2/6$	P_2	P_1	-	-
4	$P_1/6, P_2/4, P_3/1$	P_3	P_2	-	-
5	$P_1/6, P_2/3$	P_2	P_1	$P_3/1$	P_3
6	$P_1/5, P_2/2, P_3/2$	P_2	P_3	-	-
8	$P_1/5$	P_1	-	$P_3/3, P_2/5$	P_2
13	$P_2/2$	P_2	-	$P_1/10, P_3/3$	P_3
15	-	-	-	$P_2/2, P_1/10, P_3/1$	P_3
16	-	-	-	$P_2/2, P_1/10$	P_1
18	-	-	-	$P_2/2, P_1/8$	P_1
26	-	-	-	$P_2/2$	P_2
28	-	-	-	-	-

t	Ε/Ε
0	–
2	?
8	P3
31	–

SJF *Premptive – Utilization Metrics*

What is the utilization rate of the previous two cases. Why is the time not half, despite using 2 CPUs (= 2 KME) and the same 1 I/O device (= 1 E/E)?

i.

$$u_0 = \frac{21}{31} = .68$$

ii.

$$u_0 = \frac{15}{28} = .54$$

$$u_1 = \frac{6}{28} = .21$$

iii.

Page Faults

A process accesses these page numbers: **1, 3, 4, 3, 6, 1, 3, 4, 3.** Assuming 3 frames that are initially empty, compute the number of page faults for **optimal algorithm**

- Βέλτιστή

Αναφορά	Διαθέσιμα πλάισια μνήμης	σφάλμα
1	1	NAI
3	3,1	NAI
4	4,3,1	NAI
3	4,3,1	
6	6,3,1	NAI
1		
3		
4	4,3,1	NAI
3		

5

Page Faults

A process accesses these page numbers: : **1, 3, 4, 3, 6, 1, 3, 4, 3.** Assuming 3 frames that are initially empty, compute the number of page faults for FIFO algorithm

- FIFO

Αναφορα	Διαθεσιμα πλαισια μνημης	σφαλμα
1	1	NAI
3	3,1	NAI
4	4,3,1	NAI
3	4,3,1	
6	6,4,3	NAI
1	1,6,4	NAI
3	3,1,6	NAI
4	4,3,1	NAI
3	4,3,1	

7

Page Faults

A process accesses these page numbers: : 1, 3, 4, 3, 6, 1, 3, 4, 3. Assuming 3 frames that are initially empty, compute the number of page faults for LRU algorithm.

- LRU

Αναφορά	Διαθέσιμα πλάισια μνήμης	σφάλμα
1	1	NAI
3	3,1	NAI
4	4,3,1	NAI
3	3,4,1	
6	6,3,4	NAI
1	1,6,3	NAI
3	3,1,6	
4	4,3,1	NAI
3	3,4,1	
		6

Lifecycle of a Process

Ένα ΛΣ διακρίνει τις εξής καταστάσεις για τις διεργασίες του: **Σε αναμονή, Έτοιμη, Υπό εκτέλεση.** Μία διεργασία που εκτελείται προκαλεί σφάλμα σελίδας και τα δεδομένα της σελίδας χρειάζεται να ανακληθούν από το δίσκο σελιδοποίησης.

Σε τι κατάσταση περνά η διεργασία έως ότου έρθουν τα δεδομένα από το δίσκο;

Όσο περιμένει τα δεδομένα από το δίσκο είναι σε αναμονή, γιατί δεν μπορεί να χρησιμοποιήσει την ΚΜΕ.

Σε τι κατάσταση θα περάσει η διεργασία όταν η σελίδα έρθει τελικά στην κεντρική μνήμη; Δικαιολογήστε *σύντομα τις απαντήσεις σας.*

Όταν έρθει η σελίδα στη μνήμη, μπορεί πλέον να χρησιμοποιήση την ΚΜΕ και θα γίνει Έτοιμη.

Copy-on-Write

Έστω ΛΣ το όποιο για εκτέλεση προγραμμάτων χρησιμοποιεί το μοντέλο **fork()/exec(). Σχολιάστε την επίδραση της «Αντιγραφής κατά την εγγραφή»** (Copy-on-Write) στην επίδοση της παραπάνω λειτουργίας.

Η διεργασία-παιδί που δημιουργείται από τη fork() αποτελεί αντίγραφο της διεργασίας-πατέρας. Ωστόσο, τα δεδομένα της διεργασίας δεν πρόκειται να χρησιμοποιηθούν διότι η exec() τα ακυρώνει (γράφει νέα). Η τεχνική COW αποτρέπει την αντιγραφή των δεδομένων μέχρι να γίνει κάποια εγγραφή και κατα συνέπεια αποφεύγει την άχρηστη αντιγραφή.